Verification of Concurrent Systems with VerCors

Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski

University of Twente, the Netherlands

Abstract. This paper presents the VerCors approach to verification of concurrent software. It first discusses why verification of concurrent software is important, but also challenging. Then it shows how within the VerCors project we use permission-based separation logic to reason about multithreaded Java programs. We discuss in particular how we use the logic to use different implementations of synchronisers in verification, and how we reason about class invariance properties in a concurrent setting. Further, we also show how the approach is suited to reason about programs using a different concurrency paradigm, namely kernel programs using the Single Instruction Multiple Data paradigm. Concretely, we illustrate how permission-based separation logic is suitable to verify functional correctness properties of OpenCL kernels. All verification techniques discussed in this paper are supported by the VerCors tool set.

1 Introduction

The quest for software correctness is as old as software itself, and as the complexity of software is steadily increasing, also the challenges to guarantee software correctness are increasing. In the 60-ies, Floyd and Hoare for the first time proposed static techniques to guarantee that a program functioned as it was supposed to do [28,24]. For a long time, their ideas remained mainly theoretical, but during the last decade or so, we have seen a dramatic increase in the applicability of software verification tools for sequential programs. Several successful tools and techniques in this area are Dafny [39], Spec# [6], ESC/Java2 [14], OpenJML [17], KeY [7], and KIV [54].

However, this development is not sufficient to guarantee correctness of all modern software. In particular, most modern software is inherently multithreaded – as this is often necessary to efficiently exploit the underlying multi-core hardware – and often also distributed. This shift in software development has also led in a shift of software verification technology: several groups are working on tools and techniques to reason about multithreaded software. However, at the moment, many of these techniques are still difficult to apply and require expert knowledge about the underlying theory.

In this paper, we describe the current results of the VerCors project. Goal of the VerCors project is to use the advance in program verification technology for multithreaded programs to develop a practical and usable verification technology, that can also be used by non-verification-experts, and in particular by experienced software developers.

The basis for the VerCors approach to the verification of multithreaded programs is the use of *permission-based separation logic*. Separation logic [55] is an extension of Hoare logic that was originally developed to reason about pointer programs. In contrast to classical Hoare logic, separation logic explicitly distinguishes between the heap and the store. In particular, this means that one can explicitly express that one has a pointer into a certain location at the heap. In addition, separation logic uses the *separating conjunction* $\star$ to combine formulas. A formula $\phi_1 \star \phi_2$ is valid for a heap h, if the heap h can be separated into two *disjoint* parts h_1 and h_2, such that ϕ_1 is valid for the heap h_1, and ϕ_2 is valid for the heap h_2.

It was soon realised that separation logic is also suitable to reason about multithreaded programs. If threads operate on disjoint parts of the heap, they can be verified in isolation, and there is no need to explicitly consider interferences between the two threads [46]. However, classical separation logic in itself is not flexible enough to verify all interesting multithreaded programs. In particular, classical separation logic does not allow two threads to simultaneously read a shared location, even though this is perfectly acceptable for a multithreaded program. Therefore, we combine separation logic with the notion of *access permissions* [12]. Within the logic, one can express that a thread has *read* or *write* permission on a location. Permissions can be transferred between threads at synchronisation points, including thread creation and joining (i.e., waiting for a thread to terminate). Soundness of permission-based separation logic ensures that (1) there always is at most one thread that has write permission on a location, and (2) if a thread has read permission, then all other threads also only can have read permission. This implies that if a program can be verified with this logic, it is free of data races. Moreover, it also allows each thread to be verified in isolation, because when a thread has permission to access a location, its value is *stable*. If a thread has write permission, it is only this thread that can change this location. If a thread has read permission, all other threads also only have read permissions, and thus the value stored in this location cannot be changed. As a consequence, there is no need to explicitly check for non-interference freedom (in contrast to classical verification methods for concurrent programs, such as Owicki-Gries [48]).

The concrete program annotation language that we use is an extension of the JML annotation language [13]. JML, the Java Modeling Language, is a behavioural interface specification language for Java programs. It allows to specify methods by pre- and postconditions (**`requires`** and **`ensures`** clauses, respectively). Additionally, it also supports class level specifications, such as class invariants and history constraints. JML is widely supported for sequential Java, with tools for static and run-time verification [13], annotation generation [23], etc. To make sure that all this work for sequential Java can be easily reused in a concurrent setting, the VerCors annotation language combines JML with support for permissions and separation.

This paper introduces the full details of our logic and our Java program annotation language, and illustrates this on several examples. All examples are

verifiable using the VerCors tool set, which underlies all our work. The on-line version of the tool set as well as our library of examples are reachable through the project's home page [58]. The VerCors tool set encodes the verification problem of programs annotated with our specification language, into verification problems of existing verification tools, such as Chalice [37] and Boogie [38]. This allows us to leverage existing verification technology, instead of rebuilding everything from scratch.

Compared to other projects working on verification techniques for concurrent programs, the VerCors project distinguishes itself because it provides support to reason about different synchronisation mechanisms in a uniform way. Moreover, it also provides support to reason about functional properties. In a concurrent setting, it becomes more difficult to specify functional behaviour, because properties that hold for a thread in one state, might be invalidated by another thread in the next state. Only properties for which a thread holds sufficient permissions cannot be invalidated. This impacts what sort of properties can be specified for multithreaded programs. However, we show that it is possible to reason about class invariants in a concurrent setting. In particular, class invariants may be temporarily broken, provided no thread is able to observe that the invariant is broken. This technique allows one to specify and verify many meaningful reachability properties of objects in concurrent programs.

The last part of this paper discusses how permission-based separation logic also can be used to verify programs in a different concurrency paradigm: we show how it is used to verify OpenCL kernels [47]. These are a typical example of vector programs, where multiple threads execute the same instruction but on different data. Essentially, the approach distinguishes two levels of specifications: for each vector program a complete collection of permissions used by the program is specified. Additionally, for each thread the permissions necessary for that thread are specified, and it should be shown that the permissions used by the different threads together are no more than the permissions available for the complete vector program. Synchronisation between the threads in a vector program is done by a barrier; the specifications specify how at each barrier, the permissions can be redistributed between the threads. Finally, for this kind of programs, it is also possible to prove functional correctness, i.e., one can specify and verify what is computed by the program.

Overview of the Paper The remainder of this paper is organised as follows. Section 2 presents our version of permission-based separation logic, illustrates how one can reason with it, and how this is supported by the VerCors tool set. Next, Sect. 3 discusses how we can specify different synchronisation mechanisms, while Sect. 4 discusses class invariants in a concurrent setting. Then, Sect. 5 discusses how the approach is used to reason about kernel programs. Finally, Sect. 6 discusses related work, while Sect. 7 concludes and sketches our ideas for future work.

Origins of the Material More information about the details of our logic (described in Section 2 is published in LMCS [27]. An overview and architecture

of the VerCors tool set is published at FM'2014 [10]. Our uniform specifications of the different synchronisation mechanisms have appeared in PDP'14 [2]. The modular specification and verification technique for concurrent class invariants is published in FASE'14 [60]. Finally, the verification approach for OpenCL kernels is published in Bytecode'13 and in SCP [30,11].

2 Permission-based Separation Logic for Concurrent Programs

Before precisely defining the VerCors property specification language, we first give some background on separation logic and permissions in general. This description is mainly intuitive, and serves as background information.

2.1 Classical Separation Logic

The two main ingredients of formulas in classical separation logic are the points-to predicate $\mathsf{PointsTo}(x, v)$ (often written as $x \mapsto v$), and the separating conjunction $\star$. In the remainder of this paper, however, we will be using a different symbol for the separating conjunction, namely a double star $**$. This is necessary to distinguish it from the multiplication operator of Java, whose meaning we want to retain in our JML-compatible specification language. A formula $\mathsf{PointsTo}(x, v)$ intuitively is valid for a heap h if the variable x points to a location that is in the domain of this heap h, and this location contains the value v. As explained above, a separating conjunction is valid for a heap h if the two conjuncts are valid for disjoint parts of this heap.

The verification rules in classical separation logic for look-up and update of a location on the heap contain an explicit precondition $\mathsf{PointsTo}(x, v)$, as follows:

$$\frac{\text{local variable } y}{\{\mathsf{PointsTo}(x, v)\}y := x\{\mathsf{PointsTo}(x, v) ** y = v\}}\ (\text{look-up})$$

$$\frac{}{\{\mathsf{PointsTo}(x, \_)\}x := v\{\mathsf{PointsTo}(x, v)\}}\ (\text{update})$$

This means that the $\mathsf{PointsTo}$ predicate also serves as an access permission: the location on the heap can only be read or written if the program fragment actually has a reference to it. This is in contrast to classical Hoare logic, where any variable can be read or written.

2.2 Concurrent Separation Logic

When separation logic was introduced to reason about programs with pointers, it was quickly realised that it would also be suitable to reason about concurrent programs. In particular, since separation logic requires explicit access to the fields on the heap that it reads and updates, the *footprint* of each thread is known. If two threads have disjoint footprints, they work on different parts of the heap, and thus their behaviours do not interfere.

O'Hearn was the first to use this idea, and to propose *Concurrent Separation Logic (CSL)*. A major ingredient of CSL was a rule for reasoning about concurrent programs [46]. Given a collection of n parallel threads,

- if each thread i can be specified (and verified) with a pre- and postcondition P_i and Q_i, respectively,
- if all preconditions are disjoint (w.r.t. the heap), and
- all postconditions are disjoint,

then they can be combined to verify the complete parallel program. This is expressed by the following rule:

$$\frac{\{P_1\}S_1\{Q_1\} \quad \dots \quad \{P_n\}S_n\{Q_n\}}{\{P_1 \mathbin{**} \dots \mathbin{**} P_n\}S_1 \mid\mid \dots \mid\mid S_n\{Q_1 \mathbin{**} \dots \mathbin{**} Q_n\}} \quad \begin{array}{l}\forall ij.i \neq j.\mathsf{var}(P_i) \cup \mathsf{var}(Q_i) \\ \text{not modified in } S_j\end{array}$$

O'Hearn calls this the *disjoint concurrency rule*.

For concurrent programs, an important property is that a program is free of *data races*. A data race occurs when two threads potentially might access the same location simultaneously, and at least one of the two accesses is a write. Clearly, when a program is verified using the rule for disjoint concurrency, it is free of data races.

2.3 Permissions and Resources

However, it is also easy to see that the rule is overly restrictive. Any program where two threads might *read* the same variable simultaneously cannot be verified with this rule.

To solve this problem, *fractional permissions* are introduced to specify the access that a thread requires to a location in a more fine-grained way. A fractional permission is a fraction in the interval (0,1] [12]. A permission with value 1, i.e., a full permission is understood as a permission to *write* a location; any permission with a value less than 1, i.e., a fractional permission, only gives access to *read* a location.

In the specifications, permissions are explicitly added to the assertions. In particular, the PointsTo operator is decorated with a fractional permission in the interval $(0, 1]$, such that $\mathsf{PointsTo}(x, v, \pi)$ means that the variable x has access permission π to a location on the heap, and this location contains the value v.

Once permissions are introduced, the PointsTo predicate can be separated into two parts:

$$\mathsf{PointsTo}(x, v, \pi) \equiv \mathsf{Perm}(x, \pi) \mathbin{**} x = v$$

Thus, $\mathsf{Perm}(x, \pi)$ means that a thread holds an access permission π on location x. In our approach, we use the Perm operator as the primitive operator. As a consequence, in our annotation language, it has to be checked explicitly that all formulas are *self-framed*, i.e., only properties can be expressed for which one has appropriate access conditions. This is crucial to maintain soundness of the approach. The essential feature of fractional permissions that enables the flow of

permissions between the different threads is that they can be split and combined. Concretely, $\mathsf{Perm}(x, \pi)$ can be exchanged for $\mathsf{Perm}(x, \pi/2)$ ** $\mathsf{Perm}(x, \pi/2)$ and vice versa.

In the verification rules for update and look-up, the permissions are explicitly added to the precondition (and returned in the postcondition). Then, in the context of Java that we are slowly moving to, memory locations are referred to by a combination of object references and field expressions, i.e., they are of the form $e.f$ (cf. Parkinson's separation logic for Java [49]). Furthermore, in Java objects can be dynamically allocated with the **new** operator. In this case, the allocation command returns an initial full write permission of 1 on all fields of the newly created object. Thus, in permission-based separation logic for Java, one has the following rules for update, look-up, and allocation, respectively:

$$\frac{}{\{\mathsf{Perm}(e.f, 1)\}e.f := v\{\mathsf{Perm}(e.f, 1) \mathbin{**} e.f = v\}} \text{ (update)}$$

$$\frac{\text{local variable } y}{\{\mathsf{Perm}(e.f, \pi) \mathbin{**} e.f = v\}y := e.f\{\mathsf{Perm}(e.f, \pi) \mathbin{**} y = v\}} \text{ (look-up)}$$

$$\frac{}{\{\mathsf{true}\}e := \texttt{new } \mathsf{C}()\{\mathsf{Perm}(e.f_1, 1) \mathbin{**} \ldots \mathsf{Perm}(e.f_n, 1)\}} \text{ (allocate)}$$

where $f_1, \ldots, f_n$ are the fields of class C.

The soundness proof of the verification rules ensures an additional global property on the permissions in the system, namely that the total number of permissions to access a location simultaneously *never* exceeds 1. This ensures that any program that can be verified is free of data races. If a thread holds a full permission to access a location, there can never be any other thread that holds a permission to access this location simultaneously. If a thread holds a read permission to access a location, then any other thread that holds a permission simultaneously must also have a read permission only. Thus, there can never be conflicting accesses to the same location, where one of the accesses is a write, thus a verified program is free of data races.

Abstract Predicates Another commonly-used extension of separation logic are *abstract predicates* [50]. An important purpose of abstract predicates is to add inductive definitions to separation logic formulas, making it possible to define and reason about permissions on linked data structures. Abstract predicates can also be used to provide control over the visibility of specifications, which allows one to encapsulate implementation details. Another feature of abstract predicates is that they can be declared without providing a definition (similar to abstract methods in Java). This allows one to use abstract predicates as a token in specifications. This feature is used for example to specify behaviour of a program as an abstract state machine, e.g., to specify mutual exclusion.

Since abstract predicates can be a token, without a predicate body, they define more than just a set of access permissions. Therefore, we will use the term *resource* when referring to abstract predicates and/or access permissions.

```
public class Point {
  private int x, y;
  //@ invariant (x >= 0 && y >= 0) || (x <= 0 && y <= 0);

  //@ requires Perm(this.x, 1) ** Perm(this.y, 1);
  //@ ensures Perm(this.x, 1) ** Perm(this.y, 1);
  public void set(int xv, int yv){ this.x = xv; this.y = yv; }
}
```

Lst. 1. Class Point.

Abstract predicates can have parameters, which can be program variables or (fractional) permissions. The latter can be used for example to specify different access permissions to different parts of a data structure. However, many separation logic tools do not support reasoning about abstract predicates with arbitrary parameters.

2.4 The VerCors Property Specification Language

As mentioned above, the property specification language that we use in the VerCors tool set combines separation logic with features from the Java Modeling Language (JML).

Example 1. Lst. 1 shows a simple example of a Java class `Point`. Below, we will use this class with extensions and variations as a running example. The class `Point` encapsulates values for a point in a 2D Cartesian coordinate. The contract of the method `set` specifies that write permissions on both `x` and `y` are required to execute this method. Moreover, when the method is finished, the same permission will be given back to the caller. As a functional property we add a requirement that every point is always in the first or the third quarter of the Cartesian space, we do this with the **`invariant`** clause.

More formally, in our VerCors property specification language we distinguish between *resource expressions* (R, typical elements r_i) and *functional expressions* (E, typical elements e_i), with the subset of logical expressions of type boolean (B, typical elements b_i). The grammar for our specification language is the following:

$$
\begin{aligned}
R ::= {} & b \\
& \mid \mathsf{Perm}(e.f, \pi) \\
& \mid (\texttt{\textbackslash forall* T v}; b; r) \\
& \mid r_1 \texttt{ ** } r_2 \\
& \mid r_1 \texttt{ -* } r_2 \\
& \mid b_1 \texttt{ ==> } r_2 \\
& \mid e.\mathsf{P}(e_1, \ldots, e_2) \\
E ::= {} & \text{any } \textit{pure} \text{ expression} \\
B ::= {} & \text{any } \textit{pure} \text{ expression of type boolean}
\end{aligned}
$$

where `T` is an arbitrary type, `v` is a variable name, `P` is an abstract predicate of a special type **`resource`**, f is a field reference, and π denotes a fractional permission in the range $(0, 1]$.

The permission property `Perm` and the separating conjunction `**` have been discussed above. Additionally, in our specifications, we use the separating implication `-*`. A formula ϕ_1 `-*` ϕ_2 is valid for a heap h if for any heap h' for which ϕ_1 is valid, the formula ϕ_2 is valid for the heap h `**` h' (where h `**` h' denotes the heap composed of h and h'. Thus, in other words, given a heap that satisfies the formula ϕ_1, this can be exchanged for a combined heap that satisfies the formula h'. Then, we also allow the separating quantification **`\forall*`**, which is essentially an iterative separating conjunction. We have the standard logical connectives to combine first-order formulas, and guarded resource expressions denoted by `==>`. This is used to state conditional resource properties, the most typical condition being non-nullness of an object reference that is used in the following resource formula, for example, `x != null ==> Perm(x, 1)`. Finally, we can refer to any predicate `P` declared and/or defined inside any Java classes.

The grammar above defines the language that can be used in all specification clauses of a Java program annotated with JML. As we already mentioned, JML allows one to state pre- and postconditions for methods with the **`requires`** and **`ensures`** keywords, respectively, or class invariants with the **`invariant`** keyword. Additionally, we extend the syntax of JML to allow one to declare and define predicates. Each predicate is declared and defined by its signature, name and an optional body in a class-level JML comment. Below, we will also use a special class of predicates, called *groups*. Group predicates are splittable over their permission predicates; why we need them and a more precise definition are discussed in Section 2.5 below, when discussing the specification of thread joining. Additionally, our syntax also allows one to declare *ghost class and method parameters*, i.e., specification-only class and method parameters. Ghost class and method parameters are specified using a **`given`** clause (for input parameters), and a **`yields`** clause for result values. Classes can only have ghost input parameters, methods can have both ghost input and output parameters.

Example 2. Lst. 2 extends the example from Lst. 1 with a predicate to encapsulate the permissions on the fields. Additionally, it adds two methods, which given any permission on the fields read the values and perform their defined tasks.

The annotations for methods `plot` and `getQuarter` express that given any fractional permission `p` on both `x` and `y`, the object can plot and identify the quarter, respectively. Additionally, the method specifications ensure that the required permissions are returned. Clearly, given any fraction $0 < p \leq \frac{1}{2}$, two threads can simultaneously execute `plot` and `getQuarter` on the same point object. However, two threads cannot execute `set` on the same point object simultaneously.

2.5 Reasoning about Dynamic Threads

As mentioned above, permissions are transferred between threads upon synchronisation. In the next section we will look at how we specify different syn-

```
public class Point{
  //@ resource state(frac p) = Perm(this.x, p) ** Perm(this.y, p);
  private int x, y;

  //@ requires state(1); ensures state(1);
  public void set(int xv, int yv){ this.x = xv; this.y = yv; }

  //@ given frac p; requires state(p); ensures state(p);
  public void plot(){ /* plot the point on the screen */ }

  //@ given frac p; requires state(p); ensures state(p);
  public int getQuarter(){ /* return 1..4 to show the quarter. */ }
}
```

Lst. 2. Extended class of `Point`.

chronisation mechanisms in a uniform way. Here we discuss one special kind of synchronisation between threads, namely thread start and thread termination.

In Java, threads are objects. When the native `start` method is invoked on a thread object, the virtual machine will create a new thread of execution. This new thread will execute the `run` method of the thread object. The new thread will remain alive until it reaches the end of its `run` method. Other threads can wait for a thread to terminate. They do this by invoking the `join` method on a thread object. This will block the calling thread until the joined thread has terminated.

In Java, the correct way of using a thread is to first call `start` precisely once and then call `join` as many times as one would like. To enforce this order, the constructor of the `Thread` class ensures a `start` token that is required by the `start` method. The start method in turn ensures a `join` token. This join token has a fraction as argument and is defined as a group, so it can be shared between threads. To specify what permissions are transferred when threads are created and joined, we use the specification of the `run` method: the precondition of a thread is the precondition of the `run` method; the postcondition of a thread is the postcondition of the `run` method. For this purpose, we specify predicates `preFork` and `postJoin` that denote this pre- and postcondition, respectively. These predicates have trivial definitions to be extended in the thread implementing classes. Thus, we specify that the `start` method requires both the `start` token and the resources specified in `preFork` and gives them all up, i.e., it has postcondition `join(1)`. Finally, the specification of the `join` method ensures that the resources specified in the `postJoin` predicate are obtained by executing this method. Notice that the `postJoin` predicate should be a group. Below, we give the definition of a group and describe the extended role of the `join` predicate in the precondition. This results in the following specification for class `Thread` in Lst. 3.

Every class that defines a thread extends class `Thread`. It can extend the predicates `preFork` and `postJoin` to denote extra permissions that are passed to the newly created thread. To verify that the thread functions correctly, the `run`

```
class Thread implements Runnable {

  //@ resource start();
  //@ resource preFork() = true;
  //@ group resource postJoin(frac p) = true;
  //@ group resource join(frac p);

  //@ requires true; ensures start();
  public Thread();

  //@ requires preFork(); ensures postJoin(1);
  void run();

  //@ requires start() ** preFork(); ensures join(1);
  public void start();

  //@ given frac p;
  //@ requires join(p); ensures postJoin(p);
  public void join();
}
```

Lst. 3. Specification of class Thread.

method is verified w.r.t. its specification. When verifying the thread that creates or joins this thread, the calls to start and join are verified using the standard verification rule for method calls.

Example 3. To illustrate how we reason about dynamic thread creation, we use a common pattern of signal-processing applications in which a chain of threads are connected through a shared buffer, in which we store several instances of class Point defined in Lst. 2. In addition, this example also demonstrates how the join predicate is used in case multiple threads join the same thread.

The complete application uses one shared buffer and four threads: a sampler, filter processes A and B, and a plotter. The buffer encapsulates an input field and two points, see Lst. 4. First, the sampler thread assigns a value to the input field of the buffer. Next, it passes the buffer to processes A and B, which are executed in parallel. Based on the value that the sampler thread stored in the inp field of Buffer, each process calculates a point and stores its value in the

```
public class Buffer {
  //@ resource state(frac p) = Perm(inp, p) ** Perm(outa, p) ** Perm(outb, p);
  public int inp;
  public Point outa, outb;
}
```

Lst. 4. Class Buffer.

shared buffer. Finally, the computation results of both processes are displayed by the plotter.

What makes this example interesting is that both processes A and B join the sampler thread, i.e., they wait for the sampler thread to terminate, and in this way retrieve read permission on the input data that was written by the sampler thread.

In addition, the plotter waits for the two processing threads to terminate (by joining them), to retrieve their permissions on the shared buffer, and then combines these into full write permissions on all fields of the shared buffer.

Lst. 5 shows the sampler thread, Lst. 6 shows the `AFilter` class (class `BFilter` is similar and not shown here), Lst. 7 shows the `Plotter` class and finally Lst. 8 shows the main application. In the examples we sketch an outline of the correctness proof as comments in the code. We also indicate when predicates are **`fold`**-ed and **`unfold`**-ed, to encapsulate predicate definitions and expand them, respectively.

To understand the annotations, we need to explain the meaning of the `join` predicate, and why the `postJoin` predicate should be a group. In Example 3, both processes A and B join the sampler thread. If both joining threads would obtain the full set of permissions specified in `postJoin`, this would lead to unsoundness, because multiple threads would obtain a write permission on the same location simultaneously.

Instead, the full permission on the input field from the buffer must be split between these two processes. Therefore, a special `join` token predicate is introduced, which holds a fractional permission `p`. This permission specifies which part of the `postJoin` predicate can be obtained by the thread invoking the `join` method. The actual fraction of the `join` token that the joining thread currently holds is passed as an extra parameter to the `join` method, via the **`given`** clause.

However, to make this work, both predicates `postJoin` and `join` have to be splittable w.r.t. this permission. Splittable predicates are called *groups* and are declared with the **`group`** keyword. Formally, a predicate P, parametrised by permission q is a group if it respects the following equivalence $P(q) \mathrel{*\!\!-\!\!*} P(q/2) \mathbin{**} P(q/2)$. That is, the group property transitively extends the splittability of atomic permissions over fractions to predicates.

The thread that creates the thread object obtains the `join` token, containing a full permission. Formally, the `join` token is defined as an abstract predicate without a body. It can be split and distributed as any other permission. The join token is created and returned upon thread construction, see line 14 in Lst. 3.

Example 3 continued. Inside the `main` method (Lst. 8), for each thread a `Join` token is created upon initialisation. The `main` method splits the ticket to join the sampler thread, and transfers each half to the processing threads, as specified by their `preFork` predicates. Additionally, the join tokens for the processing threads are transferred to the plotter.

Thus, the processing thread A (Lst. 6) uses a half join token to join the sampler thread, and to obtain half the resources released by the sampler thread.

```
public class Sampler extends Thread {
  //@ resource preFork = Perm(buffer.inp, 1);
  //@ group resource postJoin(frac p) = Perm(buffer.inp,p);
  Buffer buffer;

  // constructor

  //@ requires preFork(); ensures postJoin(1); // inherited from thread
  public void run(){
    //@ unfold preFork;
    // { Perm(buffer.inp,1) }
    sample();
    // { Perm(buffer.inp,1) }
    //@ fold postJoin(1) ;
 }

  //@ requires Perm(buffer.inp, 1); ensures Perm(buffer.inp, 1);
  private void sample(){
    // fill buffer.inp
  }
}
```

Lst. 5. Class Sampler.

```
public class AFilter extends Thread {
  private Sampler sampler;
  private Buffer buffer;

  //@ resource preFork() = Perm(buffer.outa, 1) ** sampler.join(1/2);
  //@ group resource postJoin(frac p)=Perm(buffer.outa,p)**Perm(buffer.inp,p/2);

  // constructor

  //@ requires preFork(); ensures postJoin(1);
  public void run(){
    //@ unfold preFork; // { Perm(buffer.outa, 1) ** Join(sampler, 1/2) }
    sampler.join(); // { Perm(buffer.outa, 1) ** sampler.postJoin(1/2) }
    //@ unfold sampler.postJoin(1/2);
    // { Perm(buffer.outa, 1) ** Perm(buffer.inp, 1/2) }
    processA(); // { Perm(buffer.outa, 1) ** Perm(buffer.inp, 1/2) }
    //@ fold this.postJoin(1);
  }

  //@ requires Perm(buffer.outa, 1) ** Perm(buffer.inp, 1/2);
  //@ ensures Perm(buffer.outa, 1) ** Perm(buffer.inp, 1/2);
  private void processA(){/* reading buffer.inp and fill buffer.outa. */}
}
```

Lst. 6. Class AFilter.

```
public class Plotter extends Thread {
  private Buffer buffer; private AFilter ta; private BFilter tb;
  //@ resource preFork() = ta.join(1) ** tb.join(1);
  //@ group resource postJoin(frac p) = buffer.state(p);

  //@ ensures start() ** Perm(buffer, 1) ** Perm(ta, 1) ** Perm(tb, 1);
  public Plotter(Buffer buf, AFilter fa, BFilter fb){ buffer=buf; ta=fa; tb=fb; }

  //@ requires preFork(); ensures postJoin(1);
  public void run(){
    //@ unfold preFork // { ta.join(1) ** tb.join(1) }
    ta.join(); // { ta.postJoin(1) ** tb.join(1) }
    tb.join(); // { ta.postJoin(1) ** tb.postJoin(1) }
    //@ unfold ta.postJoin(1); unfold tb.postJoin(1); fold buffer.state(1);
    // { buffer.state(1) }
    plot(); // { buffer.state(1) }
    //@ fold this.postJoin(1);
  }

  //@ requires buffer.state(1); ensures buffer.state(1);
  private void plot(){ /* plots the calculated points from the buffer */ }
}
```

Lst. 7. Class Plotter.

```
//@ requires buf.state(1); ensures buf.state(1);
void main(){
  Sampler s=new Sampler(buf); // { s.join(1) ** buf.state(1) }
  //@ unfold buf.state(1); fold s.preFork()
  //{ s.preFork()**s.join(1)**Perm(buf.outa, 1)**Perm(buf.outb, 1) }
  AFilter a = new AFilter(buf, s);
  //{ s.preFork**s.join(1)**a.join(1)**Perm(buf.outa, 1)**Perm(buf.outb, 1) }
  //@ fold a.preFork;
  //{ s.preFork**a.preFork**s.join(1/2)**a.join(1)**Perm(buf.outb, 1) }
  BFilter b = new BFilter(buf, s);
  //{ s.preFork**a.preFork**s.join(1/2)**a.join(1)**b.join(1)**Perm(buf.outb, 1) }
  //@ fold b.preFork;
  //{ s.preFork**a.preFork**b.preFork**a.join(1)**b.join(1) }
  Plotter p = new Plotter(buf, a, b);
  //{ s.preFork**a.preFork**b.preFork**a.join(1)**b.join(1)**p.join(1) }
  //@ fold p.preFork; //{ s.preFork**a.preFork**b.preFork**p.preFork**p.join(1) }
  s.start(); a.start(); b.start(); p.start(); // { p.join(1) }
  p.join(); /* { p.postJoin(1) } */ //@ unfold p.postJoin(1);
  //@ fold buf.state(1);
}
```

Lst. 8. The main thread for the sampler, filters, and the plotter.

Similarly, the plotter thread obtains a 1/2 read permission on `inp` and a write permission on `outa` by joining process A, and another 1/2 read permission on `inp` and a write permission on `outb` by joining process B. It then combines the read permissions into a write permission on `inp` to invoke its `plot` method.

2.6 Architecture of the VerCors Tool Set

The whole verification approach as outlined above is supported by our VerCors tool set. Rather than building yet another verifier, the VerCors tool leverages existing verifiers. That is, it is designed as a compiler that translates specified programs to a simpler language. These simplified programs are then verified by a third-party verifier. If there are errors then the error messages are converted to refer to the original input code.

Figure 1 shows the overall architecture of the tool. Its main input language is Java. For prototyping, we use the toy language PVL, which is a very simple object-oriented language that can express specified GPU kernels too. The C language family front-end is work-in-progress, but will support OpenCL in the near future. We mainly use Chalice [37], a verifier for an idealised concurrent programming language, as our back-end, but for sequential programs we also use the intermediate program verification language Boogie [38].

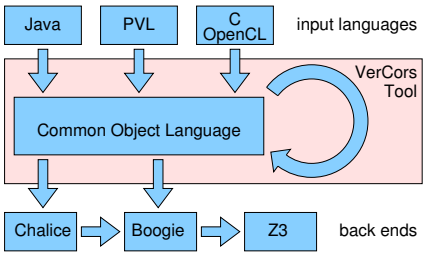

Fig. 1. VerCors tool architecture.

The implementation of the tool is highly modular. Everything is built around the Common Object Language data structure for abstract syntax trees. For Java and C, parsing happens in two passes. In the first pass an existing ANTLR4 [52] grammar is used to convert the programs into an AST while keeping all comments. In the second pass those comments that contain specifications are parsed using a separate grammar. This prevents us from having to maintain heavily modified grammars and makes it much easier to support multiple specification languages. The transformations to encode the program consist of many simple passes. Obviously, this impacts performance, but it is good for reusability and checkability of the passes. Our back-end framework allows switching between different versions, by setting up their command line execution using environment modules, a system for dynamic access to multiple versions of software modules[1].

3 Synchroniser Specifications

3.1 Reasoning about Synchronisers

Another way for threads to synchronise their behaviour is by using a lock. Locks provide a way to protect access to shared data. Only one thread at a time can

[1] `http://modules.sourceforge.net`

hold a lock, thus if threads only access the protected data while holding the lock, this means that there cannot be simultaneous access to the protected data. In Java, there are two ways to declare locks: any object can function as a lock, and be used via the `synchronized` statement. Alternatively, one can also define a lock object, and use the `lock` and `unlock` methods, declared in the `Lock` interface.

To reason about locks, we follow OHearn's approach for CSL and we associate a *resource invariant* with each lock, which specifies access to the data protected by the lock [46]. That is, the resource invariant makes the information about which data is protected by the lock explicit by naming the corresponding memory locations. A thread that acquires the lock, obtains the permissions specified in the resource invariant; when it releases the lock, it also has to release the permissions specified in the resource invariant. A particular challenge for Java is that locks might be *reentrant* [26], i.e., if a thread already holds a lock, it can obtain it once more. This does not change anything in the behaviour, except that to release the lock, it should be unlocked twice as well. Formally, a reentrancy level is maintained for the thread holding the lock. To ensure soundness, a thread should only obtain permissions when it acquires a lock for the first time, and it should only be forced to give up the permissions when it releases the lock for the last time. To manage this properly, the multi-set of locks that a thread holds has to be maintained in the specifications.

Originally, we added explicit verification rules for locks as primitives to the specification language (see [26]). However, this has the drawback that for every synchronisation mechanism, new rules have to be added to the logic. When looking into this in more detail, we realised that also for other synchronisation mechanisms, the notion of resource invariant is crucial to specify what resources can be redistributed between threads upon synchronisation.

Therefore, we took an alternative approach and we lifted the specification of synchronisation mechanisms to the API level of Java, i.e., we provide a *specification-based* approach to reason about locks. To make the approach applicable to different synchronisation mechanisms, we generalise the notion of a lock, i.e., we consider any routine that uses synchronisation to transfer a set of permissions as a *locking routine*. With our approach, we can specify arbitrary synchronisation mechanisms from the Java API in a similar way, and provide the ability to reason with these specifications modularly. We illustrate how our synchroniser specifications are used to verify code using the synchroniser.

In a separate line of work, we have derived program logic rules for atomic operations `set`, `get` and `compareAndSwap` as a synchronisation primitive [1]. We can use these specifications to show that (simplified versions of) Java's reference implementations of the various synchronisers indeed respect our specifications. For more information about these verifications, we refer to our PDP 2014 paper about synchroniser specifications [2].

3.2 Initialisation of Resource Invariants

A lock can only be used when it has been initialised, i.e., the access permissions specified in the resource invariant are stored "into" the lock. This ensures that

```
//@ ghost boolean initialized = false;
//@ group resource initialized(frac p) = PointsTo(initialized, p/2, true);
//@ requires inv(1) ** PointsTo(initialized, 1, false);
//@ ensures initialized(1);
public void commit();
```

Lst. 9. Specifications for lock initialisation.

the resources can be passed to a user upon synchronisation without introducing new resources. Initialisation of the resource invariant is done in the same way for all synchronisation mechanisms: class `Object` declares a ghost boolean field `initialized` that tracks information about the initialisation state of the resource invariant. Newly created locks are not initialised; the specification-only method `commit`, see Lst. 9, can be used by the client code to irreversibly initialise the lock. This means that the resources protected by the lock, as specified in the resource invariant predicate `inv`, become shared. To achieve this, `commit` requires the client to provide the complete resource invariant `inv(1)`, together with an exclusive permission to change `initialized` (line 3). The method consumes the invariant ("stores it into the lock"). Moreover, it ensures that `initialized` cannot be changed any more by consuming part of the permission to access this field, effectively making it read-only (lines 2 and 4). For convenience, the result of `commit` is encapsulated in a single resource predicate `initialized`, which can be passed around and used as a permission ticket for locking operations, see below. The default location for the call to `commit` is at the end of the constructor of the synchronisation object. More complex lock implementations (which are not discussed in this paper) may require moving this call to another location in the program.

The actual resource invariant is typically decided by the user of the synchronisation class, therefore it is passed as a class parameter with the type `(frac -> resource)`. For example, given a two-point coordinate class, such as in Lst. 1, using a `ReentrantLock`, the resource invariant that protects the `x` coordinate (only) is specified with `xInv`, which is passed both as a type parameter and during instantiation of the lock. By adding it as a type parameter it is specified that the declared local variable or field can only contain lock that use this particular invariant. By adding the argument during instantiation and object that has this particular invariant is created. For example:

```
//@ resource xInv(frac p) = Perm(x, p);
Lock/*@<xInv, ...>@*/ xLock=new ReentrantLock/*@< xInv >@*/();
```

As mentioned in Sect. 2, in our specifications such parameters (of which there will be more, hence the "...") are received through parameters specified with the **given** keyword.

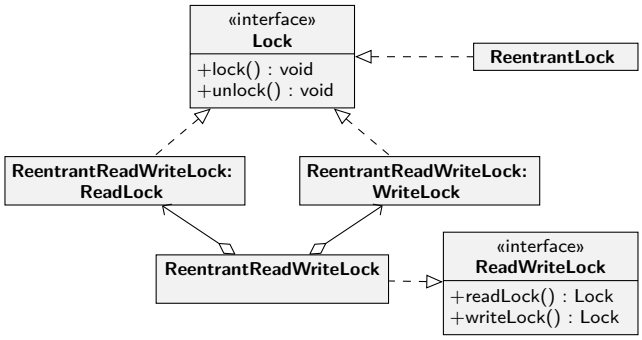

Fig. 2. The hierarchy of locks in the `java.util.concurrent` package.

3.3 Lock Hierarchy Specification

The synchronisation classes in the `Lock` hierarchy in the concurrency package are devoted to resource locking scenarios where either full (write) access is given to one particular thread or partial (read) access is given to an indefinite number of threads. The complete hierarchy of locks is depicted in Fig. 2. We first discuss the specification of the `Lock` interface, and then we proceed with specifications of different lock implementations.

`Lock` Interface Specification As explained above, our specification approach of the synchronisation mechanisms is inspired by the logic of Haack et al. [26]. However, we cannot just translate the rules from [26] into method specifications of the `Lock` interface, because the `Lock` interface can be used in different and wider settings than considered by Haack et al. In particular, `Lock` implementations may be non-reentrant; they may be used to synchronise non-exclusive access; and they may be used in *coupled* pairs to change between shared and exclusive mode (see the read-write lock specification below). Therefore, as an extension to the work of Haack et al., for the specification given in Lst. 10 the following aspects are considered:

- The locks use boolean parameters `isExclusive` and `isReentrant`, which can be correspondingly instantiated by implementations (line 2).
- To allow non-exclusive synchronisation, resource invariants have to be groups (line 1), see Sect. 2.5.
- For the non-exclusive locking scenarios, the client program has to record the amount of the resource fraction that was obtained during locking, so that the lock can reclaim the complete resource fraction upon unlocking. This information is passed around in the `held` predicate, which holds this fraction (line 5) (similar in spirit to the `join` predicate, as discussed in Section 2.5. This is purposely not declared as a group, so that clients are obliged to return their whole share of resources. The `held` predicate is returned during locking in exchange for the `initialized` predicate which is temporarily revoked for the time that the lock is acquired.

```
//@ given group (frac -> resource) inv;
//@ given boolean isExclusive, isReentrant;
public interface Lock {
  //@ group resource initialized(frac p);
  //@ resource held(frac p);

  //@ ghost public final Object parent;

  /*@ given bag<Object> S, frac p;
      requires LockSet(S) ** !(S contains this) ** initialized(p);
      requires parent != null ==> !(S contains parent);
      ensures LockSet(this::parent::S) **
        inv(isExclusive ? 1 : p) ** held(p);
    also
      requires isReentrant ** LockSet(S) **
        (S contains this) ** held(p);
      ensures LockSet(this::S) ** held(p); @*/
  void lock();

  /*@ given bag<Object> S, frac p;
      requires LockSet(this::S) ** (S contains this) ** held(p);
      ensures LockSet(S) ** held(p);
    also
      requires held(p) ** inv(isExclusive ? 1 : p);
      requires LockSet(this::parent::S) ** !(S contains this);
      ensures LockSet(S) ** initialized(p); @*/
  void unlock();
}
```

Lst. 10. Specification of the `Lock` interface.

– For situations where several locks share the same resource and are effectively coupled as one lock, we need to ensure that only one lock is locked at a time. The coupling itself is realised by holding a reference to the parent object that maintains the coupled locks (line 7). The exclusive use of coupled locks is ensured by storing and checking this parent object in the set of currently held locks.
– A separate specification case is given for reentrant locking (when `isReentrant` is true).

As a result, in the specification of method `lock()` in Lst. 10, given the multi-set of locks, i.e., `bag<Object> S`, when the lock is acquired for the first time (lines 9–13), the locking thread gets permissions from the lock. If the lock is reentrant, and the thread already holds the lock (lines 15–17), then no new permission is gained, only the multi-set of locks held by the current thread is extended with this lock (where :: denotes bag addition). For coupled locks (where the parent is not null) the presence of the parent in the lock set is also checked and recorded, to

```
//@ given group (frac -> resource) inv;
//@ given boolean reentrant;
interface ReadWriteLock {
 //@ group resource initialized(frac p);

 //@ given frac p;
 //@ requires initialized(p);
 //@ ensures \result.parent == this ** \result.initialized(p);
 /*@ pure @*/ Lock /*@< inv, false, reentrant >@*/ readLock();

 //@ given frac p;
 //@ requires initialized(p);
 //@ ensures \result.parent == this ** \result.initialized(p);
 /*@ pure @*/ Lock /*@< inv, true, reentrant >@*/ writeLock();
}
```

Lst. 11. Specification of the ReadWriteLock interface.

prevent parallel use of the coupled locks. The specification of method `unlock()` in Lst. 10 describes the reverse process: if the multi-set of locks contains the specific lock only once (lines 24–26), then this means the return of permissions to the lock (i.e., `inv` does not hold in the postcondition) according to the `held` predicate; otherwise (lines 20–22), the thread keeps the permissions, but one occurrence of the lock is removed from the multi-set.

ReentrantLock Specification Class ReentrantLock implements the Lock interface as an exclusive, reentrant lock. Thus, it inherits all specifications from Lock and appropriately instantiates the two class parameters isReentrant and isExclusive both to **true**:

```
//@ given group (frac -> resource) inv;
class ReentrantLock implements Lock /*@< inv, true, true >@*/ {
```

ReadWriteLock Specification The ReadWriteLock is not a lock itself, but a wrapper of two coupled Lock objects: one of them provides exclusive access for writing (WriteLock), while the other allows concurrent reading by several threads (ReadLock). The two classes are commonly implemented as inner classes of the class that implements the ReadWriteLock interface (see Fig. 2 on page 17). The two locks are intended to protect the same memory resources. Hence our specifications in Lst. 11 state that the two getter methods (declared as **pure**) for obtaining the two locks return a lock object with the same resource `inv`, but which are non-exclusive (line 9) and exclusive (line 14), respectively. The aggregate read-write lock has to be initialised itself (lines 7 and 12). Further, using the return value keyword **\result**, we state in the respective postconditions of the getter methods (lines 8 and 13) that the obtained locks are initialised and

hence can be acquired, and that they have the same parent object, which is an instance of the class implementing the `ReadWriteLock` interface.

Example 4. In this example we show how the specification of `ReadWriteLock` helps us to reason about a single-producer multiple-consumer application. Assume an application where one single producer produces data to be used by two separate consumer threads. The producer implemented as a `Producer` class obtains the write lock and then exclusively accesses the shared data field:

```
//@ given group (frac -> resource) pcinv;
public class Producer extends Thread {
  private final Lock/*@<pcinv, true, true>@*/ lock;
  private final SProdMCons example;

  //@ given frac p; requires lock.initialized(p); ensures lock.initialized(p);
  public void produce(){
      // { lock.initialized(p) }
      lock.lock(); // { lock.inv(1) ** lock.held(p) }
      //@ unfold lock.inv(1);
      // { Perm(example.data, 1) ** lock.held(p) } // from pcinv
      sample(); // { Perm(example.data, 1) ** lock.held(p) }
      //@ fold lock.inv(1);
      lock.unlock(); // { lock.initialized(p) }
  }
  // method run
}
```

Then, each consumer is trying to obtain a fractional permission of the shared data to use the value written by the producer:

```
//@ given group (frac -> resource) pcinv;
public class Consumer extends Thread {
  private final Lock/*@<pcinv, false, true>@*/ lock;
  private final SProdMCons example;
  private boolean flag; private int value;

  //@ given frac p; requires lock.initialized(p) ** Perm(this.value,1);
  //@ ensures lock.initialized(p) ** Perm(this.value,1);
  public void consume(){
      // { lock.initialized(p) }
      lock.lock(); // { lock.inv(p) ** lock.held(p) }
      //@ unfold lock.inv(p);
      // { Perm(example.data, p) ** lock.held(p) } // from pcinv
      this.value = example.data;
      if( flag == this.example.PRINT) print( );
      if( flag == this.example.LOG) log( );
      //@ fold lock.inv(p); // { lock.held(p) ** lock.inv(p) }
      lock.unlock(); // { lock.initialized(p) }
  }
  // methods run, print and log
}
```

The producer and consumers are then combined together in the following class:

```
public class SProdMCons {
  //@ group resource pcinv(frac p) = Perm(data, p);
  public final boolean PRINT = true, LOG = false;
  public int data;
  private ReadWriteLock/*@<pcinv, true>@*/ rwl;

  void main(){
    rwl = new ReentrantReadWriteLock/*@< pcinv >@*/();

    Producer producer =
        new Producer/*@< pcinv >@*/(this, rwl.writeLock());
    Consumer printer =
        new Consumer/*@< pcinv >@*/(this, PRINT, rwl.readLock());
    Consumer log =
        new Consumer/*@< pcinv >@*/(this, LOG, rwl.readLock());

    producer.start(); printer.start(); log.start();
    producer.join(); printer.join(); log.join();
  }
}
```

3.4 Semaphore Specification

To illustrate that similar specifications can be used to describe the behaviour of other synchronisers as well, we briefly discuss the specification of class `Semaphore`, which represents a *counting semaphore*. It is used to control threads' accesses to a shared resource, by restricting the number of threads that can access a resource simultaneously. Each semaphore is provided with a property *permits*, that represents the maximum number of threads that can access the protected resource. Accessing the resource must be preceded by acquiring a permit from the semaphore. A semaphore with `n` permits allows a maximum of `n` threads to access the same resource simultaneously. If `n` threads are holding a permit, a new thread that tries to acquire a permit blocks until it is notified that a permit is released.

When initialised with more than 1 permit, a semaphore closely corresponds to a non-reentrant `ReadLock`, but with the number of threads accessing the shared resource explicitly stated and controlled. When initialised with 1 permit, it provides exclusive access, and behaves the same as a non-reentrant `WriteLock`. Therefore, the specification of the semaphore is a stripped-down version of the `Lock` specification, see Lst. 12. In particular, semaphores are never reentrant, and they are not used in coupled combinations. Moreover, since the maximum number of threads that can access the shared resource is predefined with the `permits` field, we can also limit ourselves to simply providing each acquiring thread with an equal split of `1/permits` of the resource invariant (lines 11 and 14). Note also that there is no access permission required for the `permits` field as it is declared to be final and hence can never change after initialisation.

```
//@ given group (frac -> resource) inv;
public class Semaphore {
  //@ resource held(frac p) = initialized(p);
  //@ ghost final int permits;

  //@ requires inv(1) ** permits > 0;
  //@ ensures initialized(1) ** this.permits == permits;
  public Semaphore(int permits);

  //@ given frac p; requires initialized(p);
  //@ ensures inv(1/permits) ** held(p);
  public void acquire();

  //@ given frac p; requires inv(1/permits) ** held(p);
  //@ ensures initialized(p);
  public void release();
}
```

Lst. 12. Specification of the `Semaphore` class.

4 Reasoning about Concurrent Class Invariants

In addition to proving the absence of data races and that data is correctly protected by a synchroniser, we also wish to show properties about the state of the program. In a concurrent setting, many program state properties become *unstable*, i.e., they can be invalidated by other threads. However, this section shows that also in a concurrent setting it is possible to reason about class invariants, restricting the reachable states of an object.

4.1 Concurrent Class Invariants

In essence, a class invariant expresses a property that should always hold for every object of a given class. Concretely, it is defined as a boolean predicate that should be continuously maintained. Consider class `Point` in Lst. 1 with the two fields `x` and `y`. As mentioned, we specified an invariant property that the point object always is in the first or third quarter of the Cartesian space:

```
//@ invariant ((x >= 0 && y >= 0) || (x <= 0 && y <= 0));
```

We could also specify different invariant properties, such as that the relation `x + y >= 0` should always hold for every live `Point` object in the program:

```
//@ invariant I: (x + y >= 0);
```

Notice that we allow to explicitly name invariants (here the invariant is named `I`) as we later need to refer to them symbolically. Although the primary definition of a class invariant is a property that holds *always*, in practice this is impossible, unless the invariant expresses a relation over non-mutable locations.

Otherwise, any change of a location might *break* the invariant, i.e., invalidate the correctness of the invariant predicate. Therefore, a verification technique should allow temporarily breaking of a class invariant at certain *invisible* program states.

In a sequential setting, the standard verification technique suggests that class invariants must hold in every pre- and poststate of a public method [43]. The verifier may assume that the class invariant holds at the beginning of every method, and has an obligation to prove that it still holds at the end of the method. In particular, only these states are *visible* states in a program, and any *breaking* (invalidating) of the invariant in a method's internal state is allowed. Therefore, a class invariant specified for a given class is treated as if it is implicitly added in the pre- and postcondition of every method in this class.

However, in a concurrent program, this approach can not be directly applied. An internal method's state in which a thread invalidates an invariant, might be a prestate of a method (a *visible state*) for another thread. Therefore, assuming the validity of a class invariant at the entrance of a method would be unsound. A technique for verifying concurrent class invariants should allow a thread to break a class invariant only in a state that is *invisible* for the other program threads, so that they would not be able to observe the invalidated state of the object.

To get an intuition of *visible states* in a concurrent program, consider the `move` method in the `Point` class in which both properties `x` and `y` are modified.

```
void move(){
  lockx.lock(); // lockx protects the location x
    x--;
  lockx.unlock();
  locky.lock(); // locky protects the location y
    y++;
  locky.unlock();
 }
```

Clearly, having both updates protected by a lock, the scenario is data-race free. However, in the state after the release of the lock `lockx`, the invariant I may be invalidated, while both `x` and `y` are accessible by another thread: the invalidated state of the `Point` object is then observable/visible. This problem is sometimes called a *high-level data race* [4]. A class invariants verifier should detect an error in this scenario, reporting a visible state in which an invariant can be broken. The correct scenario could be protecting both updates with a single lock:

```
  lock.lock();
    x--;
    y++;
  lock.unlock();
```

In this way, the invalidated state is *hidden* for the other active program threads.

Further, we discuss the *class invariant protocol* for verifying concurrent class invariants. The protocol explains the conditions under which a class invariant

may be safely broken, conditions that allow to assume that an invariant holds, and the obligations when the invariant's validity must be proved. The technique is modular and is built on top of our permission-based separation logic.

4.2 Class Invariant Protocol

We consider that class invariants express properties over instance class fields only. Therefore, we refer to an invariant I defined in a class C through a specific object v of class C, and we write $v.I$. The set of locations referred to by an invariant $v.I$ is called a *footprint of* $v.I$, denoted $\mathsf{fp}(v.I)$.

To define a class invariant, special *state formulas* are used: these formulas express only properties over the shared state and are free of permission expressions. Note that the class invariant I defined in the `Point` class contains neither PointsTo nor Perm predicate. This contrasts standard permission-based separation logic, where every location in a formula must be *framed* by a positive permission.

Assuming a Class Invariant The control of the validity of a class invariant $v.I$ is kept by a predicate/token $\mathsf{holds}(v.I, 1)$. The token is produced after the creation of the valid object v, and afterwards it might be distributed among different threads. Thus, the holds token is a group, i.e., the equivalence mentioned in Sect. 4 holds:

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

The intuitive meaning of this predicate is the following: when a thread holds (part of) this token, it may assume that the invariant $v.I$ holds. This means that at the same time several threads may rely on the validity of the invariant (each of them holding part of the invariant's holds token). The invariant is then *stable* and no other thread may break it. The following verification rule states that the property expressed by a class invariant can be used under the condition that (part of) the holds token is held:

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

Example 5. Lst. 13 shows how a class invariant may be used for verifying a client code. 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 $\mathsf{holds}(s.I, 1)$ (line 3). Then, a set of new threads are forked (lines 5, 6), and each thread gets 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` (lines 19–23). To prove that each new `Point p` is a valid object (`p.x + p.y >=0`) (line 21), each thread uses the class invariant `s.I`, which is guaranteed by the token $\mathsf{holds}(s.I, \pi)$.

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

class Task {
  Point s; int k;
   // ... constructors

  //@ given frac p; requires holds(s.I. p) ** ...;
  //@ ensures holds(s.I. p) ** ...;
  void run(){
    for (int i = 1; i < 10; i ++) {
        // s.I holds (because of the holds token)
        Point p = new Point(s.x + i, s.y + k); // use s.I to verify p.I
        draw(p);
    }
  }
}
```

Lst. 13. Using a class invariant for verifying a client class

Breaking a Class Invariant A class invariant may be temporarily broken by a specific thread, under the condition that the invalid state of the object is not observable by any other thread. To this end, breaking is allowed in explicitly specified parts of the program. The developer is expected to mark the program segment where breaking of an invariant might happen with two specification commands: the command **unpack**(v.I) indicates the start of the segment, while the **pack**(v.I) specification command is required to specify the end of the segment. We call this *an unpacked segment*. In the example of the Point class, both updates should be wrapped in an unpacked segment.

All changes in the unpacked segment should stay hidden for the other program threads. To ensure that no other thread might assume the validity of the invariant $v.I$ within the unpacked segment, the unpack($v.I$) command consumes the full holds($v.I, 1$) token, which ensures that no part of this token is still owned by another thread. The unpack($v.I$) command at the same time produces a new predicate, the unpacked($v.I, 1$) token, which serves as a license for the thread to break the invariant $v.I$. Holding the unpacked token is a required condition for assigning to any location that appears in the footprint of the invariant. Once all updates are done, the running thread must reestablish the validity of $v.I$

```
//@requires holds(this.I, 1); ensures holds(this.I, 1);
void move() {
  lock.lock(); // { Perm(x, 1) ** Perm(y, 1); }
  // { Perm(x, 1) ** Perm(y, 1) ** holds(this.I, 1) }
  // { Perm(x, 1) ** Perm(y, 1) ** holds(this.I, 1) ** this.I }
  // { Perm(x, 1) ** Perm(y, 1) ** holds(this.I, 1) ** x + y >= 0 }
  //@ unpack(this.I);
  // { Perm(x, 1) ** Perm(y, 1) ** unpacked(this.I, 1) ** x + y >= 0 }
    x--;
  // { Perm(x, 1) ** Perm(y, 1) ** unpacked(this.I, 1) ** x + y >= -1 }
    y++;
  // { Perm(x, 1) ** Perm(y, 1) ** unpacked(this.I, 1) ** x + y >= 0 }
  // { Perm(x, 1) ** Perm(y, 1) ** unpacked(this.I, 1) ** this.I }
  //@ pack(this.I);
  //@ { Perm(x, 1) ** Perm(y, 1) ** holds(this.I, 1) }
  lock.unlock();
  // { holds(this.I, 1); }
}
```

Lst. 14. An unpacked segment

and call the pack($v.I$) command, which trades the unpacked($v.I, 1$) token for the holds($v.I, 1$) token. The unpack($v.I$) command is always followed by pack($v.I$) within the same method and executed by the same thread. This thread is called *a holder* of the unpacked segment. Lst. 14 shows the specified `move` method with the proof outline.

Restrictions to Unpacked Segments As explained above, the unpacked segment may contain states in which a certain object is invalidated. Therefore, all changes in the segment must not be publicly exposed, i.e., they should not be observable for any thread except for the *holder* of the segment. Because of this, within an unpacked segment it is forbidden for the running thread to release permissions and to make them accessible to other threads. In particular, only safe commands are allowed, i.e., commands that exclude any lock-related operation (acquiring, releasing or committing a lock). Note that in the example in Lst. 14, the lock is acquired and released outside the unpacked segment. A call to a method m is a safe command if the called method m itself is safe, i.e., a method composed of safe commands only. A safe method should be specified with the optional modifier **safe**.

Forking a safe thread, i.e., a thread with a `/*@safe@*/run()` method, within an unpacked segment is also allowed, under the condition that the thread must be joined before the unpacked segment ends. These threads are called *local to the segment*. A safe thread may further fork other safe threads. The breaking token might then 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, the unpacked token is also a splittable token. This

```
void move(){
  //@ unpack(this.I);
  lock.lock(); // invalid call (must happen before unpacking)
  t.fork(); // allowed if t is a safe thread
  updateY(); // allowed if updateY is a safe method
  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. 15. Restrictions to an unpacked segment

means that breaking an invariant $v.I$ does not require full unpacked token, but for any $\pi > 0$, the predicate $\mathsf{unpacked}(v.I, \pi)$ is valid breaking permission. The example on Lst. 15 shows the restrictions in an unpacked segment.

Object Initialisation Object initialisation (the object constructor) is divided into two phases: (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. (2) the init method follows mandatorily after object construction, where object fields are initialised. After this phase, the object may be used.

For every invariant $v.I$ the $\mathsf{unpacked}(v.I, 1)$ token is produced for the first time at the end of the first phase of v's initialisation: the created object v is then still empty and might be in an invalid state in which some of its invariants are broken. This means that after this first phase, every invariant of the object v is in an unpacked state.

After the second phase and the initialisation of all v's object fields, the object v should be in a valid state. For every invariant $v.I$, the $\mathsf{pack}(v.I)$ specification command is called by default at the end of the init method. Hence, at the end of v's initialisation, the invariant $v.I$ is in a packed state. We show the initialisation of a Point object in Lst. 16.

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 Point {
   int x, y;
   Lock lock;
   //@ invariant I: (x + y >= 0);

   //@ ensures Perm(x, 1) ** Perm(y, 1) ** unpacked(I, 1);
   public Point() { /* effectively calls init */ }

   //@ requires Perm(x, 1) ** Perm(y, 1) ** unpacked(I, 1);
   //@ ensures holds(I, 1);
   void init() {
     x = 0; y = 0;
     //@ resource rinv = Perm(x, 1) ** Perm(y, 1);
     lock = new Lock/*@< rinv >@*/();
     //@ pack(I);
     // { Perm(x, 1) ** Perm(y, 1) ** holds(I, 1) }
     lock.commit(); // permissions are transferred to the lock
     // { holds(I, 1) }
   }
}
```

Lst. 16. Object Initialisation

4.3 Modular Verification

To be practically useful, the verification technique should be *modular*. Rule **R2**, listed above, 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. Therefore, it is impossible for the verifier to check modularly whether rule **R2** is properly satisfied.

Consider the example in Lst. 17: the class `Line` contains references of two `Point` objects (the **rep** modifiers in lines 2 and 3 are discussed later). The invariant `I1` in the class `Line` refers to fields in `p1` and `p2`. This means that assigning to a field `x` or `y` of a `Point` object may break an invariant of an existing `Line` object: therefore, this assignment should be allowed if the invariant `I1` is also unpacked. When verifying the `Point` class, the verifier should be aware of the `Line` class, and possibly other classes that refer to the fields `x` and `y` in the `Point` class. Thus, it is impossible to verify the `Point` class in isolation.

This problem with modularity is not typical for concurrent programs, but also manifests itself when verifying class invariants in sequential programs. Several solutions are suggested for modular verification of sequential invariants [44,5,42,20]. Mostly they use the restrictions from Müller's *ownership type system* [19].

Using the restrictions of the ownership type system can also help to provide modular verification of concurrent class invariants. Below we first shortly discuss the ownership-type system and then we explain the verification technique for concurrent class invariants based on this type system. In general, in rule **R2**, to assign a location $p.f$, only the invariants of the object p are explicitly checked,

```
class Line {
  /*@ rep @*/ Point p1;
  /*@ rep @*/ Point p2;

  //@ invariant I1: (p1.x + p2.x <= 10 ** p1.y + p2.y <= 10);

  //@ requires unpacked(I1, 1) ** ... (permissions) ...;
  //@ ensures holds(I1, 1) ** holds(p1.I, 1) ** holds(p2.I, 1);
  //@ ensures ... (permissions) ...;
  void init(){
    p1 = new /*@ rep @*/ Point(0, 0);
    p2 = new /*@ rep @*/ Point(0, 5);
    //@ pack(this.I1);
  }

  //@ requires holds(this.I1, 1) ** holds(p1.I, 1);
  //@ requires Perm(p1.x, 1) ** Perm(p1.y, 1);
  //@ ensures holds(this.I1, 1) ** holds(p1.I, 1);
  //@ ensures Perm(p1.x, 1) ** Perm(p1.y, 1);
  void moveP1() {
    //@ unpack(this.I1);
    p1.move();
    //@ pack(this.I1); // trades the unpacked token for holds token
  }

class Point{
   int x;
   int y;
   //@ invariant I: (x + y >= 0);

  //@ given frac p;
  //@ requires Perm(x, 1) ** Perm(y, 1) ** holds(I, p);
  //@ ensures Perm(x, 1) ** Perm(y, 1) ** holds(I, p);
  /*@ safe @*/ void move() {
    //@ unpack(this.I);
    x--;
    y++;
    //@ pack(this.I);
  }
}
```

Lst. 17. Modular verification

while the technique guarantees that the `unpacked` token is implicitly held for all other necessary class invariants.

Ownership-Based Types The ownership type system forces all objects in the heap to be organised in a structural way and it applies certain restrictions to the operations applicable to each object reference. In particular, each object is required to respect the concept of ownership topology, where objects are organised in a hierarchy. Each object has one owner, either the root of the tree, or another object in the heap. Each ancestor of an object p in the tree is p's *transitive owner*. The developer decides the position of an object in the tree by attaching an appropriate modifier from the set {**rep**, **peer**, **rd** } when the object is created. This modifier becomes a part of the type of the object reference, which shows the relation between the object and the **this** object. For example, if a new `Point` object is created with the **rep** modifier,

```
/*@ rep @*/ Point point = new /*@ rep @*/ Point();
```

the type **rep** indicates that the new object is owned by **this** object. The type **peer** is used when creating an object that should have the same owner as the **this** object, while **rd** is used for any other object. The type is actually attached to the object reference, because it shows the relation of the object in the context of the **this** object. If another reference of the same `Point` object is used in a different context, that reference would have another modifier, calculated appropriately according to the new context. For example, if the object a owns b, while b owns c, the type of a reference of c in the context of the object b is **rep**, while the type of a reference of c in the context of a is **rd**.

Having all objects in the heap structurally organised, the ownership type system imposes certain restrictions: writing to a field $p.f$ or a call to a *non-pure* method (a method with side-effects) with a receiver p is not allowed if the ownership type of the reference p is **rd**. In this way, each object controls all updates that happen in its transitively owned objects. This guarantees the following rule:

> **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.

Verification Technique via Ownership Types To use the ownership-type system for modular verification of class invariants, additionally, the definition of a class invariant is restricted such that:

> **RCI** A class invariant $v.I$ may only express properties over fields of the object v, or fields of object that is transitively owned by v.

From the rule **RCI**, we can observe the following: a location $p.f$ may be referred to by an invariant of the object p or of an object v that is a transitive owner of p. Moreover, according to **RO**, the assignment of $p.f$ is preceded by a method call where v is a receiver. These two observations give the right to

define the following: when assigning to a location $p.f$, it is enough to require the unpacked token only for the invariants of the object p ($p.I$) that refer to $p.f$. If any other invariant $v.I$ refers to $p.f$, then v is a transitive owner of p and the check that the actual thread holds the unpacked token for $v.I$ is a requirement of the method call where object v is a receiver.

More precisely, rule **R2** listed above is replaced with the following two rules:

> **R2.1** A precondition for assigning a value to a field $p.f$ requires a token unpacked($p.I, \pi$), $\pi > 0$, for each invariant I of the object p that refers to $p.f$.
> **R2.2** A precondition for invoking a method m that assigns a field $p.f$ requires the token unpacked(this.I, π), $\pi > 0$, for each invariant I of the **this** object that refers to $p.f$.

Example 6. In the example in Lst. 17 on page 29, each `Line` object `line` owns the objects `line.p1` and `line.p2`. This hierarchy allows the invariant `I1` in the `Line` class to express properties over fields in `p1` and `p2`. The updates in the `move` method in the `Point` class (line 34) might break the invariant `I` defined in the class `Point`, as well as the invariant `I1` in the `Line` class. Therefore, before these updates happen, both `I` and `I1` have to be in an unpacked state. The invariant `I` is required to be unpacked in the method `move`, before the updates of `x` and `y` (line 35), while `I1` has to be unpacked before the call to the `move` method in the `Line` class (line 21).

It is important to note that permissions for the updating fields, `p1.x` and `p1.y`, must be obtained outside the unpacked segments of both invariants `I` and `I1`. No locks are then acquired in the `move` method, and thus the method is safe (line 34). If these permissions were obtained through a lock inside the `move` method in the `Point` class (as in Lst. 14), the `Line` class could not be verified: the unpacked segment in the `moveP1` method would then contain a method call to an unsafe method.

With this approach the verifier may perform both **R2.1** and **R2.2** having only the knowledge of the context where the assignment happens. This makes the approach modular. Although the method is applicable to ownership-based type systems only, this is not considered as a serious restriction, because ownership is a common and natural concept for organisation of objects in a program.

5 Reasoning about GPU Kernels

Above, we have considered how to reason about multithreaded Java programs, running on one or multiple CPUs. However, to achieve an increase in performance, modern hardware also uses different computing paradigms. GPUs, graphical processing units, which were initially designed to support computer graphics, are more and more used also for other programming tasks, leading to the development of the area of GPGPU (General Purpose GPU) programming.

```
kernel demo {
  global int[gsize] a,b;
  void main(){
    a[tid]:=tid;
    barrier(global);
    b[tid]:=a[(tid+1) mod gsize];
  }
}
```

Lst. 18. Basic example kernel

Until 2006 GPGPU programming was mainly done in CUDA [34], a proprietary GPU programming language from NVIDIA. However, recently a new platform-independent, low-level programming language standard for GPGPU programming, OpenCL [47], emerged. As a result, GPUs are now used in many different fields, including media processing [18], medical imaging [57] and eye-tracking [45].

The main characteristic of GPU kernels is that each kernel constitutes a massive number of parallel threads. All threads execute the same instruction, but each thread operates on its own share of memory. Barriers are used as the main synchronisation primitive between threads in a kernel.

This section shows how permission-based separation logic also can be used to reason about OpenCL kernels.

5.1 GPU Architecture

Before presenting our verification technique, we first briefly discuss the main characteristics of the GPU architecture (for more details, see the OpenCL specification [35]).

A GPU runs hundreds of threads simultaneously. All threads within the same *kernel* execute the same instruction, but on different data: the *Single Instruction Multiple Data (SIMD)* execution model. GPU kernels are invoked by a *host* program, typically running on a CPU. Threads are grouped into *work groups*. GPUs have three different memory regions: *global*, *local*, and *private* memory. Private memory is local to a single thread, local memory is shared between threads within a work group, and global memory is accessible to all threads in a kernel, and to the host program. Threads within a single work group can synchronise by using a *barrier*: all threads block at the barrier until all other threads have also reached this barrier. A barrier instruction comes with a flag to indicate whether it synchronises global or local memory, or both. Notice that threads within different work groups cannot synchronise.

Example 7. Lst. 18 shows the code of a kernel that initialises a global array `b` in such a way that position i contains $i+1$ modulo the length of the array. It does so in a complicated way. Each thread first assign its thread id `tid` to position i

of a temporary array `a`. Then all threads wait for each other (which means that this code can only run for a single working group) and then position i of array `b` is assigned by reading position $i + 1$ modulo the working group size of array `a`. If the barrier would be removed, there would be a data race on `a[i]`.

5.2 Verification of GPGPU Kernels

As mentioned, permission-based separation logic also is suitable to reason about kernel programs. When reasoning about kernel programs, we can prove that a kernel (i) does not have data races, and (ii) that it respects its functional behaviour specification. Kernels can exhibit two kinds of data races: (i) parallel threads within a work group can access the same location, either in global or in local memory, and this access is not ordered by an appropriate barrier, and (ii) parallel threads within different work groups can access the same locations in global memory. With our logic, we can verify the absence of both kinds of data races.

Concretely, for each kernel we specify all the permissions that are needed to execute the kernel. Upon invocation of the kernel, these permissions are transferred from the host code to the kernel. Within the kernel, the available permissions are distributed over the threads. Every time a barrier is reached, a barrier specification specifies how the permissions are redistributed over the threads (similar to the barrier specifications of Hobor et al. [29]). The barrier specification also specifies functional pre- and postconditions for the barrier. Essentially this specifies how knowledge about the global state upon reaching the barrier is spread over the different threads.

Traditionally, separation logic considers a single heap for the program. However, to reason about kernels, we make an explicit distinction between global and local memory. To support our reasoning method, kernels, work groups and threads are specified as follows:

- The *kernel specification* is a triple $(K_{res}, K_{pre}, K_{post})$. The resource formula K_{res} specifies all resources in global memory that are passed from the host program to the kernel, while K_{pre} and K_{post} specify the functional kernel pre- and postcondition, respectively. K_{pre} and K_{post} have to be framed by K_{res}. An invocation of a kernel by a host program is correct if the host program holds the necessary resources and fulfils the preconditions.
- The *group specification* is a triple $(G_{res}, G_{pre}, G_{post})$, where G_{res} specifies the resources in global memory that can be used by the threads in this group, and G_{pre} and G_{post} specify the functional pre- and postcondition, respectively, again framed by G_{res}. Notice that locations defined in local memory are only valid inside the work group and thus the work group always holds write permissions for these locations.
- Permissions and conditions in the work group are distributed over the work group's threads by the *thread specification* $(T^{res}_{pre}, T_{pre}, T^{res}_{post}, T_{post})$. Because threads within a work group can exchange permissions, we allow the resources before (T^{res}_{pre}) and after execution (T^{res}_{post}) to be different. The func-

tional behaviour is specified by T_{pre} and T_{post}, which must be framed by T^{res}_{pre} and T^{res}_{post}, respectively.
- A *barrier specification* $(B_{res}, B_{pre}, B_{post})$ specifies resources, and a pre- and postcondition for each barrier in the kernel. B_{res} specifies how permissions are redistributed over the threads (depending on the barrier flag, these can be permissions on local memory only, on global memory only, or a combination of global and local memory). The barrier precondition B_{pre} specifies the property that has to hold when a thread reaches the barrier. It must be framed by the resources that were specified by the previous barrier (considering the thread start as an implicit barrier). The barrier postcondition B_{post} specifies the property that may be assumed to continue verification of the thread. It must be *framed* by B_{res}.

Notice that it is sufficient to specify a single permission formula for a kernel and a work group. Since work groups do not synchronise with each other, there is no way to redistribute permissions over kernels or work groups. Within a work group, permissions are redistributed over the threads only at a barrier, the code between barriers always holds the same set of permissions.

Given a fully annotated kernel, verification of the kernel w.r.t. its specification essentially boils down to verification of the following properties:

- Each thread is verified w.r.t. the thread specification, i.e., given the thread's code T_{body}, the Hoare triple $\{T_{res} \mathbin{**} T_{pre}\}\ T_{body}\{T_{post}\}$ is verified using the permission-based separation logic rules defined in Sect. 5.4. Each barrier is verified as a method call with precondition $R_{\mathsf{cur}} \mathbin{**} B_{pre}$ and postcondition $B_{res} \mathbin{**} B_{post}$, where R_{cur} specifies all current resources.
- The kernel resources are sufficient for the distribution over the work groups, as specified by the group resources.
- The kernel precondition implies the work group's preconditions.
- The group resources and accesses to local memory are sufficient for the distribution of resources over the threads.
- The work group precondition implies the thread's preconditions.
- Each barrier redistributes only resources that are available in the work group.
- For each barrier the postcondition for each thread follows from the precondition in the thread, and the fenced conjuncts of the preconditions of all other threads in the work group.
- The universal quantification over all threads' postconditions implies the work group's postcondition.
- The universal quantification over all work groups' postconditions implies the kernel's postcondition.

The first condition is checked by the Hoare logic rules discussed below; the other conditions are encoded as additional checks in the VerCors tool set.

We will illustrate our approach on the kernel program discussed in Example 7.

Example 8. Consider the kernel in Lst. 18. For simplicity, it has a single work group, so the kernel level and group level specification are the same.

```
kernel demo {
  global int[gsize] a;
  global int[gsize] b;

  requires Perm(a[tid],100) ** Perm(b[tid],100);
  ensures Perm(b[tid],100) ** b[tid] == (tid+1) mod gsize;
  void main(){
    a[tid]:=tid;
    barrier(global) {
      requires a[tid]=tid;
      ensures Perm(a[(tid+1) mod gsize],10) ** Perm(b[tid],100);
      ensures a[(tid+1) mod gsize]=(tid+1) mod gsize; }
   b[tid]:=a[(tid+1) mod gsize];
  }
}
```

Lst. 19. VerCors tool annotated version of the code in Lst. 18.

At the kernel level, the required resources K_{res} are write permissions on arrays `a` and `b`. The kernel precondition K_{pre} states that the length of both arrays should be the same as the number of threads (denoted as *gsize* for work group size). The kernel postcondition expresses that afterwards, for any i in the range of the array, $\mathtt{b}[i] = (i+1)\%gsize$. Each thread i initially obtains a write permission at $\mathtt{a}[i]$. When thread i reaches the barrier, the property $\mathtt{a}[i] = i$ holds; this is the barrier precondition. After the barrier, each thread i obtains a write permission on $\mathtt{b}[i]$ and a read permission on $\mathtt{a}[(i+1)\%gsize]$, and it continues its computation with the barrier postcondition that $\mathtt{a}[(i+1)\%gsize] = (i+1)\%gsize$. From this, each thread i can establish the thread's postcondition $\mathtt{b}[i] = (i+1)\%gsize$, which is sufficient to establish the kernel's postcondition. See Lst. 19 for a tool-verified annotated version.

Notice that the logic contains many levels of specification. However, typically many of these specifications can be generated, satisfying the properties above by construction. As discussed in Section 5.5 below, for the tool implementation it is sufficient to provide the thread and the barrier specifications.

5.3 Kernel Programming Language

This section defines a simple kernel language. The next section defines the logic over this simplified language, however we would like to emphasise that our tool can verify real OpenCL kernels.

Our language is based on the Kernel Programming Language (KPL) of Betts et al. [9]. However, the original version of KPL did not distinguish between global and local memory, while we do. As kernel procedures cannot recursively call themselves, we restrict the language to a single block of kernel code, without loss of generality. Fig. 3 presents the syntax of our language. Each kernel is merely

Reserved global identifiers (constant within a thread):

tid	Thread identifier with respect to the kernel
gid	Group identifier with respect to the kernel
lid	Local thread identifier with respect to the work group
tcount	The total number of threads in the kernel
gsize	The number of threads per work group

Kernel language:

b ::= boolean expression over global constants and private variables
e ::= integer expression over global constants and private variables
S ::= $\mathsf{v} := \mathsf{e} \mid v := \mathsf{rdloc}(e) \mid v := \mathsf{rdglob}(e) \mid \mathsf{wrloc}(e_1, e_2) \mid \mathsf{wrglob}(e_1, e_2)$
$\quad \mid \mathsf{nop} \mid S_1; S_2 \mid \mathsf{if}\, b\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \mid \mathsf{while}\, b\, \mathsf{do}\, S \mid bid : \mathsf{barrier}(F)$
F ::= $\emptyset \mid \{\mathsf{local}\} \mid \{\mathsf{global}\} \mid \{\mathsf{local}, \mathsf{global}\}$

Fig. 3. Syntax for Kernel Programming Language

a single statement, which is executed by all threads, where threads are divided into one or more work groups. For simplicity, but without loss of generality, global and local memory are assumed to be single shared arrays (similar to the original KPL presentation [9]). There are 4 memory access operations: read from location e_1 in local memory ($v := \mathsf{rdloc}(e_1)$); write e_2 to location e_1 in local memory ($\mathsf{wrloc}(e_1, e_2)$); read from global memory ($v := \mathsf{rdglob}(e)$); and write to global memory ($\mathsf{wrglob}(e_1, e_2)$). Finally, there is a barrier operation, taking as argument a subset of the flags local and global, which describes which of the two memories are fenced by the barrier. Each barrier is labelled with an identifier *bid*.

5.4 Kernel Program Logic

This section formally defines the rules to reason about OpenCL kernels. As explained above, we distinguish between two kinds of formulas: resource formulas (in permission-based separation logic), and property formulas (in first-order logic).

Syntax of Resource Formulas Before presenting the verification rules, we first define the syntax of resource formulas. Section 2 on page 4 defined the syntax of resource formulas. However, our kernel programming language uses a very simple form of expressions only, and the syntax explicitly distinguishes between access to global and local memory. Therefore, in our kernel specification language we follow the same pattern, and we explicitly use different permission statements for local and global memory.

As mentioned above, the behaviour of kernels, groups, threads and barriers is defined as tuples $(K_{res}, K_{pre}, K_{post})$, $(G_{res}, G_{pre}, G_{post})$, $(T^{res}_{pre}, T_{pre}, T^{res}_{post}, T_{post})$, and $(B_{res}, B_{pre}, B_{post})$, respectively, where the resource formulas are defined by

$$\frac{}{\{R, P[v := e]\}\mathsf{v} := \mathsf{e}\{R, P\}}\ (\text{assign})$$

$$\frac{}{\{R \mathbin{**} \mathsf{LPerm}(e,\pi), P[v := L[e]]\}v := \mathsf{rdloc}(e)\{R \mathbin{**} \mathsf{LPerm}(e,\pi), P\}}\ (\text{read-local})$$

$$\frac{}{\{R \mathbin{**} \mathsf{LPerm}(e_1,\mathsf{rw}), P[L[e_1] := e_2]\}\mathsf{wrloc}(e_1,e_2)\{R \mathbin{**} \mathsf{LPerm}(e_1,\mathsf{rw}), P\}}\ (\text{write-local})$$

$$\frac{}{\{R_{\mathsf{cur}}, B_{pre}(bid)\}bid : \mathsf{barrier}(F)\{B_{res}(bid), B_{post}(bid)\}}\ (\text{barrier})$$

Fig. 4. Hoare logic rules

the following grammar:

$$\begin{aligned} E &::= \text{expressions over global constants, private variables, } \mathsf{rdloc}(E), \mathsf{rdglob}(E) \\ R &::= \mathsf{true} \mid \mathsf{LPerm}(E,p) \mid \mathsf{GPerm}(E,p) \mid E \mathbin{==>} R \\ &\quad \mid R_1 \mathbin{**} R_2 \mid (\texttt{\textbackslash forall* T v}; E(v); R(v)) \end{aligned}$$

Resource formulas can *frame* first-order logic formulas. For this purpose, we define the footprint of a resource formula as all global and local memory locations that are accessed to evaluate the formula (see [11] for more details).

Hoare Triples for Kernels Since in our logic we explicitly separate the resource formulas and the first-order logic properties, we first have to redefine the meaning of a Hoare triple in our setting, where the pre- and the postcondition consist of a resource formula, and a first-order logic formula, such that the pair is properly framed.

$$\begin{aligned} &\{R_1, P_1\}S\{R_2, P_2\} = \\ &\quad \forall \mathcal{R}\, \gamma.(\Gamma \vdash \mathcal{R}; \gamma \models R_1 \mathbin{**} P_1) \wedge (S, (\mathcal{R}_{\mathsf{mg}}, \mathcal{R}_{\mathsf{ml}}, \gamma), \mathsf{R}) \rightarrow^* (\epsilon, (\sigma, \delta, \gamma'), F) \Rightarrow \\ &\qquad \forall \mathcal{R}'.\mathcal{R}'_{\mathsf{mg}} = \sigma \wedge \mathcal{R}'_{\mathsf{ml}} = \delta.\Gamma \vdash \mathcal{R}'; \gamma' \models R_2 \mathbin{**} P_2 \end{aligned}$$

Fig. 4 summarises the most important Hoare logic rules to reason about kernel threads; in addition there are the standard rules for sequential compositional, conditionals, and loops. Rule (assign) applies for updates to local memory. Rules (read-local) and (write-local) specifies look-up and update of local memory (where $L[e]$ denotes the value stored at location e in the local memory array, and substitution is as usually defined for arrays, cf. [3]):

$$L[e][L[e_1] := e_2] = (e = e_1)?e_2 : L[e]$$

Similar rules are defined for global memory (not given here, for space reasons).

The rule (barrier) reflects the functionality of the barrier from the point of view of one thread. First, the resources before (R_{cur}) are replaced with the barrier resources for the thread ($B_{res}(bid)$). Second, the barrier precondition ($B_{pre}(tid)$) is replaced by the post condition ($B_{post}(tid)$). The requirement that the preconditions within a group imply the postconditions is not enforced by this rule; it must be checked separately.

5.5 Example: Binomial Coefficient

Finally we discuss the verification of a more involved kernel, to illustrate the power of our verification technique. The full example is available on-line and can be tried in the on-line version of our tool set [58].

The kernel program in Lst. 20 computes the binomial coefficients

$$\binom{N-1}{0} \cdots \binom{N-1}{N-1}$$

using N threads forming a single work group. Due to space restrictions, only the critical parts of the specifications have been given. The actual verified version has longer and more tedious specifications.

The intended output is the global array `bin`. The local array `tmp` is used for exchanging data between threads. The algorithm proceeds in $N-1$ iterations and in each iteration `bin` contains a row from Pascal's triangle as the first part, and ones for the unused part.

On line 10 the entire `bin` array is initialised to 1. This satisfies the invariants on line 11/12 that states that the array `bin` contains the N^{th} row of Pascal's triangle, followed by ones. The loop body first copies the `bin` array to the `tmp` array, then using a barrier that fences the local variable. These values are then transmitted to the next thread and the write permission on `tmp` is exchanged for a read permissions. Then, for the relevant subset of threads, the equation

$$\binom{N}{k} = \binom{N-1}{k-1} + \binom{N-1}{k}$$

is used to update `bin`, and the second barrier returns write permission on `tmp`.

Note that the first barrier fences the local variables, which is necessary to ensure that the next thread can see the values. The second barrier does not fence any variables because it is only there to ensure that the value has been read and processed, making it safe to write the next value in `tmp`.

6 Related Work

To conclude this paper, we briefly give some pointers to related work. We do not intend this discussion to be complete; for this we refer to the related work sections of our individual papers.

6.1 Tools for Verification of Java Programs

The examples in this paper are specified in a dialect of separation logic that extends the JML [36] specification language with concurrent features. Many different verification tools for JML already exist [13]. Also tools for separation logic exist, such as VeriFast [33], SmallFoot [8], and jStar [21]. We also mention Chalice [37] here. Strictly speaking this is not a separation logic tool, however its

```
kernel binomial {
  global int[gsize] bin;
  local int[gsize] tmp;

  requires gsize > 1 ** Perm(bin[tid],1) ** Perm(tmp[tid],100);
  ensures Perm(bin[tid],100) ** bin[tid]=binom(gsize-1,tid);
  void main(){
    int temp;
    int N:=1;
    bin[tid]:=1;
    invariant Perm(bin[tid],100) ** Perm(tmp[tid],100);
    invariant tid<N ? bin[tid]=binom(N,tid) : bin[tid]=1;
    while(N<gsize-1){
      tmp[tid]:=bin[tid];
      barrier(1,{local}){
        ensures Perm(bin[tid],100) ** Perm(tmp[(tid-1) mod gsize],10);
        ensures 0<tid & tid<=N -> tmp[(tid-1) mod gsize]=binom(N,tid-1);
      }
      N := N+1;
      if(0<tid & tid<N){
        temp:=tmp[(tid-1) mod gsize];
        bin[tid]:=temp+bin[tid];
      }
      barrier(2,{}){
        ensures Perm(bin[tid],100) ** Perm(tmp[tid],100);
      }
    }
  }
}
```

Lst. 20. Kernel program for binomial coefficients.

specification language (implicit dynamic frames [56]) is equivalent to separation logic [51].

SmallFoot and jStar support basic separation logic, without fractional permissions. The Chalice specification language is quite similar to ours, but more restricted. For example, it does not allow predicates with parameters and it does not support the magic wand. Moreover, its programming language does not have inheritance. The VeriFast tool supports both C and Java. The main difference with the VerCors tool is that it uses a pure version of separation logic rather than our free form.

One of the core components of a specification language for concurrent software is the access permission model, which determines how flexible the specification languages for access permissions is. Separation logic is not the only one available. One of the simplest models is the permission model of the Spec# programming system for C# [6]. This model organises access permissions as a forest of trees. It is also used in the VCC verifier for C code [15].

6.2 Synchronisers

As mentioned, our synchroniser specifications extend our earlier formalisation of reentrant locks [26]. Several other built-in formalisations of locks and synchronisation primitives exist. Chalice [37] formalises simple non-reentrant locks built into the Chalice language. The work of Gotsman et al. [25] is similar to our earlier formalisation, and we believe that our high-level approach could also be easily applied there to treat a wider range of synchronisation primitives.

Similarly, the work of Hobor and Gherghina on formalising Pthread-style barriers in Separation Logic [29] follows very similar principles. This work was the basis for our barrier specifications for OpenCL kernels. However, since OpenCL barriers are simpler, our barrier specifications for kernel programs also are much simpler. We are currently working on specifying and verifying also a Java API version of a cyclic barrier.

Finally, the VeriFast tool [33] adopts an approach similar to ours – locking is also specified on the API level, but only for simple and non-reentrant locks, and so-called higher-order abstract predicates are functionally similar to our class level specification parameters.

6.3 Class Invariants

The early developed techniques for verification of class invariants in sequential programs [43,41] support invariants with restricted definition only. This work is unsound for more complex data structures, for example if an invariant captures properties over different objects. Later, Poetzsch-Heffter [53] and Huizing et al. [31] developed sound techniques that do not restrict the invariant definition or the program itself; however, both approaches are not modular.

Müller et al. [44] 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 [42], are designed for ownership-based type systems. All these techniques are captured in Drossopoulou et al.'s abstract unified framework [22].

Weiß models class invariants with a boolean model field *inv* [59]. Their validity is checked only on demand. Specifications use *inv* explicitly where needed, while this.*inv* is implicitly generated in each method pre- and postcondition.

Jacobs et al. [32] suggest a technique for verifying multithreaded programs with class invariants, using the *Boogie methodology* [5] for sequential programs. With this approach, a thread is allowed to break a class invariant of an object only if it completely owns the object, i.e., no other thread can access any field of this object. This is in contrast with the approach presented in this paper, where breaking a class invariant is independent of permissions on heap memory.

A different approach for modular verification of object invariants in concurrent programs is proposed by Cohen [16], implemented in VCC [15]. 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.

6.4 GPUs

There already exists some work on the verification of GPU kernels. However, these approaches mainly focus on the verification of data race freedom of the interleaving of two arbitrary threads, whereas we verify an arbitrary single thread, and also consider functional correctness.

Li and Gopalakrishnan [40] verify CUDA programs by symbolically encoding thread interleavings. They focus on data race freedom, and were the first to observe that to ensure data race freedom it was sufficient to verify the interleavings of two arbitrary threads.

Betts et al. [9] verify GPU programs by encoding their behaviour as a BoogiePL program. The GPUVerify tool is highly efficient at automatically verifying data race freedom and absence of barrier divergence. However, it abstracts away from all data and cannot easily prove functional correctness.

7 Summary and Directions for Future Work

This paper illustrated the VerCors approach to verification of concurrent software. The approach uses permission-based separation logic as the underlying logic to handle the concurrency-related features. However, compared to other projects handling verification of concurrent software, the VerCors project focuses on making verification practical. It achieves this by using an easily accessible specification language, that reuses important aspects of the JML specification language, and by concentrating on also verifying functional program properties.

We discussed some distinguishing features of the VerCors project in more detail. First of all, we showed how different synchronisers can be specified uniformly in the specification language, and how these specifications are used to verify other programs. Moreover, (simplified versions of) Java's reference implementations of these synchronisers have been proven correct w.r.t. these specifications. This verification has not been discussed in detail in this paper, but it is important to know that the specifications are indeed correct. Second, we also discussed how an important class of functional correctness properties, namely that of class invariants can be specified and verified in a modular way in a concurrent setting. Key ingredient of the approach is that the annotator explicitly can control when the invariant may be broken, but that it has to be ensured that the broken invariant is not visible to other threads (until it is reestablished again). Last, we also showed how the approach can be used to verify other concurrency paradigms, and in particular how it is used to verify vector programs, following the Single Instruction Multiple Data paradigm. Concretely, we apply this technique to prove data race freedom and functional correctness of OpenCL kernel programs.

Future Work The work described in this paper is part of an ongoing project and much more work remains to be done. In particular, the tool support needs to be improved and tested further, and on larger applications. At the moment, a user has to add many proof hints and annotations by hand. To make the approach

scale to larger applications and usable for other software developers, a large effort is needed on generating annotations and proof hints automatically.

Additionally, we also plan to expand the application domain of the VerCors tool set. We would like to study what effort is needed to extend the verification techniques to other programming languages, e.g., C and Scala. We are also developing techniques to study more advanced functional properties; in particular to be able to verify that an application eventually computes an expected result.

For the GPU verification, we plan to study parallellisation and optimisations in more detail. When a sequential program is verified, and then parallellised to a vector program by a parallellising compiler, how can we make sure that the resulting vector program is also correct? And when the vector program is further optimised, to increase performance, how can we make sure that correctness of the optimised program is maintained?

Acknowledgments This work was supported by ERC grant 258405 for the VerCors project (Amighi, Blom, Huisman, Mostowski, and Zaharieva-Stojanovski), and EU STREP project 287767 CARP (Blom, Darabi, and Huisman).

References

1. A. Amighi, S. Blom, and M. Huisman. Resource protection using atomics: Patterns and verifications. Technical Report TR-CTIT-13-10, CTIT, University of Twente, 2013.
2. A. Amighi, S. Blom, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Formal specifications for Java's synchronisation classes. In A. L. Lafuente and E. Tuosto, editors, *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 725–733. IEEE Computer Society, 2014.
3. K. R. Apt. Ten years of Hoare's logic: A survey – Part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, Oct. 1981.
4. C. Artho, K. Havelund, and A. Biere. High-level data races. *Softw. Test., Verif. Reliab.*, 13(4):207–227, 2003.
5. 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.
6. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices Workshop (CASSIS 2004)*, volume 3362 of *LNCS*, pages 151–171. Springer-Verlag, 2005.
7. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
8. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. de Boer, M. Bonsangue, S. Graf, and W. de Roever, editors, *Formal Methods for Components and Objects (FMCO), 4th International Symposium 2005*, volume 4111 of *LNCS*, pages 115–137. Springer-Verlag, 2006.

9. A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: A verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 113–132, New York, NY, USA, 2012. ACM.
10. S. Blom and M. Huisman. The VerCors Tool for verification of concurrent programs. In C. Jones, P. Pihlajasaari, and J. Sun, editors, *Formal Methods (FM) 2014*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
11. S. Blom, M. Huisman, and M. Mihelčić. Specification and verification of GPGPU programs, 2013. Accepted to appear in Science of Computer Programming.
12. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
13. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7:212–232, June 2005.
14. P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 342–363. Springer-Verlag, 2006.
15. 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 *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
16. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification*, pages 480–494, 2010.
17. D. R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *LNCS*, pages 472–479. Springer-Verlag, 2011.
18. B. Cowan and B. Kapralos. GPU-based acoustical occlusion modeling with acoustical texture maps. In *Proceedings of the 6th Audio Mostly Conference: A Conference on Interaction with Sound*, AM '11, pages 55–61, New York, NY, USA, 2011. ACM.
19. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
20. 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.
21. D. DiStefano and M. Parkinson. jStar: Towards practical verification for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 213–226. ACM, 2008.
22. 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.
23. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.
24. R. W. Floyd. Assigning meanings to programs. *Proc. Symp. Appl. Math*, 19:19–31, 1967.
25. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *5th Asian Conference on Programming languages and Systems*, volume 4807 of *LNCS*, pages 19–37. Springer, 2007.

26. C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In G. Ramalingam, editor, *6th Asian Conference on Programming languages and Systems*, volume 5356 of *LNCS*, pages 171–187. Springer, 2008.
27. C. Haack, M. Huisman, C. Hurlin, and A. Amighi. Permission-based separation logic for Java. Submitted to Logical Methods in Computer Science.
28. C. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
29. A. Hobor and C. Gherghina. Barriers in concurrent separation logic. In G. Barthe, editor, *20th European Symposium on Programming (ESOP)*, volume 6602 of *LNCS*, pages 276–296. Springer, 2011.
30. M. Huisman and M. Mihelčić. Specification and verification of GPGPU programs using permission-based separation logic, 2013.
31. K. Huizing and R. Kuiper. Verification of object oriented programs using class invariants. In *Fundamental Approaches to Software Engineering*, pages 208–221, 2000.
32. B. Jacobs, F. Piessens, K. R. M. Leino, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Software Engineering and Formal Methods*, pages 137–147, 2005.
33. 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*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.
34. E. K. Jason Sanders. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.
35. Khronos OpenCL Working Group. The OpenCL specification, 2008–2013.
36. G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, Feb. 2007. Dept. of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.
37. K. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Lecture notes of FOSAD*, volume 5705 of *LNCS*. Springer, 2009.
38. K. R. M. Leino. This is Boogie 2. Technical report, Microsoft Research, June 2008.
39. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
40. G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *SIGSOFT FSE 2010, Santa Fe, NM, USA*, pages 187–196. ACM, 2010.
41. B. Liskov and J. Guttag. *Abstraction and specification in program development*. MIT Press, Cambridge, MA, USA, 1986.
42. Y. Lu, J. Potter, and J. Xue. Validity invariants and effects. In *European Conference on Object-Oriented Programming*, pages 202–226, 2007.
43. B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.
44. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.
45. J. B. Mulligan. A GPU-accelerated software eye tracking system. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA '12, pages 265–268, New York, NY, USA, 2012. ACM.
46. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

47. The OpenCL 1.2 specification, 2011.
48. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica Journal*, 6:319–340, 1975.
49. M. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, 2005.
50. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–258. ACM, 2005.
51. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.
52. T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.
53. A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. PhD thesis, Habilitation thesis, Technical University of Munich, 1997.
54. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications, Vol II.1*, pages 13–39. Kluwer, 1998.
55. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.
56. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1), 2012.
57. S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, Z.-P. Liang, and B. P. Sutton. Accelerating advanced MRI reconstructions on GPU-s. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 261–272, New York, NY, USA, 2008. ACM.
58. VerCors project homepage, 2014. `http://www.utwente.nl/vercors/`.
59. B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.
60. M. Zaharieva-Stojanovski and M. Huisman. Verifying class invariants in concurrent programs. In *Fundamental Approaches to Software Engineering*, volume 8411 of *LNCS*, pages 230–244. Springer, 2014.