Permission-Based Separation Logic for Multithreaded Java Programs[⋆]

Christian Haack[1]⋆⋆, Marieke Huisman[2]⋆⋆⋆, and Clément Hurlin[3]†

[1] aicas GmbH, Karlsruhe, Germany
[2] University of Twente, Netherlands
[3] IRISA/Université de Rennes 1, France

Abstract. This paper motivates and presents a program logic for reasoning about multithreaded Java-like programs with concurrency primitives such as dynamic thread creation, thread joining and reentrant object monitors. The logic is based on concurrent separation logic. It is the first detailed adaptation of concurrent separation logic to a multithreaded Java-like language.
The program logic associates a unique static access permission with each heap location, ensuring exclusive write accesses and ruling out data races. Concurrent reads are supported through fractional permissions. Permissions can be transferred between threads upon thread starting, thread joining, initial monitor entrancies and final monitor exits.
This paper presents the basic principles to reason about thread creation and thread joining. It finishes with an outlook how this logic will evolve into a full-fledged verification technique for Java (and possibly other multithreaded languages).

1 Introduction

1.1 Motivation and Context

In the last decade, researchers have spent great efforts on developing advanced program analysis tools for popular object-oriented programming languages, like Java or C#. Such tools include software model-checkers [28], static analysis tools for data race and deadlock detection [21,22], type-and-effect systems for atomicity [9,1], and program verification tools based on interactive theorem proving [15]. A particularly successful line of research is concerned with static contract checking based on Hoare logic. Examples include ESC/Java [8] — a highly automatic, but deliberately unsound, tool based on a weakest precondition calculus and an

⋆ This work was funded in part by the 6th Framework programme of the EC under the MOBIUS project IST-FET-2005-015905.

⋆⋆ Part of the work done while the author was at Radboud University Nijmegen, Netherlands.

⋆⋆⋆ Part of the work done while the author was at INRIA Sophia Antipolis – Méditerranée, France.

† Part of the work done while the author was at INRIA Sophia Antipolis – Méditerranée, France, and visiting the University of Twente, Netherlands; and then at INRIA – Bordeaux Sud-Ouest, France and Microsoft R&D, France.

SMT solver, the Key tool [4] — a sound verification tool for Java programs based on dynamic logic and symbolic execution, and Spec# [3] — a verification tool for C# programs that achieves modular soundness by imposing a dynamic object ownership discipline. While still primarily used in academics, these tools are mature and usable enough, so that programmers other than the tool developers can employ them for constructing realistic, verified programs. A restriction, however, is that their support for concurrency is limited. As most real-world applications written in Java or C# are multithreaded, this limitation is a serious obstacle for bringing assertion-based verification to the real world. Support for concurrency is therefore the most important next step.

What makes verification of shared-variable concurrent programs difficult is the possibility of thread interference. Any assertion that has been established by one thread can potentially be invalidated by any other thread at any time. Some traditional program logics for shared-variable concurrency, *e.g.*, Owicki-Gries [25] or Jones's rely-guarantee method [18], account for thread interference in the most general way. Unfortunately, the generality of these logics makes them tedious to use, perhaps even unsuitable as a practical foundation for verifying Java-like programs. In comparison to these logics, Hoare's logics for conditional critical regions [13] and monitors [14] are much simpler, because they rely on syntactically enforceable synchronization disciplines that limit thread interference to few synchronization points (see [2] for a survey).

Because Java's main thread synchronization mechanism is based on monitors, Hoare's logic for monitors is a good basis for the verification of Java-like programs. Unfortunately, however, a safe monitor synchronization discipline cannot be enforced syntactically for Java. This is so, because Java threads typically share heap memory including possibly aliased variables. Recently, O'Hearn [23] generalized Hoare's logic to programming languages with heap. To this end, he extended a new program logic, called *separation logic* [17,27], which had previously been used for reasoning about sequential pointer programs. O'Hearn's *concurrent separation logic (CSL)* enforces correct synchronization of heap accesses *logically*, rather than *syntactically*. Logical enforcement of correct synchronization has the desirable consequence that all CSL-verified programs are guaranteed to be data-race free. In this paper, we adapt CSL to a Java-like language.

Adapting CSL to Java requires a number of substantial extensions: Firstly, while O'Hearn's CSL assumes a static set of locks, in Java locks have the same status as other objects that are dynamically allocated and stored on the heap, and can be aliased. Secondly, while O'Hearn's CSL assumes structured parallelism[1], Java threads are based on thread identifiers (represented by thread objects) that are dynamically allocated on the heap, can be stored on the heap and can be aliased. A join-operation that is parametrized by a thread identifier allows threads to wait for the termination of other threads. Thirdly, while O'Hearn's CSL assumes that programs go through a global initialization phase to establish

[1] For presentational reasons, O'Hearn's paper [23] assumes a fixed, static set of threads, but remarks that this can be generalized to structured parallelism, as done by Brookes [7].

all invariants, this assumption is inappropriate for Java programs where objects and locks are created dynamically and, consequently, initialize their invariants dynamically. Fourthly, while O'Hearn considers classical monitors that cannot be reentered, Java's monitors are reentrant. Reentrant monitors have the advantage of avoiding deadlocks due to attempted reentrancy. Such deadlocks would, for instance, occur when synchronized methods call synchronized methods on the current self: a very common call-pattern in Java. Fifthly, O'Hearn's CSL does not allow multiple threads to read the same location simultaneously. This is more restrictive than necessary: to avoid data races read-write and write-write conflicts must be avoided, but concurrent reads are harmless.

CSL has since been extended in various directions to overcome some of these limitations. For instance, Bornat and others have combined separation logic with permission accounting in order to support concurrent reads [5], while Gotsman and others have generalized concurrent separation logic to cope with Posix-style threads and locks [10]. This paper takes the ideas from concurrent separation logic into another direction, namely towards reasoning about multithreaded *Java-like* programs. The resulting proof system supports Java's main concurrency primitives: dynamically created threads and monitors that can be stored on the heap, thread joining, and monitor reentrancy, thus allowing reasoning about multithreaded programs written in Java. Since the use of Java is widespread (*e.g.*, internet applications, mobile phones and smart cards), this is an important step towards reasoning about realistic software.

1.2 Separation Logic Informally

We first informally present the features of separation logic that are most important for our logic.

Formulas as Access Tickets Separation logic [27] combines the usual logical operators with the points-to predicate $x.f \mapsto v$ and the resource conjunction $F * G$.

The predicate $x.f \mapsto v$ has a *dual purpose*: firstly, it asserts that the object field $x.f$ contains data value v and, secondly, it represents a *ticket* that grants permission to access the field $x.f$. This is formalized by separation logic's Hoare rules for reading and writing fields (where $x.f \mapsto \_$ is short for $(\exists v)(x.f \mapsto v)$):

$$\{x.f \mapsto \_\}x.f = v\{x.f \mapsto v\} \qquad \{x.f \mapsto v\}y = x.f\{x.f \mapsto v \;\texttt{*}\; v \texttt{==} y\}$$

The crucial difference to standard Hoare logic is that both rules have a precondition of the form $x.f \mapsto \_$ that functions as an *access ticket* for $x.f$.

It is important that tickets are not forgeable: one ticket is not the same as two tickets! For this reason, the resource conjunction `*` is not idempotent: F is not equivalent to $F * F$. Intuitively, the formula $F * G$ represents two access tickets F and G to *separate* parts of the heap. In other words, the part of the heap that F permits to access is *disjoint* from the part of the heap that G permits to access. As a consequence, separation logic's `*` implicitly excludes interfering heap accesses through aliases: this is why the Hoare rules shown above are sound. It is noteworthy that given two objects `x` and `y` with field `f`, the assertion $\texttt{x.f} \mapsto \_ * \texttt{y.f} \mapsto \_$ does not mean the same as $\texttt{x.f} \mapsto \_ \wedge \texttt{y.f} \mapsto \_$: the first assertion implies that `x` and `y` are distinct, while the second assertion can be satisfied even if `x` and `y` are aliases.

Local Reasoning A crucial feature of separation logic is that it allows to reason locally about methods. This means that, when calling a method, one can identify (1) the (small) part of the heap accessed by that method and (2) the rest of the heap that is left unaffected. Formally, this is expressed by the (Frame) rule:

$$\frac{\{F\}c\{F'\}}{\{F * G\}c\{F' * G\}}\ \text{(Frame)}$$

This rule expresses that given a command c which only accesses the part of the heap described by F, one can reason locally about command c ((Frame)'s premise) and deduce something globally, i.e., in the context of a bigger heap $F * G$ ((Frame)'s conclusion). In this rule, G is called the frame and represents the part of the heap unaffected by executing c. It is important that the (Frame) rule can be added to our verification rules without harming soundness.

1.3 Contributions

Using the aspects of separation logic described above, we have developed a program logic for a concurrent language with Java's main concurrency primitives. Our logic combines separation logic with fraction-based permissions. This results in an expressive and flexible logic, which can be used to verify many realistic applications. The logic ensures the absence of data races, but is not overly restrictive, as it allows concurrent reads.

Because of the use of fraction-based permission permissions, as proposed by Boyland [6], our program logic prevents data races, but allows multiple threads to read a location simultaneously. Permissions are fractions in the interval $(0, 1]$. Each access to the heap is associated with a permission. If a thread has full permission (*i.e.*, with value 1) to access a location, it can write this location, because the thread is guaranteed to have exclusive access to it. If a thread has a partial permission (less than 1), it can read a location. However, since other threads might also have permission to read the same location, a partial permission does not allow to write a location. Soundness of the approach is ensured by the guarantee that the total permissions to access a location are never more than 1.

Permissions can be transferred from one thread to another upon thread creation and thread termination. If a new thread is forked, the parent thread transfers the necessary permissions to this new thread (and thus the creating thread abandons these permissions, to avoid permission duplication). Once a thread terminates, its permissions can be transferred to the remaining threads. The mechanism for doing this in Java is by joining a thread: if a thread t joins another thread u, it blocks until u has terminated. After this, t can take hold of u's permissions. In order to soundly account for permissions upon thread joining, a special join-permission is used. Only threads that hold (a fraction of) this join-permission can take hold of (the same fraction of) the permissions that have been released by the terminating thread. Note that, contrary to Posix threads, Java threads allow multiple joiners of the same thread. Our logic supports multiple thread joiners. For example, the logic can verify programs where multiple threads join the same thread t in order to gain shared read-access to the part of the heap that was previously owned by t.

Just as in O'Hearn's approach [23], locks are associated with so-called resource invariants. If a thread acquires a lock, it may assume the lock's resource invariant and obtain access to the resource invariant's footprint (i.e., to the part of the heap that the resource invariant depends on). If a thread releases a lock, it has to establish the lock's resource invariant and transfers access to the resource invariant's footprint back to the lock. Previous variants of concurrent separation logic prohibit threads to acquire locks that they already hold. In contrast, Java's locks are reentrant. Our program logic supports reentrant locks. To this end, the logic distinguishes between initial lock entries and lock reentries. Permissions are transferred upon initial lock entries only, but not upon reentries.

1.4 Overview

This paper describes the main ideas of our permission-based separation logic to reason about multithreaded Java programs. Section 2 introduces the specification language and basic proof rules for single-threaded programs. Section 3 extends this to multithreaded programs with dynamic thread creation and termination. Finally, Section 4 concludes and discusses future work. This paper does not present formally how the logic handles reentrant locks, and it also leaves out how we use abstract predicates [26] to model encapsulation of objects, and inheritance of specifications. For full details, we refer to the full version of this paper [11] and Hurlin's PhD thesis [16].

2 Separation Logic for a Java-like Language

As mentioned above, we have developed the program logic for a Java-like languages. The semantics of this language is standard, and for space reasons we therefore do not give a formal definition of this language. This section discusses how we reason about sequential programs written in this language; the next sections extends this to a multithreaded setting. We first present separation logic formulas formally, and then present the main ingredients of the proof system.

2.1 Separation Logic

To write method contracts, we use *intuitionistic* separation logic [17,27,26]. This is most suitable to reason about properties that are invariant under heap extensions, and to reason about garbage-collected languages like Java. Contrary to classical separation logic, intuitionistic separation logic admits weakening. Informally, this means that one can "forget" a part of the state, which makes it appropriate for garbage-collected languages.

Specification formulas F are defined by the following grammar:

$$\begin{array}{rl} lop \in \{\texttt{*}, \texttt{-*}, \texttt{\&}, \texttt{|}\} & \qquad qt \in \{\texttt{ex}, \texttt{fa}\} \\ F \in \textsf{Formula} ::= & e \mid \texttt{PointsTo}(e.f, \pi, e) \mid F \; lop \; F \mid \texttt{(}qt \; T \; \alpha\texttt{)(}F\texttt{)} \end{array}$$

We now explain these formulas:

The *points-to predicate* `PointsTo(`$e.f, \pi, v$`)` is ASCII for $e.f \overset{\pi}{\longmapsto} v$ [5]. Superscript π must be a fractional permission [6] i.e., a fraction $\frac{1}{2^n}$ in the interval $(0, 1]$. As explained earlier, formula `PointsTo(`$e.f, \pi, v$`)` has a dual meaning: firstly, it asserts that field $e.f$ contains value v, and, secondly, it represents access right π to $e.f$. Permission $\pi = 1$ grants write access, while any permission π grants read access.

The *resource conjunction* F `*` G (a.k.a *separating conjunction*) expresses that resources F and G are independently available: using either of these resources leaves the other one intact. Resource conjunction is not idempotent: F does *not* imply F `*` F. Because Java is a garbage-collected language, we allow dropping assertions: F `*` G implies F.

The *resource implication* F `-*` G (a.k.a. *linear implication* or *magic wand*) means "consume F yielding G". Resource F `-*` G permits to trade resource F to receive resource G in return. Resource conjunction and implication are related by the modus ponens: F `* (`F `-*` G`)` implies G. Most related work omit the magic wand. We include it, because it can be added without any difficulties, and we found it useful to specify programming patterns such as iterators [12].

Quantified formulas have the shape `(`$qt\ T\ \alpha$`)(`F`)`, where qt is a universal or existential quantifier, α is a variable whose scope is formula F, and T is α's type.

2.2 Hoare Triples

Most Hoare rules to reason about sequential programs are standard, we only discuss the most important ones. Appendix B of Hurlin's PhD thesis [16] lists the complete collection.

First, we present the rule for field writing . The rule's precondition[2] requires that the heap contains at least the object dereferenced and the field mentioned. In addition, it requires permission 1 to this object's field, i.e., write-permission. The rule's postcondition simply ensures that the heap has been updated with the value assigned. It should be noted that this rule is *small* [24]: it does not require anything more than a single `PointsTo` predicate. The frame rule (discussed in Section 1.2) is used to build proofs in bigger contexts.

$$\frac{\Gamma \vdash u, w : U, W \quad W\ f \in \mathsf{fld}(U)}{\Gamma; v \vdash \{\texttt{PointsTo(}u.f, 1, W\texttt{)}\}u.f \texttt{=} w\{\texttt{PointsTo(}u.f, 1, w\texttt{)}\}}\ \text{(Fld Set)}$$

The rule for field reading requires a `PointsTo` predicate with *any* permission π:

$$\frac{\Gamma \vdash u, \pi, w : U, \texttt{perm}, W \quad W\ f \in \mathsf{fld}(U) \quad W <: \Gamma(\ell)}{\Gamma; v \vdash \{\texttt{PointsTo(}u.f, \pi, w\texttt{)}\}\ell \texttt{=} u.f\{\texttt{PointsTo(}u.f, \pi, w\texttt{)} \texttt{*} \ell \texttt{==} w\}}\ \text{(Get)}$$

The rule for creating new objects has precondition `true`, because we do not check for out of memory errors. After creating an object, all its fields are writable: the predicate ℓ`.init` abbreviates a `*`-conjunction of the predicates `PointsTo(`$\ell.f, 1, \mathsf{df}(T)$`)` for all fields $T\ f$ in ℓ's class:

[2] Where `PointsTo(`$u.f, 1, W$`)` abbreviates `(ex` $W\ w$`)(PointsTo(`$u.f, 1, w$`))`, as defined in Section 2.1.

$$\frac{C\texttt{<}\bar{T}\,\bar{\alpha}\texttt{>} \in ct \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\alpha] \quad C\texttt{<}\bar{\pi}\texttt{>} <: \Gamma(\ell)}{\Gamma; v \vdash \{\texttt{true}\}\ell\,\texttt{=}\,\texttt{new}\, C\texttt{<}\bar{\pi}\texttt{>}\{\ell.\texttt{init} \,\texttt{*}\, C \;\texttt{classof}\; \ell\}} \text{(New)}$$

The rule for method calls is verbose, but standard: it looks up the method specification, type checks the method call, requires the precondition to hold before the call, and ensures the postcondition holds after the call.

$$\frac{\begin{array}{c}\mathsf{mtype}(m, t\texttt{<}\bar{\pi}\texttt{>}) = \texttt{<}\bar{T}\,\bar{\alpha}\texttt{>}\;\texttt{requires}\, G;\texttt{ensures}\;\texttt{(ex}\, U\,\alpha'\texttt{)}\,\texttt{(}G'\texttt{);}\\ U\; m\;\texttt{(}t\texttt{<}\bar{\pi}\texttt{>}\;\imath_0 \bar{W}\bar{\imath}\texttt{)}\\ \sigma = (u/\imath_0, \bar{\pi}'/\bar{\alpha}, \bar{w}/\bar{\imath}) \quad \Gamma \vdash u, \bar{\pi}', \bar{w} : t\texttt{<}\bar{\pi}\texttt{>}, \bar{T}[\sigma], \bar{W}[\sigma] \quad U[\sigma] <: \Gamma(\ell)\end{array}}{\Gamma; v \vdash \{u \,\texttt{!=}\,\texttt{null} \,\texttt{*}\, G[\sigma]\}\ell\,\texttt{=}\,u.m\texttt{(}\bar{w}\texttt{)}\{\texttt{(ex}\, U[\sigma]\;\alpha'\texttt{)}\,\texttt{(}\alpha' \,\texttt{==}\, \ell \,\texttt{*}\, G'[\sigma]\texttt{)}\}} \text{(Call)}$$

Soundness of the logic is proven via a preservation theorem [16]. As corollary we can prove that any verified program never dereferences null, and is partially correct. The latter means that if a verified program contains a specification command `assert(`F`)`, then F holds whenever the assertion is reached at runtime:

Theorem 1 (Partial Correctness).
If $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow^{*}_{ct} \langle h, \texttt{assert(}F\texttt{);}\ c,\ s\rangle$*, then* $(\Gamma \vdash \mathcal{E}; (h, \mathcal{P}); s \models F[\sigma])$ *for some* $\Gamma, \mathcal{E} = \mathcal{F}_{ct}(\mathcal{E}), \mathcal{P}$ *and* $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$.

3 Separation Logic for dynamic threads

This section discusses how the logic is extended to reason about a multithreaded language with fork and join primitives, à la Java. The rules allow to transfer permissions between threads upon thread creation and termination. The resulting program logic is still sound.

We assume that class tables always contain a declaration of class `Thread`, where class `Thread` contains methods `fork`, `join`, and `run`. As in Java, the methods `fork` and `join` are assumed to be implemented natively; their behavior is specified as follows:

– o.`fork()` creates a new thread, whose thread identifier is o, and executes o.`run()` in this thread. Method `fork` should not be called more than once on o: any subsequent call results in blocking of the calling thread.
– o.`join()` blocks until thread o has terminated.

The `run`-method is meant to be overridden while methods `join` and `fork` should not be overridden (in this simplified setting).

3.1 Separation Logic for fork/join

To extend our assertion language to deal with `fork` and `join` primitives, we introduce a `Join` predicate that controls how threads access postconditions of terminated threads.

The `Join` *predicate* To model `join`'s behavior, we add a new formula to the assertion language defined in Section 2.1. This formula will be used to govern exchange of permissions from terminated threads to alive threads:

$$F \;::=\; \ldots \mid \texttt{Join(}e, \pi\texttt{)} \mid \ldots$$

The intuitive meaning of $\texttt{Join}(e, \pi)$ is as follows: it allows to pick up fraction π of thread e's postcondition after e has terminated. As a specific case, if π is 1, the thread in which the `Join` predicate appears can pick up thread e's entire postcondition when e terminates.

3.2 Contracts for fork and join

Since we can specify contracts in the program logic for `fork` and `join` in class `Thread`, we do not need to give new Hoare rules for them. Instead, rules for `fork` and `join` are simply instances of the rule for method call. The contracts for `fork` and `join` model how permissions to access the heap are exchanged between threads. Intuitively, newly created threads obtain a part of the heap from their parent thread. Dually, when a terminated thread is joined, (a part of) its heap is transferred to the joining threads.

Specifications for Class `Thread`. To specify the methods in class `Thread` we use so-called abstract predicates `preFork` and `postJoin`. For a more precise definition of those predicates, we refer to the full version of this paper [11]. Class `Thread` is specified as follows:

```
class Thread extends Object{
 pred  preFork = true;
 pred postJoin<perm p> = true;
 requires preFork; ensures true;
 final void fork();
 requires Join(this,p); ensures postJoin<p>;
 final void join();
 final requires preFork; ensures postJoin<1>;
 void run() { null }
}
```

Predicates `preFork` and `postJoin` describe the pre- and postcondition of `run`, respectively. Notice that the contracts of `fork`, `join`, and `run` are tightly related: (1) `fork`'s precondition is similar to `run`'s precondition and (2) `run`'s postcondition includes predicate `postJoin<1>` while `join`'s postcondition is `postJoin<p>`. Point (1) models that when a thread is forked, its `run` method is executed: part of the parent thread's state is transferred to the forked thread. Point (2) models that `join` returns after `run` terminates. Further, (2) represents that threads joining a thread might pick up a part of the joined thread's state. The fact that permission `p` appears both as an argument to `Join` and to `postJoin` (in `join`'s contract) models that joining threads pick up a part of the terminated thread's state which is *proportional* to `Join`'s argument. Because one $\texttt{Join}(o, 1)$ predicate is issued per thread o, and this cannot be duplicated, our system enforces that threads joining o do not pick up more than thread o's postcondition. Method `run`'s contract in class `Thread` is fixed, which means that programmers have to specify `run` by adapting the predicates `preFork` and `postJoin`. In our examples, this proved to be convenient; however we have not investigated consequences of this choice on more intricate examples.

Since `run`'s contract is fixed, `run`'s contract cannot be parameterized by logical parameters. One could consider that this reduces expressiveness. But this is wrong, in fact it would be unsound to allow logical parameters for method `run`. As `run`'s pre and postconditions are interpreted in different threads, one cannot guarantee that logical parameters are instantiated in a similar way between callers to `fork` and callers to `join`. Hence, logical parameters have to be forbidden for `run`.

We highlight that method `run` can also be called directly, without forking a new thread. Our system allows such behavior which is used in practice to flexibly control concurrency (cf Java's `Executors` [20]).

Soundness of the logic has also been proven for this extension. In addition to absence of null dereferencing and partial correctness, we can also prove that any verified program is free of data races.

4 Conclusions and Future Work

In this paper, we have presented a variant of permission-based separation logic that allows to reason about object-oriented concurrent programs with dynamic threads. The full logic also allows to reason about reentrant locks and supports abstract predicates, see [11]. The main selling point of this logic is that it combines several existing specification techniques, *and* that it is not developed for an idealized programming language. Together this makes it powerful and practical enough to reason about real-life concurrent Java programs.

An essential ingredient of the logic is the use of permissions. These ensure that in a verified program, data races cannot occur, while multiple simultaneous reads are allowed. Thus concurrent execution of the program is restricted as little as possible.

The work described in this paper will be continued in the VerCors project, see `http://fmt.cs.utwente.nl/projects/VerCors/`. A first point for future work is to develop tool support for the existing logic. This involves several topics: (1) improving readability of the specification language, for example by extending an existing specification language such as JML [19]; (2) development of appropriate proof theories to automatically discharge proof obligations; and (3) development of techniques to reason about the absence of aliasing in the context of lock-reentrancy. We also plan to study whether permission annotations can be generated, instead of being written by the programmer.

In the longer term, we plan to study how the logic can be used in a more flexible way for concurrent data structures. In particular, specifications should be split into a functional and a concurrency part, in such a way that changing the locking policy or concurrency or synchronization primitives of an implementation would only affect validity of the concurrency specification, and not of the functional specification. Thus, if correctness of a program depends only on the functional specification of the data structure, then the change in the data structure's concurrency mechanism does not change correctness of the program. Eventually, this should also lead to a technique to reason about lock-free data structures, where some benign data races may be explicitly allowed by the logic.

References

1. M. Abadi, C. Flanagan, and S. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. on Prog. Languages and Systems*, 28(2):207–255, 2006.
2. G. Andrews. *Concurrent Programming: Principles and Practice.* 1991.
3. M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *J. of Object Technology*, 3(6):27–56, 2004.
4. B. Beckert, R. Hähnle, and P.H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* Number 4334 in LNCS. Springer, 2007.
5. R. Bornat, P.W. O'Hearn, C. Calcagno, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages*, pages 259–270. ACM Press, 2005.
6. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
7. S. Brookes. A semantics for concurrent separation logic. In *Conference on Concurrency Theory*, volume 3170 of *LNCS*, pages 16–34. Springer, 2004.
8. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Languages Design and Implementation*, volume 37, pages 234–245, June 2002.
9. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Programming Languages Design and Implementation*, volume 38 of *ACM SIGPLAN Notices*, pages 338–349. ACM Press, May 2003.
10. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In Z. Shao, editor, *Asian Programming Languages and Systems Symposium*, volume 4807 of *LNCS*, pages 19–37. Springer, 2007.
11. C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for multithreaded Java programs. Submitted, 2010.
12. C. Haack and C. Hurlin. Resource usage protocols for iterators. *J. of Object Technology*, 8(4):55–83, 2009.
13. C.A.R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, pages 61–71, New York, NY, USA, 1972. Academic Press.
14. C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
15. M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle.* PhD thesis, Computing Science Institute, University of Nijmegen, 2001.
16. C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic.* PhD thesis, Université Nice Sophia Antipolis, 2009.
17. S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, pages 14–26, 2001.
18. C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. on Prog. Languages and Systems*, 5(4):596–619, 1983.
19. G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D.R. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, Feb. 2007. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.
20. Sun Microsystems. Java's documentation: `http://java.sun.com/`.
21. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Programming Languages Design and Implementation*, pages 308–319. ACM Press, 2006.
22. M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, pages 386–396, 2009.
23. P.W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.
24. P.W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001. Invited paper.
25. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica Journal*, 6:319–340, 1975.
26. M. Parkinson. *Local Reasoning for Java.* PhD thesis, Univ. of Cambridge, 2005.
27. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
28. W. Visser, K. Havelund, G.P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.