

# Program Verification Under Weak Memory Consistency Using Separation Logic

Viktor Vafeiadis<sup>(✉)</sup>

MPI-SWS, Kaiserslautern, Saarbrücken, Germany  
viktor@mpi-sws.org

**Abstract.** The semantics of concurrent programs is now defined by a weak memory model, determined either by the programming language (e.g., in the case of C/C++11 or Java) or by the hardware architecture (e.g., for assembly and legacy C code). Since most work in concurrent software verification has been developed prior to weak memory consistency, it is natural to ask how these models affect formal reasoning about concurrent programs.

In this overview paper, we show that verification is indeed affected: for example, the standard Owicki-Gries method is unsound under weak memory. Further, based on concurrent separation logic, we develop a number of sound program logics for fragments of the C/C++11 memory model. We show that these logics are useful not only for verifying concurrent programs, but also for explaining the weak memory constructs of C/C++.

## 1 Introduction

In a uniprocessor machine with a non-optimizing compiler, the semantics of a concurrent program is given by the set of interleavings the memory accesses of its constituent threads, a model which is known as *sequential consistency* (SC) [15]. In multiprocessor machines and/or with optimizing compilers, however, more behaviors are possible; they are formally described by what is known as a weak memory model. Simple examples of such “weak” behaviors are in the SB (store buffering) and LB (load buffering) programs below:

$$\begin{array}{ll} \begin{array}{l} x := 1; \\ a := y; \text{ // } 0 \end{array} \parallel \begin{array}{l} y := 1; \\ b := x; \text{ // } 0 \end{array} & \text{(SB)} \qquad \begin{array}{l} a := x; \text{ // } 1 \\ y := 1; \end{array} \parallel \begin{array}{l} b := y; \text{ // } 1 \\ x := 1; \end{array} & \text{(LB)} \end{array}$$

Assuming all variables are 0 initially, the weak behaviours in question are the ones in which  $a$  and  $b$  have the values mentioned in the program comments. In the SB program on the left this behaviour is allowed by all existing weak memory models, and can be easily explained in terms of reordering: the hardware may execute the independent store to  $x$  and load from  $y$  in reverse order. Similarly, the behaviour in the LB program on the right, which is allowed by some memory models, can be explained by reordering the load from  $x$  and the subsequent store to  $y$ . This explanation remains the same whether the hardware itself performs

out-of-order execution, or the compiler, as a part of its optimisation passes, performs these transformations, and the hardware runs a reordered program.

In this paper, we will address two questions:

1. How do such non-SC behaviours affect existing techniques for verifying concurrent programs?
2. How can we verify concurrent programs in spite of weak memory behaviours?

For the first question, we will note that even rather basic proof methods for SC concurrency are unsound under weak memory. Specifically, in Sect. 2, we will show that this is the case for the Owicki-Gries (OG) proof method [21].

To answer the second question, there are two main approaches. One approach is to determine a class of programs for which weak memory consistency does not affect their correctness. One such a class of programs are data-race-free (DRF) programs, namely programs that under SC semantics have no concurrent conflicting accesses (two accesses to the same location, at least one of which a write). Ensuring that a memory model ascribes only SC behaviours to DRF programs has become a standard sanity requirement for weak memory models [1]. For specific memory models, one can develop larger classes of programs, whose behaviour is unaffected by the weak memory consistency (e.g., [3, 12, 20]).

An alternative approach is to develop proof techniques for reasoning directly about programs under a certain weak memory model. To do so, we usually take an existing proof technique that has been developed for SC concurrency and adapt it to make it sound under a specific weak memory model. We may then further extend the method to make the proof technique more useful for reasoning about specific weak memory features. As an example of this approach, in [13], we applied it to the OG proof method by weakening OG’s non-interference check to restore its soundness under *release-acquire* (RA) consistency.

In this paper, we will focus on this latter approach, but apply it to *concurrent separation logic* (CSL) [19]. Compared with OG, CSL is much better suited for reasoning under weak memory consistency, because by default it can reason only about DRF programs. As such, it is trivially sound under weak memory. We will then gradually extend CSL with features suitable for reasoning about the various synchronisation primitives provided by C11, and conclude with a discussion of some remaining challenges. In order to keep the exposition as simple as possible, I will elide inessential technical details and not discuss the soundness proofs of the presented proof rules. The missing details can be found in [5, 6, 25, 27, 28].

## 2 Owicki-Gries is Unsound Under Weak Memory!

To motivate why developing program logics for weak memory consistency is non-trivial, we start by showing that the Owicki-Gries (OG) system is unsound.

In 1976, Owicki and Gries [21] introduced a proof system for reasoning about concurrent programs, which formed the basis of rely/guarantee reasoning. Their system includes the usual Hoare logic rules for sequential programs, a rule for introducing auxiliary variables, and the following parallel composition rule:

$$\frac{\{P_1\} c_1 \{Q_1\} \quad \{P_2\} c_2 \{Q_2\} \quad \text{the two proofs are non-interfering}}{\{P_1 \wedge P_2\} c_1 \parallel c_2 \{Q_1 \wedge Q_2\}}$$

This rule allows one to compose two verified programs into a verified concurrent program that assumes both preconditions and ensures both postconditions. The soundness of this rule requires that the two proofs are *non-interfering*, namely that every assertion  $R$  in the one proof is stable under any  $\{P\}x := e$  (guarded) assignment in the other and vice versa; i.e., for every such pair,  $R \wedge P \vdash R[e/x]$ .

The OG system relies quite heavily on sequential consistency. In fact, OG is complete for verifying concurrent programs under SC [22], and is therefore unsound under any weakly consistent memory semantics. Auxiliary variables are instrumental in achieving completeness—without them, OG is blatantly incomplete; e.g., it cannot verify that  $\{x = 0\} x := x + 1 \parallel x := x + 1 \{x = 2\}$  where “ $:=$ ” denotes atomic assignment.

Nevertheless, many useful OG proofs do not use auxiliary variables, and one might wonder whether such proofs are sound under weak memory models. This is sadly not the case. Figure 1 presents an OG proof that the SB program cannot return  $a = b = 0$  whereas under all known weak memory models it can in fact do so. Intuitively speaking, the proof is invalid under weak memory because the two threads may have different views of memory before executing each command. Thus, when thread II terminates, thread I may perform  $a := y$  reading  $y = 0$  and storing 0 in  $a$ , thereby invalidating thread II’s last assertion.

$\left\{ \begin{array}{l} \{a \neq 0\} \\ x := 1; \\ \{x \neq 0\} \\ a := y \\ \{x \neq 0\} \end{array} \right\} \parallel \left\{ \begin{array}{l} \{\top\} \\ y := 1; \\ \{y \neq 0\} \\ b := x \\ \{y \neq 0 \wedge \\ (a \neq 0 \vee b = x)\} \end{array} \right\}$	<p>The non-interference checks are straightforward. For example,</p> $y \neq 0 \wedge (a \neq 0 \vee b = x) \wedge a \neq 0$ $\vdash y \neq 0 \wedge (a \neq 0 \vee b = 1)$ <p>and</p> $y \neq 0 \wedge (a \neq 0 \vee b = x) \wedge x \neq 0$ $\vdash y \neq 0 \wedge (y \neq 0 \vee b = x)$ <p>show stability of the last assertion of thread II under <math>\{a \neq 0\}x := 1</math> and <math>\{x \neq 0\}a := y</math>.</p>
---	---

Fig. 1. OG proof that SB cannot return  $a = b = 0$ .

### 3 RC11 Preliminaries

For concreteness, we will now introduce a simple programming language containing all the features of RC11, the rectified version of the C/C++11 memory model due to Lahav et al. [14]. Programs are given by the following grammar:

$$\begin{aligned} e &::= x \mid n \mid e + e \mid e - e \mid e \leq e \mid \dots \\ c &::= \text{skip} \mid c; c \mid c \parallel c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \mid x := e \mid \\ &\quad [e]_o := e \mid x := [e]_o \mid x := \text{CAS}_o(e, e, e) \mid x := \text{alloc} \mid \text{fence}_o \\ o &::= \text{na} \mid \text{rlx} \mid \text{acq} \mid \text{rel} \mid \text{acq-rel} \mid \text{sc} \end{aligned}$$

Expressions,  $e$ , are built out of program variables, constants and arithmetic and logical operators. Commands,  $c$ , contain the empty command, sequential and parallel composition, conditionals and loops, assignments to local variables, memory accesses (loads, stores, and compare and swap), allocation, and fences.

Memory accesses are annotated with an access mode,  $o$ , which indicates the level of consistency guarantees provided by the access, which in turn determines its implementation cost.

The weakest access mode is *non-atomic* (**na**), which is intended for normal data loads and stores. Races on non-atomic accesses are treated as program errors: it is the responsibility of the programmer to ensure that such races never occur. The remaining access modes are intended for synchronisation between threads and, as such, allow races. The strongest and most expensive mode are *sequentially consistent* (**sc**) accesses, whose primary purpose is to restore the simple interleaving semantics of sequential consistency [15] if a program (when executed under SC semantics) has races only on SC accesses. Weaker than SC atomics are *acquire* (**acq**) loads and *release* (**rel**) stores,<sup>1</sup> which can be used to perform “message passing” between threads without incurring the implementation cost of a full SC access; and weaker and cheaper still are *relaxed* (**rlx**) accesses, which provide only minimal synchronisation guarantees.

RC11 also supports language-level fence instructions, which provide finer-grained control over where hardware fences are to be placed and can be used in conjunction with relaxed accesses to synchronise between threads. Fences are also annotated with an access mode,  $o \in \{\text{acq}, \text{rel}, \text{acq-rel}, \text{sc}\}$ .

We will discuss the semantics of these access modes and fences in more detail as we introduce program logic rules to reason about them.

## 4 Reasoning About Non-atomic Accesses Using CSL

We start with non-atomics, which have to be accessed in a data-race-free (DRF) fashion. To reason about them, it is natural to consider O’Hearn’s *concurrent separation logic* (CSL) [19], because it rules out data races by construction. In CSL, accessing a memory location,  $\ell$ , requires the command to have the permission to access that location in its precondition in the form of a points-to assertion,  $\ell \mapsto v$ . This formula asserts that the memory at location  $\ell$  stores the value  $v$ , moreover it gives permission to the bearer of this assertion to access and possibly modify the contents of memory at location  $\ell$ . Formally, the permission is generated by the allocation rule and required in the preconditions of the load and store rules.

$$\begin{aligned}
 \{\text{emp}\} \ x := \text{alloc} \ \{x \mapsto \_ \} & \quad (\text{ALLOC}) \\
 \{\ell \mapsto v\} \ x := [\ell]_{\text{na}} \ \{\ell \mapsto v \wedge x = v\} & \quad (\text{R-NA}) \\
 \{\ell \mapsto v\} \ [\ell]_{\text{na}} := v' \ \{\ell \mapsto v'\} & \quad (\text{W-NA})
 \end{aligned}$$

<sup>1</sup> The acquire mode is meant to be used for loads, whereas the release mode for stores: there is also a combined *acquire-release* (**acq-rel**) mode that can be used for CAS.

The load rule further asserts that the value read is the one recorded in the points-to assertion, while the store rule allows one to update this value.

$$\begin{array}{c}
\frac{}{\{P\} \mathbf{skip} \{P\}} \quad (\text{SKIP}) \\
\frac{\{P\} c_1 \{Q\} \quad \{Q\} c_2 \{R\}}{\{P\} c_1; c_2 \{R\}} \quad (\text{SEQ}) \\
\frac{\{P \wedge B\} c_1 \{Q\} \quad \{P \wedge \neg B\} c_2 \{Q\}}{\{P\} \mathbf{if} B \mathbf{then} c_1 \mathbf{else} c_2 \{Q\}} \quad (\text{IF}) \\
\frac{\{P \wedge B\} c \{P\}}{\{P\} \mathbf{while} B \mathbf{do} c \{P \wedge \neg B\}} \quad (\text{WHILE}) \\
\frac{\{P_1\} c_1 \{Q_1\} \quad \{P_2\} c_2 \{Q_2\} \quad \text{fv}(P_1, c_1, Q_1) \cap \text{wr}(c_2) = \emptyset \quad \text{fv}(P_2, c_2, Q_2) \cap \text{wr}(c_1) = \emptyset}{\{P_1 * P_2\} c_1 \parallel c_2 \{Q_1 * Q_2\}} \quad (\text{PAR}) \\
\frac{}{\{[e/x]P\} x := e \{P\}} \quad (\text{ASSIGN}) \\
\frac{\{P\} c \{Q\} \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{\{P'\} c \{Q'\}} \quad (\text{CONSEQ}) \\
\frac{\{P_1\} c \{Q\} \quad \{P_2\} c \{Q\}}{\{P_1 \vee P_2\} c \{Q\}} \quad (\text{DISJ}) \\
\frac{\{P\} c \{Q\} \quad x \notin \text{fv}(c, Q)}{\{\exists x. P\} c \{Q\}} \quad (\text{EX}) \\
\frac{\{P\} c \{Q\} \quad \text{fv}(R) \cap \text{wr}(c) = \emptyset}{\{P * R\} c \{Q * R\}} \quad (\text{FRAME})
\end{array}$$

**Fig. 2.** Proof rules of CSL (without resource invariants).

The other CSL rules are listed in Fig. 2: these include the standard rules from Hoare logic (SKIP, ASSIGN, SEQ, IF, WHILE), the parallel composition rule (PAR), the consequence rule (CONSEQ), the disjunction and existential elimination rules (DISJ, EX), and the frame rule (FRAME). In our presentation of the rules, we exclude any mention of “resource invariants” and the rules for dealing with them, as we will not ever directly use this feature of the logic. In preparation for the extensions in the next section, our formulation of the consequence rule uses ghost implication ( $\Rightarrow$ ) instead of normal logical implication. Ghost implication is a generalisation of normal implication that in addition allows frame-preserving updates to any ghost resources mentioned in the assertions.

CSL’s parallel composition rule requires the preconditions of the two threads to be disjoint (i.e.,  $P_1 * P_2$ ), which (together with the load and store rules) precludes the two threads of accessing the same location simultaneously. The disjointness conditions of the rule check that each thread does not modify any of the variables appearing in the other thread’s program or specification.

## 5 RSL: Reasoning About Release-Acquire Synchronisation

Next, let us consider C11’s *acquire loads* and *release stores*, whose main mode of use is to establish synchronisation between two threads. The basic synchronisation pattern is illustrated by the following “message passing” idiom:

$$\begin{array}{l} [x]_{\text{na}} := 1; \parallel a := [y]_{\text{acq}}; \text{ // } 1 \\ [y]_{\text{rel}} := 1; \parallel \text{ if } a \neq 0 \text{ then } b := [x]_{\text{na}}; \text{ // } 0 \end{array} \quad (\text{MP})$$

Here, assuming that initially  $[x] = [y] = 0$ , the program cannot read  $a = 1$  and  $b = 0$ . According to C11, when an acquire load reads from a release store, this results in a synchronisation. As a result, any memory access happening before the release store (by being performed previously either by the same thread or by some previously-synchronising thread) also happens before the acquire load and any access happening after it. In the MP program, this means that the  $[x]_{\text{na}} := 1$  write happens before the  $b := [x]_{\text{na}}$  load, and thus the reading thread *must* return  $b = 1$  in the case it read  $a = 1$ .

To reason about release and acquire accesses, Vafeiadis and Narayan [28] introduced *relaxed separation logic* (RSL), which extends CSL assertions with two new assertion forms:

$$P, Q ::= \dots \mid \mathbf{W}(\ell, \mathcal{Q}) \mid \mathbf{R}(\ell, \mathcal{Q})$$

These represent the permission to perform a release store or an acquire load respectively, and attach to location  $\ell$  a mapping  $\mathcal{Q}$  from values to assertions. This mapping describes the manner in which the location  $\ell$  is used by the program. We can roughly consider it as an invariant stating: “if location  $\ell$  holds value  $v$ , then the assertion  $\mathcal{Q}(v)$  is true.”

At any point in time, a non-atomic location may be converted into an atomic location with the following ghost implication:

$$\ell \mapsto v * \mathcal{Q}(v) \Rightarrow \mathbf{W}(\ell, \mathcal{Q}) * \mathbf{R}(\ell, \mathcal{Q}) \quad (\text{MK-ATOM})$$

In the antecedent of the ghost move, the invariant should hold for the value of the location; as a result, we get the permissions to write and read that location.

RSL’s release write rule

$$\{\mathbf{W}(\ell, \mathcal{Q}) * \mathcal{Q}(v)\} [\ell]_{\text{rel}} := v \{\mathbf{W}(\ell, \mathcal{Q})\} \quad (\text{W-REL})$$

says that in order to do a release write of value  $v$  to location  $\ell$ , we need to have a permission to do so,  $\mathbf{W}(\ell, \mathcal{Q})$ , and we have to satisfy the invariant specified by that permission, namely  $\mathcal{Q}(v)$ . After the write is done, we no longer own the resources specified by the invariant (so that readers can obtain them).

The acquire read rule

$$\{\mathbf{R}(\ell, \mathcal{Q})\} x := [\ell]_{\text{acq}} \{\mathbf{R}(\ell, \mathcal{Q}[x := \text{emp}]) * \mathcal{Q}(x)\} \quad (\text{R-ACQ})$$

complements the release write rule. To perform an acquire read of location  $\ell$ , one must have an acquire permission for  $\ell$ . Just as with a release permission, an acquire permission carries a mapping  $\mathcal{Q}$  from values to assertions. In case of an acquire permission, this mapping describes what resource will be acquired by reading a certain value; so if the value  $v$  is read, resource  $\mathcal{Q}(v)$  is acquired.

This rule is slightly complicated by a technical detail. In the postcondition, we cannot simply retain the full acquire permission for location  $\ell$ , because that would enable us to read the location again and acquire the ownership of  $\mathcal{Q}(v)$  a second time. To prevent this, the acquire permission's mapping in the postcondition becomes  $\mathcal{Q}[x:=\text{emp}] \triangleq \lambda y. \text{if } y=x \text{ then emp else } \mathcal{Q}(y)$ .

As a simple application of these rules, Fig. 3 shows a slightly abbreviated proof of the MP program. Initially, the rule MK-ATOM is applied to set up the invariant for location  $y$ . By the parallel composition rule, the first thread receives the permission to access  $x$  (specifically,  $x \mapsto 0$ ) and the release write permission to  $y$ , which it uses to transfer away the  $x \mapsto 1$  resource. The second thread starts with the acquire read permission and uses it to get hold of the invariant of  $y$ , which, in the case that  $a \neq 0$ , gives enough permission to the thread to access  $x$  non-atomically and establish  $b = 1$ . In the proof outline, we often use the consequence rule to forget permissions that are no longer relevant.

$$\begin{array}{c}
 \{x \mapsto 0 * y \mapsto 0\} \\
 \{x \mapsto 0 * \mathbf{W}(y, \mathcal{Q}) * \mathbf{R}(y, \mathcal{Q})\} \\
 \{x \mapsto 0 * \mathbf{W}(y, \mathcal{Q})\} \parallel \{\mathbf{R}(y, \mathcal{Q})\} \\
 [x]_{\text{na}} := 1; \quad a := [y]_{\text{acq}} \\
 \{x \mapsto 1 * \mathbf{W}(y, \mathcal{Q})\} \parallel \{(a = 0 \vee x \mapsto 1) * \mathbf{R}(y, \mathcal{Q}[a := \text{emp}])\} \\
 [y]_{\text{rel}} := 1; \quad \{a = 0 \vee x \mapsto 1\} \\
 \{\mathbf{W}(y, \mathcal{Q})\} \quad \text{if } a \neq 0 \text{ then } b := [x]_{\text{na}} \\
 \{\top\} \quad \{a = 0 \vee (x \mapsto 1 \wedge b = 1)\} \\
 \{a = 0 \vee b = 1\}
 \end{array}$$

**Fig. 3.** Proof outline of MP using the invariant  $\mathcal{Q}(v) \triangleq (v = 0 \vee x \mapsto 1)$ .

To allow multiple concurrent readers and writers, RSL's write permissions are duplicable, whereas its read permissions are splittable as follows:

$$\begin{array}{ll}
 \mathbf{W}(\ell, \mathcal{Q}) \iff \mathbf{W}(\ell, \mathcal{Q}) * \mathbf{W}(\ell, \mathcal{Q}) & (\text{W-SPLIT}) \\
 \mathbf{R}(\ell, \lambda v. \mathcal{Q}_1(v) * \mathcal{Q}_2(v)) \iff \mathbf{R}(\ell, \mathcal{Q}_1) * \mathbf{R}(\ell, \mathcal{Q}_2) & (\text{R-SPLIT})
 \end{array}$$

The reason why read permissions cannot simply be duplicated is the same as why the read permission is modified in the postcondition of the R-ACQ rule. If read permissions were made duplicable, then multiple readers would incorrectly be able to acquire ownership of the same resource.

## 6 FSL: Reasoning About Relaxed Accesses and Fences

Next, let us consider *relaxed* accesses. Unlike release stores and acquire loads, relaxed accesses do not synchronise on their own, but only when used together

with release/acquire fences. Consider the following variant of the MP example using relaxed accesses and fences.

$$\begin{array}{l|l}
 [x]_{na} := 1; & a := [y]_{rlx}; \text{ // } 1 \\
 \textbf{fence}_{rel}; & \textbf{if } a \neq 0 \textbf{ then} \\
 [y]_{rlx} := 1 & \quad \textbf{fence}_{acq}; \\
 & \quad b := [x]_{na} \\
 & \textbf{end-if}
 \end{array} \quad (\text{MP-fences})$$

Like MP, MP-fences also satisfies the postcondition,  $a = 0 \vee b = 1$  (and so do the variants where either thread is replaced by the corresponding thread of MP), but if we remove any of the fences, the program will have undefined behaviour. (The reason for the latter is that in the absence of synchronisation, the non-atomic  $x$ -accesses are racy.)

In essence, we can think of resource transfer in the following way. When releasing a resource by a combination of a release fence and a relaxed write, at the fence we should decide what is going to be released, and not use that resource until we send it away by doing the write. Conversely, when acquiring a resource using a relaxed read together with an acquire fence, once we do the read, we know which resources we are going to get, but we will not be able to use those resources until we reach the synchronisation point marked by the acquire fence.

To formally represent this intuition, *fenced separation logic* (FSL) [5] introduces two modalities into RSL's assertion language:

$$P, Q ::= \dots \mid \triangle P \mid \nabla P$$

We use  $\triangle$  to mark the resources that have been prepared to be released, and  $\nabla$  to mark those waiting for an acquire fence. We require the invariants appearing in  $\mathbf{W}(\ell, \mathcal{Q})$  and  $\mathbf{R}(\ell, \mathcal{Q})$  permissions to contain no modalities, a condition called *normalisability* in [5]. In essence, these modalities are meant to appear only in the proof outlines of individual threads and to never be nested.

FSL supports all the rules we have seen so far. In addition, it has rules for relaxed accesses and fences. The rule for relaxed writes is almost exactly the same as W-REL.

$$\{\mathbf{W}(\ell, \mathcal{Q}) * \triangle \mathcal{Q}(v)\} [\ell]_{rlx} := v \{\mathbf{W}(\ell, \mathcal{Q})\} \quad (\text{W-RLX})$$

As in W-REL, we have to have a write permission as well as the resource specified by its attached invariant. The only additional requirement is that the latter resource has to be under the  $\triangle$  modality stating that it can be released by a relaxed write. As we will later see, this ensures that any writes transferring away non-empty resources are placed after a release fence.

Similarly, the rule for relaxed reads differs from R-ACQ only in a single modality appearance:

$$\{\mathbf{R}(\ell, \mathcal{Q})\} x := [\ell]_{rlx} \{\mathbf{R}(\ell, \mathcal{Q}[x:=\text{emp}]) * \nabla \mathcal{Q}(x)\} \quad (\text{R-RLX})$$



While after acquire read, we gain ownership of the resource described by the  $\mathbf{R}$  permission, in the case of a relaxed read, we get the same resource under the  $\nabla$  modality. This makes the resource unusable before we reach an acquire fence.

The fence rules simply manage the two modalities as follows:

$$\{P\} \text{fence}_{\text{rel}} \{\triangle P\} \quad (\text{F-REL}) \quad \{\nabla P\} \text{fence}_{\text{acq}} \{P\} \quad (\text{F-ACQ})$$

Release fences protect resources that are to be released by putting them under the  $\triangle$  modality, while acquire fences clear the  $\nabla$  modality making resources under it usable.

Figure 4 shows a proof outline of MP-fences using the rules presented in this section. Except for the treatment modalities, the proof itself essentially identical to that of MP. In the first thread, we use a combination of F-REL and the frame rule to put only  $x \mapsto 1$  under the  $\triangle$  modality. In the second thread, after the relaxed load, we use the consequence rule to forget the unnecessary  $\mathbf{R}$  permission and push the  $\nabla$  modality under the disjunction.

$$\begin{array}{c}
 \{x \mapsto 0 * y \mapsto 0\} \\
 \{x \mapsto 0 * \mathbf{W}(y, \mathcal{Q}) * \mathbf{R}(y, \mathcal{Q})\} \\
 \{x \mapsto 0 * \mathbf{W}(y, \mathcal{Q})\} \\
 [x]_{\text{na}} := 1; \\
 \{x \mapsto 1 * \mathbf{W}(y, \mathcal{Q})\} \\
 \text{fence}_{\text{rel}}; \\
 \{\triangle x \mapsto 1 * \mathbf{W}(y, \mathcal{Q})\} \\
 [y]_{\text{rlx}} := 1; \\
 \{\mathbf{W}(y, \mathcal{Q})\} \\
 \{\top\}
 \end{array}
 \parallel
 \begin{array}{c}
 \{\mathbf{R}(y, \mathcal{Q})\} \\
 a := [y]_{\text{rlx}} \\
 \{\nabla(a = 0 \vee x \mapsto 1) * \mathbf{R}(y, \mathcal{Q}[a := \text{emp}])\} \\
 \{a = 0 \vee \nabla x \mapsto 1\} \\
 \text{if } a \neq 0 \text{ then} \\
 \quad \{\nabla x \mapsto 1\} \\
 \quad \text{fence}_{\text{acq}} \\
 \quad \{x \mapsto 1\} \\
 \quad b := [x]_{\text{na}} \\
 \quad \{x \mapsto 1 \wedge b = 1\} \\
 \quad \{a = 0 \vee (x \mapsto 1 \wedge b = 1)\} \\
 \{a = 0 \vee b = 1\}
 \end{array}$$

**Fig. 4.** Proof outline of MP-fences using the invariant  $\mathcal{Q}(v) \triangleq (v = 0 \vee x \mapsto 1)$ .

## 7 Reasoning About Read-Modify-Write Instructions

Next, consider compare-and-swap, which is a typical example of a read-modify-write (RMW) instruction.  $\mathbf{CAS}_o(\ell, v, v')$  reads the location  $\ell$  and if its value is  $v$ , it updates it atomically to  $v'$ . If  $\mathbf{CAS}$  reads some value other than  $v$ , then the update is not performed. In either case,  $\mathbf{CAS}$  returns the value read. The  $o \in \{\text{rlx}, \text{rel}, \text{acq}, \text{acq-rel}, \text{sc}\}$  tells us the type of event generated by a successful  $\mathbf{CAS}$  operation.

To reason about  $\mathbf{CAS}$ , we introduce a new type of assertion:

$$P, Q ::= \dots \mid \mathbf{U}(\ell, \mathcal{Q})$$

which denotes the permission to perform a **CAS** on location  $\ell$ . As with the **W** and **R** assertions, it records a mapping from values to assertions, which governs the transfer of resources via a **CAS** operation.

The **U** permission is obtained in a similar fashion as the **W** and **R** permissions. At any point in time, a non-atomic location may be converted into an atomic location with the following ghost implication:

$$\ell \mapsto v * \mathcal{Q}(v) \Rightarrow \mathbf{U}(\ell, \mathcal{Q}) \quad (\text{MK-ATOM-U})$$

The update permission **U** is duplicable, and interacts with the **W** and **R** permissions, allowing us to perform not only updates, but also reads and writes, when holding an update permission.

$$\mathbf{U}(\ell, \mathcal{Q}) \Leftrightarrow \mathbf{U}(\ell, \mathcal{Q}) * \mathbf{U}(\ell, \mathcal{Q}) \quad (\text{U-SPLIT})$$

$$\mathbf{U}(\ell, \mathcal{Q}) \Leftrightarrow \mathbf{U}(\ell, \mathcal{Q}) * \mathbf{W}(\ell, \mathcal{Q}) \quad (\text{UW-SPLIT})$$

$$\mathbf{U}(\ell, \mathcal{Q}) \Leftrightarrow \mathbf{U}(\ell, \mathcal{Q}) * \mathbf{R}(\ell, \lambda v. \text{emp}) \quad (\text{UR-SPLIT})$$

According to UW-SPLIT, when holding the  $\mathbf{U}(\ell, \mathcal{Q})$ , we also have  $\mathbf{W}(\ell, \mathcal{Q})$ , allowing us to write to  $\ell$  using the appropriate atomic write rule. On the other hand, UR-SPLIT tells us that we are allowed to read when holding the  $\mathbf{U}(\ell, \mathcal{Q})$  permission, but we cannot gain any ownership (more precisely, no matter the value read, the acquired resource will always be the empty resource **emp**).

We next consider the following rule for the acquire-release **CAS**.<sup>2</sup>

$$\frac{\begin{array}{c} \mathcal{Q}(v) \Rightarrow A * T \\ P * T \Rightarrow \mathcal{Q}(v') \end{array} \quad \begin{array}{c} \text{pure}(\varphi) \\ P * \mathcal{Q}(v) \Rightarrow \varphi \end{array}}{\{\mathbf{U}(\ell, \mathcal{Q}) * P\} x := \mathbf{CAS}_{\text{acq-rel}}(\ell, v, v') \left\{ \begin{array}{l} x = v \wedge \mathbf{U}(\ell, \mathcal{Q}) * A \wedge \varphi \vee \\ x \neq v \wedge \mathbf{U}(\ell, \mathcal{Q}) * P \end{array} \right\}} \quad (\text{CAS-AR})$$

In the precondition, we have permission to perform the **CAS** and some further resource,  $P$ , to be transferred away if the **CAS** succeeds.

If the **CAS** succeeds, we have at our disposal the resource  $\mathcal{Q}(v)$ , which is split into two parts,  $A$ , and  $T$ . Resource  $A$  is the part that we are going to acquire and keep it for ourselves in the postcondition. Resource  $T$  will remain in the invariant  $\mathcal{Q}$ . The second premise requires that the resource  $P$  (which we have in our precondition) together with the resource  $T$  (which we left behind when acquiring ownership) are enough to satisfy  $\mathcal{Q}(v')$ , thus reestablishing the invariant for the newly written value. If, in addition to merely reestablishing the invariant, we manage to prove some additional facts,  $\varphi$ , we can carry those facts into the postcondition. It is required, however, for these facts to be *pure*, meaning that the assertion  $\varphi$  is a logical fact and does not say anything about the ownership of resources or the state of the heap.

<sup>2</sup> This rule was proposed by Alex Summers and is a slightly stronger than the one in [6]. Its soundness has been established in Coq alongside with the other FSL rules.

If the **CAS** fails, then no resource transfer occurs, and the postcondition contains the same resources as the precondition.

$$\begin{aligned}
\{\mathbf{U}(\ell, \mathcal{Q}) * P\} x &:= \mathbf{CAS}_{\text{rel}}(\ell, v, v') \left\{ \begin{array}{l} x = v \wedge \mathbf{U}(\ell, \mathcal{Q}) * \nabla A \wedge \varphi \vee \\ x \neq v \wedge \mathbf{U}(\ell, \mathcal{Q}) * P \end{array} \right\} & (\text{CAS-REL}) \\
\{\mathbf{U}(\ell, \mathcal{Q}) * \triangle P\} x &:= \mathbf{CAS}_{\text{acq}}(\ell, v, v') \left\{ \begin{array}{l} x = v \wedge \mathbf{U}(\ell, \mathcal{Q}) * A \wedge \varphi \vee \\ x \neq v \wedge \mathbf{U}(\ell, \mathcal{Q}) * \triangle P \end{array} \right\} & (\text{CAS-ACQ}) \\
\{\mathbf{U}(\ell, \mathcal{Q}) * \triangle P\} x &:= \mathbf{CAS}_{\text{rlx}}(\ell, v, v') \left\{ \begin{array}{l} x = v \wedge \mathbf{U}(\ell, \mathcal{Q}) * \nabla A \wedge \varphi \vee \\ x \neq v \wedge \mathbf{U}(\ell, \mathcal{Q}) * \triangle P \end{array} \right\} & (\text{CAS-RLX})
\end{aligned}$$

**Fig. 5.** Rules for the other kinds of CAS weaker than **acq-rel**. All of these rules implicitly have the same premises as the CAS-AR rule.

The rules for the other types of **CAS** accesses are slight modifications of the CAS-AR rule in the same vein as the ones that get us from R-ACQ and W-REL to R-RLX and W-RLX (see Fig. 5). Namely, wherever the access type is relaxed,  $\triangle$  and  $\nabla$  modalities are introduced to ensure a proper fence placement. Since the premises in these rules are the same as in CAS-AR, we avoid repeating them.

- A release **CAS** is treated as a release write and a relaxed read. Therefore, in CAS-REL sends away  $P$  without any restrictions, but the acquired resource,  $A$ , is placed under the  $\nabla$  modality, requiring the program to perform an acquire fence before accessing the resource.
- Conversely, for an acquire **CAS**, the resource to be transferred away is under the  $\triangle$  modality requiring a release fence before the **CAS**, while the resource acquired is immediately usable.
- A relaxed **CAS** is relaxed as both read and write. This is reflected in the CAS-RLX rule by having both modalities in play.

$ \begin{aligned} &\text{mk-lock}() : \\ &\quad \{J\} \\ &\quad res := \text{alloc} \\ &\quad \{res \mapsto \_ * J\} \\ &\quad [res]_{\text{na}} := 0 \\ &\quad \{res \mapsto 0 * \mathcal{Q}(0)\} \\ &\quad \{\mathbf{Lock}(res)\} \\ &\text{release-lock}(\ell) : \\ &\quad \{\mathbf{Lock}(\ell) * J\} \\ &\quad [\ell]_{\text{rel}} := 0 \\ &\quad \{\mathbf{Lock}(\ell)\} \end{aligned} $	$ \begin{aligned} &\text{acquire-lock}(\ell) : \\ &\quad \{\mathbf{Lock}(\ell)\} \\ &\quad x := \mathbf{CAS}_{\text{acq}}(\ell, 0, 1); \\ &\quad \{\mathbf{U}(\ell, \mathcal{Q}) * (J \vee x \neq 0)\} \\ &\quad \mathbf{while} \ x \neq 0 \ \mathbf{do} \\ &\quad \quad \{\mathbf{U}(\ell, \mathcal{Q})\} \\ &\quad \quad \mathbf{while} \ x \neq 0 \ \mathbf{do} \ x := [\ell]_{\text{rlx}} \\ &\quad \quad \{\mathbf{U}(\ell, \mathcal{Q})\} \\ &\quad \quad x := \mathbf{CAS}_{\text{acq}}(\ell, 0, 1); \\ &\quad \quad \{\mathbf{U}(\ell, \mathcal{Q}) * (J \vee x \neq 0)\} \\ &\quad \{\mathbf{Lock}(\ell) * J\} \end{aligned} $
--	---

**Fig. 6.** Lock library verification using  $\mathcal{Q}(v) \triangleq (J \vee v \neq 0)$  and  $\mathbf{Lock}(\ell) \triangleq \mathbf{U}(\ell, \mathcal{Q})$ .

Finally, Fig. 6 presents a proof outline for verifying a spinlock implementation as an example of using the **CAS** rules. In these proof outlines, the use of the consequence rule is left implicit. Specifically, in **mk-lock**, we apply the **MK-ATOM-U** rule to generate the update permission; in **acquire-lock**, we apply the **UR-SPLIT** rule to generate a read permission, while in **release-lock**, we apply the **UW-SPLIT** rule to generate a write permission.

## 8 GPS: Adding Protocols

The assertions so far have attached an invariant,  $\mathcal{Q}$ , to each location that is meant to be used atomically. While such simple invariants suffice for reasoning about simple ownership transfer patterns, on their own they are too weak for establishing even basic coherence properties. Consider, for example, the following program, where initially  $[x] = 0$ .

$$\begin{array}{l} [x]_{\text{rlx}} := 1; \quad \parallel \quad a := [x]_{\text{rlx}}; \\ [x]_{\text{rlx}} := 2 \quad \parallel \quad b := [x]_{\text{rlx}} \end{array} \quad (\text{COH})$$

Although RC11 ensures that  $a \leq b$  in every execution of this program, it is not possible to establish this postcondition with the separation logic rules we have seen thus far. To achieve this, we need a more expressive logic incorporating some limited form of rely-guarantee reasoning (e.g., as already available in OG).

A convenient way to support such reasoning has emerged in the context of program logics for SC concurrency, such as CAP [4], CaReSL [26], TaDA [23], and Iris [9], in the form of *protocols*. The idea is to attach to each atomic location an acyclic state transition system describing the ways in which the value of the location can be updated, and to have assertions talk about the current state of a location's protocol. Formally a protocol,  $\tau$ , is a tuple  $\langle \Sigma_\tau, \sqsubseteq_\tau, \mathcal{Q}_\tau \rangle$ , where  $\Sigma_\tau$  is the (non-empty) set of protocol states,  $\sqsubseteq_\tau$  is a partial order on  $\Sigma_\tau$  relating a state to its possible future states, and  $\mathcal{Q}_\tau$  is a mapping from protocol states and values to assertions, attaching an invariant about the value of the location to each protocol state.

We extend the language of assertions with two new assertion forms:

$$P, Q ::= \dots \mid \mathbf{WP}_\tau(\ell, s) \mid \mathbf{RP}_\tau(\ell, s)$$

which assert that  $\ell$  is governed by the protocol  $\tau$  and its current state is reachable from the state  $s$ .  $\mathbf{WP}_\tau(\ell, s)$  represents an exclusive write permission to the protocol, whereas  $\mathbf{RP}_\tau(\ell, s)$  is a duplicable read permission. As usual, these permissions can be generated from a points-to assertion with a ghost move.

$$\begin{aligned} \ell &\mapsto v * \tau(s, v) \Rightarrow \mathbf{WP}_\tau(\ell, s) \\ \mathbf{WP}_\tau(\ell, s_1) * \mathbf{WP}_\tau(\ell, s_2) &\Rightarrow \text{false} \\ \mathbf{WP}_\tau(\ell, s_1) * \mathbf{RP}_\tau(\ell, s_2) &\Leftrightarrow \mathbf{WP}_\tau(\ell, s_1) \wedge s_2 \sqsubseteq_\tau s_1 \\ \mathbf{RP}_\tau(\ell, s_1) * \mathbf{RP}_\tau(\ell, s_2) &\Leftrightarrow \exists s. \mathbf{RP}_\tau(\ell, s) \wedge s_1 \sqsubseteq_\tau s \wedge s_2 \sqsubseteq_\tau s \end{aligned}$$

Consider the following two simplified proof rules for relaxed reads and writes.

$$\frac{\text{emp} \Rightarrow \mathcal{Q}_\tau(s', v) \wedge s \sqsubseteq_\tau s'}{\{\mathbf{WP}_\tau(\ell, s)\} [\ell]_{\text{rlx}} := v \{\mathbf{WP}_\tau(\ell, s')\}} \quad \frac{\forall s' \exists_\tau s. \mathcal{Q}_\tau(s', x) \Rightarrow \varphi \quad \text{pure}(\varphi)}{\{\mathbf{RP}_\tau(\ell, s)\} x := [\ell]_{\text{rlx}} \{\exists s'. \mathbf{RP}_\tau(\ell, s') \wedge \varphi\}}$$

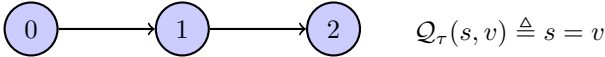
To perform a relaxed write, the thread must own the exclusive write permission for that location; it then has to chose a future state  $s'$  of the current state and establish the invariant of that state. Since it is a relaxed write, no ownership transfer is possible (at least without fences). So, in this somewhat simplified rule, we require  $\mathcal{Q}_\tau(s', v)$  to hold of the empty heap.

Conversely, to perform a relaxed read, the thread must own a shared read permission for that location stating that it is at least in state  $s$ . It then knows that the location is in some future protocol state  $s'$  of  $s$ , and gets to know that the invariant of  $\mathcal{Q}_\tau$  holds for that state and the value that it read. Since the read is relaxed, to avoid incorrect ownership transfers, the postcondition gets only the pure part of this invariant.

$$\begin{array}{c} \{x \mapsto 0\} \\ \{\mathbf{WP}_\tau(x, 0)\} \\ \{\mathbf{WP}_\tau(x, 0)\} \parallel \{\mathbf{RP}_\tau(x, 0)\} \\ [x]_{\text{rlx}} := 1; \quad \left\| \begin{array}{l} a := [x]_{\text{rlx}} \\ \{\mathbf{RP}_\tau(x, a) \wedge 0 \leq a \leq 2\} \end{array} \right. \\ \{\mathbf{WP}_\tau(x, 1)\} \parallel \{\mathbf{RP}_\tau(x, a) \wedge 0 \leq a \leq 2\} \\ [x]_{\text{rlx}} := 2; \quad \left\| \begin{array}{l} b := [x]_{\text{rlx}} \\ \{\mathbf{RP}_\tau(x, b) \wedge 0 \leq a \leq b \leq 2\} \end{array} \right. \\ \{\mathbf{WP}_\tau(x, 2)\} \parallel \{\mathbf{RP}_\tau(x, b) \wedge 0 \leq a \leq b \leq 2\} \\ \{\mathbf{WP}_\tau(x, 2) \wedge 0 \leq a \leq b \leq 2\} \end{array}$$

**Fig. 7.** Proof outline of COH using the protocol  $\langle\{0, 1, 2\}, \leq, \lambda(s, v). s = v\rangle$ .

These rules can be extended to use the FSL modalities to allow ownership transfer in combination with fences, but even these basic rules are sufficient for verifying the COH example. Returning to the example, we take as the protocol  $\tau$  of  $x$  to consist of three states ordered linearly ( $0 \sqsubseteq_\tau 1 \sqsubseteq_\tau 2$ ), each saying that  $x$  has the respective value. Pictorially, we have:



The proof outline for COH is rather straightforward and is shown in Fig. 7. In the writer thread, each write moves to the next state. In turn, the reader can assert that each read gets a value greater or equal to the last state it observed.

Besides protocols, GPS also introduced ghost state in the form of ghost resources and escrows/exchanges. These features enable GPS to support ownership transfer over release-acquire synchronization. For an explanation of these features, we refer the reader to [10, 25, 27].

*A Note About the Different Versions of GPS.* GPS was initially developed by Turon et al. [27] for a fragment of the programming language of Sect. 3 containing only non-atomic and release/acquire accesses. It was later extended by Tassarotti et al. [25] with “exchanges” and used to verify a version of the RCU algorithm. Later, Kaiser et al. [10] developed a slight variant of GPS within the Iris framework featuring a simpler “single writer” rule. All these three works had their soundness proofs verified in Coq, but cannot handle relaxed accesses. In a different line of work, He et al. [7] have extended GPS to also cover relaxed accesses albeit without a mechanised soundness proof.

## 9 Conclusion: Challenges Ahead

In this section, we will review three main challenges in this line of work. The first two have to do with the soundness of the presented logic, while the third has to do with their practical usage.

### 9.1 Soundness Under Weaker Memory Models

All the program logics discussed so far have been proved sound with respect to the RC11 weak memory model [14], which forbids load-store reordering for atomic accesses. Reordering a relaxed-atomic load past a later relaxed-atomic store, however, is allowed in some weaker memory models, such as the “promising” model of Kang et al. [11], as it is key to explaining the weak behaviour of the LB example from the introduction.

$$\begin{array}{c}
 \{x \mapsto 0 * y \mapsto 0 * \Delta z \mapsto 0\} \\
 \{W(x, Q) * R(x, Q) * W(y, Q) * R(y, Q) * \Delta z \mapsto 0\} \\
 \{R(x, Q) * W(y, Q) * \Delta z \mapsto 0\} \parallel \{W(x, Q) * R(y, Q)\} \\
 a := [x]_{\text{rlx}}; \quad b := [y]_{\text{acq}} \\
 \{Q(a) * W(y, Q) * \Delta z \mapsto 0\} \parallel \{W(x, Q) * Q(b)\} \\
 \{a = 0 \wedge W(y, Q) * \Delta Q(1)\} \parallel \{[x]_{\text{rel}} := b\} \\
 [y]_{\text{rlx}} := 1; \quad \{W(x, Q)\} \\
 \{a = 0 \wedge W(y, Q)\} \parallel \{a = 0\}
 \end{array}$$

**Fig. 8.** FSL proof outline of LB+dep where  $Q(v) \triangleq v = 0 \vee z \mapsto 0$ .

Extending the soundness of these logics to weaker models permitting the weak behaviour of LB is rather challenging. In fact, FSL with its current model of assertions (not discussed in this paper but presented in [5]) is unsound under such models as shown by the proof outline in Fig. 8.

Under the assumption that  $z \mapsto 0 * \Delta z \mapsto 0$  is unsatisfiable (used in the middle of thread I to deduce that  $Q(a) * \Delta z \mapsto 0 \implies a = 0 \wedge z \mapsto 0$ ), the proof

establishes that  $a = 0$ , whereas the program in question may clearly yield  $a = 1$  if the load and the store of thread I are reordered. Although  $z \mapsto 0 * \Delta z \mapsto 0$  is unsatisfiable in the current model of assertions, it is quite possible to devise a different model of assertions, according to which the aforementioned assertion is satisfiable, and thus potentially restore the soundness of FSL under some weaker memory models.

## 9.2 Reasoning About SC Accesses and Fences

As the reader will have noticed, in this paper we have not presented any rules for reasoning about **sc** atomics. Naturally, since **sc** atomics are stronger than the release/acquire ones, the presented release/acquire rules are also sound for **sc** accesses and fences. The question is whether we can get any stronger proof rules for **sc** accesses and fences.

For **sc** fences, it seems quite likely that we can get better rules. An **sc** fence can be thought of as a combination of a **acq-rel** fence and an **acq-rel** RMW over a ghost location. Therefore, we should be able to extend the Hoare triples with a global invariant,  $J$ , which can be accessed at **sc** fences:

$$\frac{J * P * P' \Rightarrow J * Q * Q'}{J \vdash \{P * \nabla P'\} \text{fence}_{\text{sc}} \{Q * \Delta Q'\}} \quad (\text{F-SC})$$

Such an invariant may also be used for providing rules for **sc** accesses. The fragment of RC11 restricted to accesses only of **na** or **sc** kind corresponds exactly to the language targeted by CSL [19] (by treating **sc** accesses as being surrounded by atomic blocks). Thus, for this fragment at least, one can easily derive sound rules for **sc** accesses from the CSL rules involving resource invariants. The open question is whether one can extend the soundness of such rules to the full RC11 model, especially in cases where the same location may accessed both using **sc** and non-**sc** accesses.

## 9.3 Tool Support

The soundness proofs of the aforementioned adaptations of separation logic have all been mechanised in the Coq proof assistant (see RSL [28], FSL [5], GPS [27], FSL++ [6]) together with some example proofs. Nevertheless, doing proofs in those program logics in Coq without any additional infrastructure is quite cumbersome. What is very much needed is some support for more automated proofs.

Such support already exists for various flavours of (concurrent) separation logic. There exist a wide range of tools, from fully automated ones for suitable fragments of the logic to tactic libraries for assisting the manual derivation of mechanised proofs (e.g., [2, 8, 9, 16–18]). For the work described here, the two most relevant tools are probably Viper [17] and the Iris framework [9].

Viper [17] is a generic program verifier for resource-based logics. Recently, Summers and Müller [24] have encoded versions of the RSL/FSL proof rules into

Viper and have used them to verify among other examples a slightly simplified version of the ARC library verified in [6]. While their encoding is not expressive enough to verify the actual ARC implementation, it is much more convenient to use for the programs falling in its domain than the FSL’s Coq formalisation.

Iris [9] is a generic logical framework built around a higher-order variant of separation logic. It is deeply embedded in Coq and comes with a useful set of Coq tactics for doing proofs in that framework. In recent work, Kaiser et al. [10] have encoded a slight variant of GPS into Iris (thereby reproving its soundness within Iris) and have used Iris’s infrastructure to get a convenient way of constructing GPS proofs in Coq.

**Acknowledgments.** The work reported here was done in collaboration with a number of people—Hoang-Hai Dang, Marko Doko, Derek Dreyer, João Ferreira, Mengda He, Jan-Oliver Kaiser, Ori Lahav, Chinmay Narayan, Shengchao Qin, Aaron Turon—who are coauthors of the relevant publications. I would also like to thank the CAV’17 chairs for inviting me to write this paper.

## References

1. Adve, S.V., Boehm, H.: Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM* **53**(8), 90–101 (2010)
2. Berdine, J., Calcagno, C., O’Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) *FMCO 2005*. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006). doi:[10.1007/11804192\\_6](https://doi.org/10.1007/11804192_6)
3. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) *ESOP 2013*. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-37036-6\\_29](https://doi.org/10.1007/978-3-642-37036-6_29)
4. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D’Hondt, T. (ed.) *ECOOP 2010*. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-14107-2\\_24](https://doi.org/10.1007/978-3-642-14107-2_24)
5. Doko, M., Vafeiadis, V.: A program logic for C11 memory fences. In: Jobstmann, B., Leino, K.R.M. (eds.) *VMCAI 2016*. LNCS, vol. 9583, pp. 413–430. Springer, Heidelberg (2016). doi:[10.1007/978-3-662-49122-5\\_20](https://doi.org/10.1007/978-3-662-49122-5_20)
6. Doko, M., Vafeiadis, V.: Tackling real-life relaxed concurrency with FSL++. In: Yang, H. (ed.) *ESOP 2017*. LNCS, vol. 10201, pp. 448–475. Springer, Heidelberg (2017). doi:[10.1007/978-3-662-54434-1\\_17](https://doi.org/10.1007/978-3-662-54434-1_17)
7. He, M., Vafeiadis, V., Qin, S., Ferreira, J.F.: Reasoning about fences and relaxed atomics. In: *PDP 2016*, pp. 520–527. IEEE Computer Society (2016)
8. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) *NFM 2011*. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-20398-5\\_4](https://doi.org/10.1007/978-3-642-20398-5_4)
9. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: Rajamani, S.K., Walker, D. (eds.) *POPL 2015*, pp. 637–650. ACM, New York (2015)



10. Kaiser, J.O., Dang, H.H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: reasoning about release-acquire consistency in Iris. In: ECOOP 2017 (2017)
11. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Castagna, G., Gordon, A.D. (eds.) POPL 2017, pp. 175–189. ACM, New York (2017)
12. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: POPL 2016, pp. 649–662. ACM (2016)
13. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 311–323. Springer, Heidelberg (2015). doi:[10.1007/978-3-662-47666-6\\_25](https://doi.org/10.1007/978-3-662-47666-6_25)
14. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI 2017. ACM (2017)
15. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979)
16. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-17511-4\\_20](https://doi.org/10.1007/978-3-642-17511-4_20)
17. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). doi:[10.1007/978-3-662-49122-5\\_2](https://doi.org/10.1007/978-3-662-49122-5_2)
18. Nanevski, A., Morrisett, J.G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. **18**(5–6), 865–911 (2008)
19. O’Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. **375**(1–3), 271–307 (2007)
20. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: D’Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-14107-2\\_23](https://doi.org/10.1007/978-3-642-14107-2_23)
21. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inform. **6**(4), 319–340 (1976)
22. Owicki, S.S.: Axiomatic proof techniques for parallel programs. Ph.D. thesis, Cornell University (1975)
23. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: a logic for time and data abstraction. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 207–231. Springer, Heidelberg (2014). doi:[10.1007/978-3-662-44202-9\\_9](https://doi.org/10.1007/978-3-662-44202-9_9)
24. Summers, A.J., Müller, P.: Automating deductive verification for weak-memory programs (2017)
25. Tassarotti, J., Dreyer, D., Vafeiadis, V.: Verifying read-copy-update in a logic for weak memory. In: PLDI 2015, pp. 110–120. ACM (2015)
26. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In: Morrisett, G., Uustalu, T. (eds.) ICFP 2013, pp. 377–390. ACM (2013)
27. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: navigating weak memory with ghosts, protocols, and separation. In: OOPSLA 2014, pp. 691–707. ACM (2014)
28. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: OOPSLA 2013, pp. 867–884. ACM (2013)

Computer Aided Verification

29th International Conference, CAV 2017, Heidelberg,  
Germany, July 24-28, 2017, Proceedings, Part I

Majumdar, R.; Kunčák, V. (Eds.)

2017, XIX, 601 p. 142 illus., Softcover

ISBN: 978-3-319-63386-2