Automating Deductive Verification for Weak-Memory Programs

Alexander J. Summers and Peter Müller

Department of Computer Science, ETH Zurich, Switzerland
{alexander.summers, peter.mueller}@inf.ethz.ch

Abstract. Writing correct programs for weak memory models such as the C11 memory model is challenging because of the weak consistency guarantees these models provide. The first program logics for the verification of such programs have recently been proposed, but their usage has been limited thus far to manual proofs. Automating proofs in these logics via first-order solvers is non-trivial, due to features such as higher-order assertions, modalities and rich permission resources.
In this paper, we provide the first encoding of a weak memory program logic using existing deductive verification tools. Our work enables, for the first time, the (unbounded) verification of C11 programs at the level of abstraction provided by the program logics; the only necessary user interaction is in the form of specifications written in the program logic.
We tackle three recent program logics: Relaxed Separation Logic and two forms of Fenced Separation Logic, and show how these can be encoded using the Viper verification infrastructure. In doing so, we illustrate several novel encoding techniques which could be employed for other logics. Our work is implemented, and has been evaluated on examples from existing papers as well as the Facebook open-source Folly library.

1 Introduction

Reasoning about programs running on weak memory is challenging because weak memory models admit executions that are not sequentially consistent, that is, cannot be explained by a sequential interleaving of concurrent threads. Moreover, weak-memory programs employ a range of operations to access memory, which require dedicated reasoning techniques. These operations include fences as well as read and write accesses with varying degrees of synchronisation.

Some of these challenges are addressed by the first program logics for weak-memory programs, in particular, Relaxed Separation Logic (RSL) [37], GPS [35], Fenced Separation Logic (FSL) [17], and FSL++ [18]. These logics apply to interesting classes of C11 programs, but their tool support has been limited to embeddings in Coq. Verification based on these embeddings requires substantial user interaction, which is an obstacle to applying and evaluating these logics.

In this paper, we present a novel approach to automating deductive verification for weak memory programs. We encode large fractions of RSL, FSL, and FSL++ (collectively referred to as *the RSL logics*) into the intermediate

$$s ::= l := \texttt{alloc}_{\texttt{na}}() \mid l := \texttt{alloc}_{\rho}(\mathcal{Q}) \mid [l]_{\sigma} := e \mid x := [l]_{\sigma}$$
$$\mid \texttt{fence}_{\texttt{acq}} \mid \texttt{fence}_{\texttt{rel}}(A) \mid x := \texttt{CAS}_{\tau}(l, e_1, e_2)$$
$$\textit{where } \rho \in \{\texttt{acq}, \texttt{RMW}\}, \quad \sigma ::= \texttt{na} \mid \tau, \quad \tau \in \{\texttt{acq}, \texttt{rel}, \texttt{rel\_acq}, \texttt{rlx}\}$$

Fig. 1. Syntax for memory accesses. na indicates a non-atomic operation; τ indicates an atomic access mode (as defined in C11), discussed in later sections. ρ, and assertions A and invariants $\mathcal{Q}$ are program annotations, needed as input for our encoding. Expressions e include boolean and arithmetic operations, but no heap accesses. We assume that source programs are type-checked.

verification language Viper [27], and use the existing Viper verification backends to reason automatically about the encoded programs. This encoding reduces all concurrency and weak-memory features as well as logical features such as higher-order assertions and custom modalities to a much simpler sequential logic.

Defining an encoding into Viper is much more lightweight than developing a dedicated verifier from scratch, since we can reuse the existing automation for a variety of advanced program reasoning features. Compared to an embedding into an interactive theorem prover such as Coq, our approach leads to a significantly higher degree of automation than that typically achieved through tactics. Moreover, it allows users to interact with the verifier on the abstraction level of source code and annotations, without exposing the underlying formalism. Verification in Coq can provide foundational guarantees, whereas in our approach, errors in the encoding or bugs in the verifier could potentially invalidate verification results. We mitigate the former risk by a soundness argument for our encoding and the latter by the use of a mature verification system. We are convinced that both approaches are necessary: foundational verification is ideal for meta-theory development and application areas such as safety-critical systems, whereas our approach is well-suited for prototyping and evaluating logics, and for making a verification technique applicable by a wider user base.

The contributions of this paper are: (1) The first automated deductive verification approach for weak-memory logics. We demonstrate the effectiveness of this approach on examples from the literature, which are available online [3]. (2) An encoding of large fractions of RSL, FSL, and FSL++ into Viper. Various aspects of this encoding (such as the treatment of higher-order features and modalities, as well as the overall proof search strategy) are generic and can be reused to encode other advanced separation logics. (3) A prototype implementation, which is available online [4].

Related Work. The existing weak-memory logics RSL [37], GPS [35], FSL [17], and FSL++ [18] have been formalized in Coq and used to verify small examples. The proofs were constructed mostly manually, whereas our approach automates most of the proof steps. As shown in our evaluation, our approach reduces the overhead by more than an order of magnitude. The degree of automation in Coq could be increased through logic-specific tactics (e.g. [32, 13]), whereas our approach benefits from Viper's automation for the intermediate language, which is independent of the encoded logic.

$$A ::= e \mid l \overset{k}{\mapsto} e \mid A_1 * A_2 \mid e \Rightarrow A \mid (e\ ?\ A_1 : A_2)$$
$$\mid \mathsf{Uninit}(l) \mid \mathsf{Acq}(l, \mathcal{Q}) \mid \mathsf{Rel}(l, \mathcal{Q}) \mid \mathsf{Init}(l) \mid \triangle A \mid \triangledown A \mid \mathsf{RMWAcq}(l, \mathcal{Q})$$

Fig. 2. Assertion syntax of the RSL logics. The top row of constructs are standard for separation logics; those in the second row are specific to the RSL logics, and explained throughout the paper. Invariants $\mathcal{Q}$ are *functions* from values to assertions (cf. Sec. 3).

Jacobs [20] proposed a program logic for the TSO memory model that has been encoded in VeriFast [21]. This encoding requires a substantial amount of annotations, whereas our approach provides a higher degree of automation and handles the more complex C11 memory model.

Weak-memory reasoning has been addressed using techniques based on model-checking (e.g. [11, 6, 5]) and static analyses (e.g. [16, 7]). These approaches are fully automatic, but do not analyse code modularly, which is e.g. important for verifying libraries independently from their clients. Deductive verification enables compositional proofs by requiring specifications at function boundaries. Such specifications can express precise information about the (unbounded) behaviour of a program's constituent parts.

Automating logics via encodings into intermediate verification languages is a proven approach, as witnessed by the many existing verifiers (e.g. [14, 15, 24, 25]) which target Boogie [8] or Why3 [9]. Our work is the first that applies this approach to logics for weak-memory concurrency. Our encoding benefits from Viper's native support for separation-logic-style reasoning and several advanced features such as quantified permissions and permission introspection [27, 26], which are not available in other intermediate verification languages.

Outline. The next four sections present our encoding for the core features of the C11 memory model: we discuss non-atomic locations in Sec. 2, release-acquire accesses in Sec. 3, fences in Sec. 4, and compare-and-swap in Sec. 5. We discuss soundness and completeness of our encoding in Sec. 6 and evaluate our approach in Sec. 7. Sec. 8 concludes. Further details of our encoding and examples are available in our accompanying technical report (hereafter, TR) [34]. A prototype implementation of our encoding (with all examples) is available as an artifact [4].

2 Non-atomic Locations

We present our encoding for a small imperative programming language similar to the languages supported by the RSL logics. C11 supports *non-atomic* memory accesses and different forms of *atomic* accesses. The access operations are summarised in Fig. 1. We adopt the common simplifying assumption [37, 35] that memory locations are partitioned into those accessed only via non-atomic accesses (*non-atomic locations*), and those accessed only via C11 atomics (*atomic locations*). Read and write statements are parameterised by a mode σ, which is either `na` (non-atomic) or one of the atomic access modes τ. We focus on non-atomic accesses in this section and discuss atomics in subsequent sections.

$$\frac{}{\vdash \{\mathsf{true}\}\ l := \mathtt{alloc}_{\mathsf{na}}()\ \{\mathsf{Uninit}(l)\}} \qquad \frac{}{\vdash \{l \overset{1}{\mapsto} \_ \lor \mathsf{Uninit}(l)\}\ [l]_{\mathsf{na}} := e\ \{l \overset{1}{\mapsto} e\}}$$

$$\frac{}{\vdash \{l \overset{k}{\mapsto} e\}\ x := [l]_{\mathsf{na}}\ \{x = e * l \overset{k}{\mapsto} e\}} \qquad (l \overset{k}{\mapsto} e\ *\ l \overset{k'}{\mapsto} e') \Leftrightarrow (e = e' * l \overset{k+k'}{\mapsto} e)$$

Fig. 3. Adapted RSL rules for non-atomics. Read access requires a non-zero permission. Write access requires either write permission or that the location is uninitialised. The underscore _ stands for an arbitrary value.

RSL proof rules. Non-atomic memory accesses come with no synchronisation guarantees; programmers need to ensure that all accesses to non-atomic locations are data-race free. The RSL logics enforce this requirement using standard separation logic [28, 31]. We show the syntax of assertions in Fig. 2, which will be explained throughout the paper. A *points-to assertion* $l \overset{k}{\mapsto} e$ denotes a transferrable *resource*, providing permission to access the location l, and expressing that l has been initialised and its current value is e. Here, k is a fraction $0 < k \leq 1$; $k = 1$ denotes the *full* (or exclusive) permission to read and write location l, whereas $0 < k < 1$ provides (non-exclusive) read access [12]. Points-to resources can be split and recombined, but never duplicated or forged; when transferring such a resource to another thread it is removed from the current one, avoiding data races by construction. The RSL assertion $\mathsf{Uninit}(l)$ expresses exclusive access to a location l that has been allocated, but not yet initialised; l may be written to but not read from. The main proof rules for non-atomic locations, adapted from RSL [37], are shown in Fig. 3.

Encoding. The Viper intermediate verification language [27] supports an assertion language based on Implicit Dynamic Frames [33], a program logic related to separation logic [29], but which separates permissions from value information. Viper is object-based; the only memory locations are field locations $e.f$ (in which e is a reference, and f a field name). Permissions to access these heap locations are described by *accessibility predicates* of the form `acc(`$e.f$`, `k`)`, where k is a fraction as for points-to predicates above (k defaults to 1). Assertions that do not contain accessibility predicates are called *pure*. Unlike in separation logics, heap locations may be read in pure assertions.

We model C-like memory locations l using a field `val` of a Viper reference l. Consequently, a separation logic assertion $l \overset{k}{\mapsto} e$ is represented in Viper as `acc(`l`.val, `k`) && `l`.val == e`. We assume that memory locations have type `int`, but a generalisation is trivial. Viper's conjunction `&&` treats permissions like a separating conjunction, requiring the sum of the permissions in each conjunct, and acts as logical conjunction for pure assertions (just as $*$ in separation logic).

Viper provides two key statements for encoding proof rules: **`inhale`** A *adds* the permissions denoted by the assertion A to the current state, and *assumes* pure assertions in A. This can be used to model gaining new resources, e.g., acquiring a lock in the source program. Dually, **`exhale`** A checks that the current state satisfies A (otherwise a verification error occurs), and *removes* the permissions

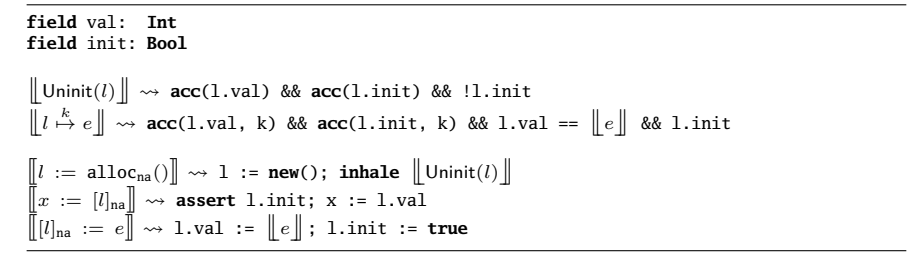

Fig. 4. Viper encoding of the RSL assertions and the rules for non-atomic memory accesses from Fig. 3.

that A denotes; the values of any locations to which no permission remains are *havoced* (assigned arbitrary values). For example, when forking a new thread, its precondition is exhaled to transfer the necessary resources from the forking thread. Inhale and exhale statements can be seen as the permission-aware analogues of the assume and assert statements of first-order verification languages [25].

The encoding of the rules for non-atomics from Fig. 3 is presented in Fig. 4. $\llbracket A \rrbracket \rightsquigarrow \ldots$ denotes the encoding of an RSL assertion A as a Viper assertion, and analogously $[\![s]\!] \rightsquigarrow \ldots$ for source-level statements s.

The first two lines show background declarations. The assertion encodings follow the explanations above. Allocation is modelled by obtaining a fresh reference (via `new()`) and inhaling permissions to its `val` and `init` fields; assuming `!l.init` reflects that the location is not yet initialised. Viper implicitly checks the necessary permissions for field accesses (verification fails otherwise). Hence, the translation of a non-atomic read only needs to check that the read location is initialised before obtaining its value. Analogously, the translation of a non-atomic write only stores the value and records that the location is now initialised.

Note that Viper's implicit permission checks are both necessary and sufficient to encode the RSL rules in Fig. 3. In particular, the assertions $l \overset{1}{\mapsto} \_$ and $\mathsf{Uninit}(l)$ both provide the permissions to write to location l. By including `acc(`l`.val)` in the encoding of both assertions, we avoid the disjunction of the RSL rule.

Like the RSL logics, our approach requires programmers to annotate their code with access modes for locations (as part of the `alloc` statement), and specifications such as pre and postconditions for methods and threads. Given these inputs, Viper constructs the proof automatically. In particular, it automatically proves entailments, and splits and combines fractional permissions (hence, the equivalence in Fig. 3 need not be encoded). Automation can be increased further by inferring some of the required assertions, but this is orthogonal to the encoding presented in this paper.

3 Release-Acquire Atomics

The simplest form of C11 atomic memory accesses are *release write* and *acquire read* operations. They can be used to synchronise the transfer of ownership of

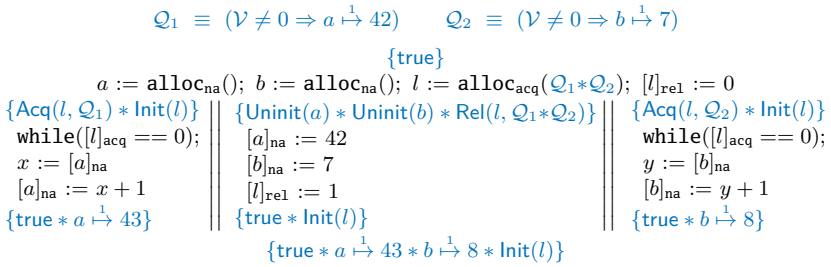

Fig. 5. An example illustrating "message passing" of non-atomic ownership, using release acquire atomics (inspired by an example from [17]). Annotations are shown in blue. This example corresponds to `RelAcqDblMsgPassSplit` in our evaluation (Sec. 7).

(and information about) other, non-atomic locations, using a *message passing idiom*, illustrated by the example in Fig. 5. This program allocates two non-atomic locations a and b, and an atomic location l (initialised to 0), which is used to synchronise the three threads that are spawned afterwards. The middle thread makes changes to the non-atomics a and b, and then signals completion via a release write of 1 to l; conceptually, it gives up ownership of the non-atomic locations via this signal. The other threads loop attempting to acquire-read a non-zero value from l. Once they do, they each gain ownership of one non-atomic location via the acquire read of 1 and access that location. The release write and acquire reads of value 1 enforce *ordering constraints* on the non-atomic accesses, preventing the left and right threads from racing with the middle one.

RSL proof rules. The RSL logics capture message-passing idioms by associating a *location invariant* $\mathcal{Q}$ with each atomic location. Such an invariant is a function from values to assertions; we represent such functions as assertions with a distinguished variable symbol $\mathcal{V}$ as parameter. Location invariants prescribe the intended ownership that a thread obtains when performing an acquire read of value $\mathcal{V}$ from the location, and that must correspondingly be given up by a thread performing a release write. The main proof rules [37] are shown in Fig. 6.

When allocating an atomic location for release/acquire accesses (first proof rule), a location invariant $\mathcal{Q}$ must be chosen (as an annotation on the allocation). The assertions $\mathsf{Rel}(l, \mathcal{Q})$ and $\mathsf{Acq}(l, \mathcal{Q})$ record the invariant to be used with subsequent release writes and acquire reads. To perform a release write of value e (second rule), a thread must hold the $\mathsf{Rel}(l, \mathcal{Q})$ assertion and *give up* the assertion $\mathcal{Q}[e/\mathcal{V}]$. For example, the line $[l]_{\texttt{rel}} := 1$ in Fig. 5 causes the middle thread to give up ownership of both non-atomic locations a and b. The assertion $\mathsf{Init}(l)$ represents that atomic location l is initialised; both $\mathsf{Init}(l)$ and $\mathsf{Rel}(l, \mathcal{Q})$ are *duplicable* assertions; once obtained, they can be passed to multiple threads.

Multiple acquire reads might read the value written by a single release write operation; RSL prevents ownership of the transferred resources from being obtained (unsoundly) by multiple readers in two ways. First, $\mathsf{Acq}(l, \mathcal{Q})$ assertions cannot be duplicated, only split by partitioning the invariant $\mathcal{Q}$ into disjoint parts.

$$\frac{}{\vdash \{\mathsf{true}\}\ l := \mathtt{alloc}_{\mathtt{acq}}(\mathcal{Q})\ \{\mathsf{Rel}(l,\mathcal{Q}) * \mathsf{Acq}(l,\mathcal{Q})\}}$$

$$\frac{}{\vdash \{\mathcal{Q}(e) * \mathsf{Rel}(l,\mathcal{Q})\}\ [l]_{\mathtt{rel}} := e\ \{\mathsf{Init}(l) * \mathsf{Rel}(l,\mathcal{Q})\}}$$

$$\frac{}{\vdash \{\mathsf{Init}(l) * \mathsf{Acq}(l,\mathcal{Q})\}\ x := [l]_{\mathtt{acq}}\ \{\mathcal{Q}[x/\mathcal{V}] * \mathsf{Acq}(l, (\mathcal{V} \neq x \Rightarrow \mathcal{Q}))\}}$$

$$\mathsf{Init}(l) \ \Leftrightarrow\ \mathsf{Init}(l) * \mathsf{Init}(l) \qquad \mathsf{Rel}(l,\mathcal{Q}) \ \Leftrightarrow\ \mathsf{Rel}(l,\mathcal{Q}) * \mathsf{Rel}(l,\mathcal{Q})$$

$$\mathsf{Acq}(l, \mathcal{Q}_1 * \mathcal{Q}_2) \ \Leftrightarrow\ \mathsf{Acq}(l,\mathcal{Q}_1) * \mathsf{Acq}(l,\mathcal{Q}_2) \qquad \mathcal{Q}_1 \models \mathcal{Q}_2 \ \Rightarrow\ \mathsf{Acq}(l,\mathcal{Q}_1) \models \mathsf{Acq}(l,\mathcal{Q}_2)$$

Fig. 6. Adapted RSL rules for release-acquire atomics.

For example, in Fig. 5, $\mathsf{Acq}(l, \mathcal{Q}_1)$ is given to the left thread, and $\mathsf{Acq}(l, \mathcal{Q}_2)$ to the right. Second, the rule for acquire reads adjusts the invariant in the Acq assertion such that subsequent reads of the same value will not obtain any ownership.

Encoding. A key challenge for encoding the above proof rules is that Rel and Acq are parameterised by the invariant $\mathcal{Q}$; higher-order assertions are not directly supported in Viper. However, for a given program, only finitely many such parameterisations will be required, which allows us to apply defunctionalisation [30], as follows. Given an annotated program, we assign a unique *index* to each syntactically-occurring invariant $\mathcal{Q}$ (in particular, in allocation statements, and as parameters to Rel and Acq assertions in specifications). Furthermore, we assign unique indices to all *immediate conjuncts* of these invariants. We write *indices* for the set of indices used. For each i in *indices*, we write $inv(i)$ for the invariant which i indexes. For an invariant $\mathcal{Q}$, we write $\langle\mathcal{Q}\rangle$ for its index, and $\langle\!\langle\mathcal{Q}\rangle\!\rangle$ for the set of indices assigned to its immediate conjuncts.

Our encoding of the RSL rules from Fig. 6 is summarised in Fig. 7. To encode duplicable assertions such as $\mathsf{Init}(l)$, we make use of Viper's *wildcard permissions* [27], which represent unknown positive permission amounts. When exhaled, these amounts are chosen such that the amount exhaled will be *strictly smaller* than the amount held (verification fails if no permission is held) [19]. So after inhaling an $\mathsf{Init}(l)$ assertion (that is, a **`wildcard`** permission), it is possible to exhale two **`wildcard`** permissions, corresponding to two $\mathsf{Init}(l)$ assertions. Note that for atomic locations, we only use the `init` field's permissions, not its value.

We represent a $\mathsf{Rel}(l, \_)$ assertion for *some* invariant via a **`wildcard`** permission to a `rel` field; this is represented via the `SomeRel(l)` macro[1], and is used as the precondition for a release write (we must hold *some* Rel assertion, according to Fig. 6). The specific invariant associated with the location l is represented by storing its index as the *value* of the `rel` field; when encoding a release write, we branch on this value to exhale the appropriate assertion.

Analogously to Rel, we represent an Acq assertion for *some* invariant using a **`wildcard`** permission (the `SomeAcq` macro), which is the precondition for executing an acquire read. However, to support splitting, we represent the invariant in a

[1] Viper macros can be defined for assertions or statements, and are syntactically expanded (and their arguments substituted) on use.

```
field rel: Int
field acq: Bool
predicate AcqConjunct(l: Ref, idx: Int)

function valsRead(l: Ref, i: Int): Set[Int]
  requires AcqConjunct(l, i)

define SomeRel(l)  acc(l.rel, wildcard)
define SomeAcq(l)  acc(l.acq, wildcard) && l.acq == true

⟦Init(l)⟧ ⇝ acc(l.init, wildcard)
⟦Rel(l, Q)⟧ ⇝ SomeRel(l) && l.rel == ⟨Q⟩
⟦Acq(l, Q)⟧ ⇝ SomeAcq(l) && (foreach i in ⟪Q⟫:
  AcqConjunct(l, i) && valsRead(l, i) == Set[Int]() end)

⟦l := alloc_acq(Q)⟧ ⇝ l := new(); inhale ⟦Rel(l, Q)⟧ && ⟦Acq(l, Q)⟧

⟦[l]_rel := e⟧ ⇝ assert SomeRel(l);
  foreach i in indices do
    if (i == l.rel) { exhale inv(i)[e/V] }
  end
  inhale Init(l)

⟦x := [l]_acq⟧ ⇝ assert Init(l) && SomeAcq(l); x := havoc(); // unknown Int
  foreach i in indices do
    if (perm(AcqConjunct(l, i)) == 1 && !(x in valsRead(l, i))) {
      inhale inv(i)[x/V]
      tmpSet := valsRead(l, i)
      exhale AcqConjunct(l, i)
      inhale AcqConjunct(l, i) && valsRead(l,i) == tmpSet union Set(x)
    }
  end
```

Fig. 7. Viper encoding of the RSL rules for release-acquire atomics from Fig. 6. The operations in italics (e.g. *foreach*) are expanded statically in our encoding into conjunctions or statement sequences. The value of the `acq` field will be explained in Sec. 5.

more fine-grained way, by recording individual conjuncts separately. Each conjunct i of the invariant is modelled as an abstract *predicate* instance `AcqConjunct(`l`, `i`)`, which can be inhaled and exhaled individually. This encoding handles the common case that invariants are split along top-level conjuncts, as in Fig. 5. More complex splits can be supported through additional annotations: see App. C of the TR [34].

A release write is encoded by checking that some Rel assertion is held, and then exhaling the associated invariant for the value written. Moreover, it records that the location is initialised. The RSL rule for acquire reads adjusts the Acq invariant by obliterating the assertion for the value read. Instead of directly representing the adjusted invariant (which would complicate our numbering scheme), we track the set of values read as state in our encoding. We complement each `AcqConjunct` predicate instance with an (uninterpreted) Viper function `valsRead(`l`, `i`)`, returning a set of indices[2].

An acquire read checks that the location is initialised and that we have *some* Acq assertion for the location. It assigns an unknown value to the lhs variable

[2] Viper's heap-dependent functions are mathematical functions of their parameters and the resources stated in their preconditions (here, `AcqConjunct(`l`,`i`)`).

$$\frac{}{\{A\}\ \texttt{fence}_{\texttt{rel}}\ \{\triangle A\}} \qquad \frac{}{\{\triangledown A\}\ \texttt{fence}_{\texttt{acq}}\ \{A\}}$$

$$\frac{}{\{\triangle\mathcal{Q}(e) * \mathsf{Rel}(l,\mathcal{Q})\}\ [l]_{\texttt{rlx}} := e\ \{\mathsf{Init}(l) * \mathsf{Rel}(l,\mathcal{Q})\}}$$

$$\frac{}{\{\mathsf{Init}(l) * \mathsf{Acq}(l,\mathcal{Q})\}\ x := [l]_{\texttt{rlx}}\ \{\triangledown\mathcal{Q}[x/\mathcal{V}] * \mathsf{Acq}(l,\mathcal{V} \neq x \Rightarrow \mathcal{Q})\}}$$

$$(A_1 \Rightarrow A_2) \Leftrightarrow (\triangle A_1 \Rightarrow \triangle A_2) \Leftrightarrow (\triangledown A_1 \Rightarrow \triangledown A_2)$$

$$\triangledown(A_1 * A_2) \equiv (\triangledown A_1) * (\triangledown A_2) \textit{ and analogously for } \triangle \textit{ and other binary connectives}$$

Fig. 8. Adapted FSL rules for relaxed atomics and fences.

x, which is subsequently constrained by the invariant associated with the Acq assertion as follows: We check for each index whether we both currently hold an `AcqConjunct` predicate for that index[3], and if so, have not previously read the value x from that conjunct of our invariant. If these checks succeed, we inhale the indexed invariant for x, and then include x in the values read.

The encoding presented so far allows us to automatically verify annotated C11 programs using release writes and acquire reads (e.g., the program of Fig. 5) without any custom proof strategies [3]. In particular, we can support the higher-order Acq and Rel assertions through defunctionalisation and enable the splitting of invariants through a suitable representation.

4 Relaxed Memory Accesses and Fences

In contrast to release-acquire accesses, C11's *relaxed* atomic accesses provide no synchronisation: threads may observe reorderings of relaxed accesses and other memory operations. Correspondingly, RSL's proof rules for relaxed atomics provide weak guarantees, and do not support ownership transfer. Memory fence instructions can eliminate this problem. Intuitively, a *release fence* together with a subsequent relaxed write allows a thread to transfer away ownership of resources, similarly to a release write. Dually, an *acquire fence* together with a prior relaxed read allows a thread to obtain ownership of resources, similarly to an acquire read. This reasoning is justified by the ordering guarantees of the C11 model [17].

FSL proof rules. FSL and FSL++ provide proof rules for fences (see Fig. 8). They use *modalities* $\triangle$ ("up") and $\triangledown$ ("down") to represent resources that are transferred through relaxed accesses and fences. An assertion $\triangle A$ represents a resource A which has been prepared, via a release fence, to be transferred by a relaxed write operation; dually, $\triangledown A$ represents resources A obtained via a relaxed read, which may not be made use of until an acquire fence is encountered. The proof rule for relaxed write is identical to that for a release write (cf. Fig. 6), except that the assertion to be transferred away must be under the $\triangle$ modality;

[3] A **`perm`** expression yields the permission fraction held for a field or predicate instance.

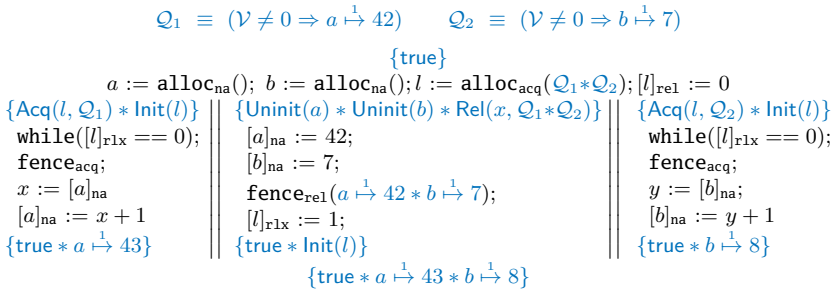

Fig. 9. A variant of the message-passing example of Fig. 5, combining relaxed memory accesses and fences to achieve ownership transfer. The example is also a variant of Fig. 2 of the FSL paper [17], which is included in our evaluation (`FencesDblMsgPass`) in Sec. 7.

this can be achieved by the rule for release fences. The rule for a relaxed read is the same as that for acquire reads, except that the gained assertion is under the $\bigtriangledown$ modality. The modality can be removed by a subsequent acquire fence. Finally, assertions may be rewritten under modalities, and both modalities distribute over all other logical connectives.

Fig. 9 shows an example program, which is a variant of the message-passing example from Fig. 5. Comparing the left-hand one of the three parallel threads, a relaxed read is used in the spin loop; after the loop, this thread will hold the assertion $\bigtriangledown a \overset{1}{\mapsto} 42$. The subsequent $\texttt{fence}_{\texttt{acq}}$ statement allows the modality to be removed, allowing the non-atomic location a to be accessed. Dually, the middle thread employs a $\texttt{fence}_{\texttt{rel}}$ statement to place the ownership of the non-atomic locations under the $\bigtriangleup$ modality, in preparation for the relaxed write to l.

Encoding. The main challenge in encoding the FSL rules for fences is how to represent the two new modalities. Since these modalities guard assertions which cannot be currently used or combined with modality-free assertions, we model them using two *additional heaps* to represent the assertions under each modality. The program heap (along with associated permissions) is a built-in notion in Viper, and so we cannot directly employ three heaps. Therefore, we construct the additional "up" and "down" heaps, by axiomatising bijective mappings `up` and `down` between a real program reference and its counterparts in these heaps. That is, technically our encoding represents each source location through three references in Viper's heap (rather than one reference in three heaps). Assertions $\bigtriangleup A$ are then represented by replacing *all references* r in the encoded assertion A with their counterpart $\texttt{up}(r)$. We write $\lceil A \rceil^{up}$ for the transformation which performs this replacement. For example, $\lceil$`acc(x.val) && x.val == 4`$\rceil^{up} \rightsquigarrow$ `acc(up(x).val) && up(x).val == 4`. We write $\lceil A \rceil^{down}$ for the analogous transformation for the `down` function.

The extension of our encoding is shown in Fig. 10. We employ a Viper *domain* to introduce and axiomatise the mathematical functions for our `up` and `down` mappings. By axiomatising inverses for these mappings, we guarantee bijectivity.

```
domain threeHeaps {
  function up(x: Ref) : Ref;    function upInv(x: Ref) : Ref;
  function down(x: Ref) : Ref;  function downInv(x: Ref) : Ref;
  function heap(x: Ref) : Int;  // identifies which heap a Ref is from
  axiom { forall r:Ref :: upInv(up(r)) == r &&
    (heap(r) == 0 ==> heap(up(r)) == 1 }
  axiom { forall r:Ref :: up(upInv(r)) == r &&
    (heap(r) == 1 ==> heap(upInv(r)) == 0 }
  axiom { forall r:Ref :: downInv(down(r)) == r &&
    (heap(r) == 0 ==> heap(down(r)) == -1 }
  axiom { forall r:Ref :: down(downInv(r)) == r &&
    (heap(r) == -1 ==> heap(downInv(r)) == 0 }
}
```

$\llbracket \triangle A \rrbracket \rightsquigarrow \lceil \llbracket A \rrbracket \rceil^{up}$ $\qquad$ $\llbracket \triangledown A \rrbracket \rightsquigarrow \lceil \llbracket A \rrbracket \rceil^{down}$

$\llbracket [l]_{\mathtt{rlx}} := e \rrbracket \rightsquigarrow \ldots$ *encoded as for release writes (Fig. 7) except using* $\lceil inv(i) \rceil^{up}$ *in place of* $inv(i)$

$\llbracket x := [l]_{\mathtt{rlx}} \rrbracket \rightsquigarrow \ldots$ *encoded as for acquire reads (Fig. 7) except using* $\lceil inv(i) \rceil^{down}$ *in place of* $inv(i)$

$\llbracket \mathtt{fence}_{\mathtt{rel}}(A) \rrbracket \rightsquigarrow$ **exhale** $\llbracket A \rrbracket$; **inhale** $\lceil \llbracket A \rrbracket \rceil^{up}$

$\llbracket \mathtt{fence}_{\mathtt{acq}} \rrbracket \rightsquigarrow$

```
var rs : Set[Ref];  rs := havoc() // unknown set of Refs
assume forall r: Ref :: r in rs <==> perm(down(r).val) > none
inhale forall r: Ref :: r in rs ==> acc(r.val, perm(down(r).val))
assume forall r: Ref :: r in rs ==> r.val == down(r).val
exhale forall r: Ref :: r in rs ==> acc(down(r).val, perm(down(r).val))
// analogously for each other field, predicate (in place of val)
```

Fig. 10. Viper encoding of the FSL rules for relaxed atomics and memory fences from Fig. 8. We omit triggers for the quantifiers for simplicity, but see [3].

Bijectivity allows Viper to conclude that (dis)equalities and other information is preserved under these mappings. Consequently, we do not have to explicitly encode the last two rules of Fig. 8; they are reduced to standard assertion manipulations in our encoding. An additional `heap` function labels references with an integer identifying the heap to which they belong (`0` for real references, `-1` and `1` for their "down" and "up" counterparts); this labelling provides the verifiers with the (important) information that these notional heaps are disjoint.

Our handling of relaxed reads and writes is almost identical to that of acquire reads and release writes in Fig. 7; this similarity comes from the proof rules, which only require that the modalities be inserted for the invariant. Our encoding for release fences requires an annotation in the source program to indicate which assertion to prepare for release by placing it under the $\triangle$ modality.

Our encoding for acquire fences does *not* require any annotations. *Any* assertion under the $\triangledown$ modality can (and should) be converted to its corresponding version without the modality because $\triangledown A$ is strictly less-useful than A itself. To encode this conversion, we find *all* permissions currently held in the down heap, and transfer these permissions and the values of the corresponding locations over to the real heap. These steps are encoded for each field and predicate separately; Fig. 10 shows the steps for the `val` field. We first define a set `rs` to be precisely the set of all references `r` to which *some* permission to `down(r).val` is currently held, i.e., **`perm(down(r).val) > none`**. For each such reference, we **`inhale`** exactly

$$\frac{}{\{\mathsf{true}\}\ l := \mathtt{alloc}_{\mathtt{RMW}}(\mathcal{Q})\ \{\mathsf{Rel}(l,\mathcal{Q}) * \mathsf{RMWAcq}(l,\mathcal{Q})\}}$$

$$\frac{\begin{array}{ll} x \notin FV(P) & P' \equiv \begin{cases} P & \text{if } \tau \in \{\mathtt{rel}, \mathtt{rel\_acq}\} \\ \triangle P & \text{otherwise} \end{cases} \\ x \notin FV(e) & \\ \mathcal{Q}[e/\mathcal{V}] \models A * T & A' \equiv \begin{cases} A & \text{if } \tau \in \{\mathtt{acq}, \mathtt{rel\_acq}\} \\ \triangledown A & \text{otherwise} \end{cases} \\ P * T \models \mathcal{Q}[e'/\mathcal{V}] & \end{array}}{\left\{\begin{array}{l} \mathsf{Init}(l) * \mathsf{Rel}(l,\mathcal{Q}) * \\ \mathsf{RMWAcq}(l,\mathcal{Q}) * P' \end{array}\right\}\ x := \mathtt{CAS}_\tau(l,e,e')\ \left\{\begin{array}{l} (x = e\ ?\ A' : P') * \mathsf{Init}(l) * \\ \mathsf{Rel}(l,\mathcal{Q}) * \mathsf{RMWAcq}(l,\mathcal{Q}) \end{array}\right\}}$$

$$\mathsf{RMWAcq}(l,\mathcal{Q}) \quad \Leftrightarrow \quad \mathsf{RMWAcq}(l,\mathcal{Q}) * \mathsf{RMWAcq}(l,\mathcal{Q})$$

Fig. 11. Adapted FSL++ rules for compare and swap operations. FV yields the free variables of an assertion.

the same amount of permission to the corresponding `r.val` location, equate the heap values, and then remove the permission to the down locations.

With our encoding based on multiple heaps, reasoning about assertions under modalities inherits all of Viper's native automation for permission and heap reasoning. We will reuse this idea for a different purpose in the following section.

5 Compare and Swap

C11 includes atomic *read-modify-write* operations, commonly used to implement high-level synchronisation primitives such as locks. FSL++ [18] provides proof rules for *compare-and-swap* (CAS) operations. An atomic compare and swap $\mathtt{CAS}_\tau(l, e, e')$ reads and returns the value of location l; if the value read is equal to e, it also writes the value e' (otherwise we say that the CAS *fails*).

FSL++ proof rules. FSL++ provides an assertion $\mathsf{RMWAcq}(l, \mathcal{Q})$, which is similar to $\mathsf{Acq}(l, \mathcal{Q})$, but is used for CAS operations instead of acquire reads. A successful CAS *both* obtains ownership of an assertion via its read operation and gives up ownership of an assertion via its write operation.

FSL++ does not support general combinations of atomic reads and CAS operations on the same location; the way of reading must be chosen at allocation via the annotation ρ on the allocation statement (see Fig. 1). In contrast to the Acq assertions used for atomic reads, RMWAcq assertions can be freely duplicated and their invariants need not be adjusted for a successful CAS: when using only CAS operations, each value read from a location corresponds to a different write.

Our presentation of the relevant proof rules is shown in Fig. 11. Allocating a location with annotation `RMW` provides a Rel and a RMWAcq assertion, such that the location can be used for release writes and CAS operations.

For the CAS operation, we present a single, general proof rule instead of four rules for the different combinations of access modes in FSL++. The rule requires that l is initialised (since its value is read), Rel and RMWAcq assertions, and an assertion P' that provides the resources needed for a successful CAS. If the CAS fails (that is, $x \neq e$), its precondition is preserved.

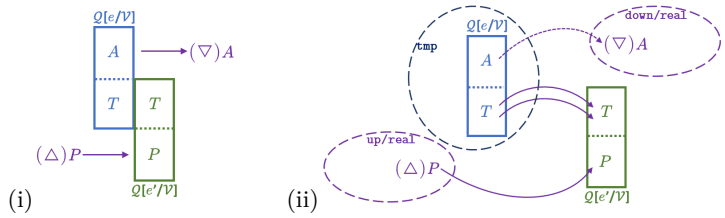

Fig. 12. An illustration of (i) the proof rule for CAS operations and (ii) our Viper encoding; the dashed regions denote the relevant heaps employed in the encoding.

If the CAS succeeds, it has read value e and written value e'. Assuming for now that the access mode τ permits ownership transfer, the thread has acquired $\mathcal{Q}[e/\mathcal{V}]$ and released $\mathcal{Q}[e'/\mathcal{V}]$. As illustrated in Fig. 12(i), these assertions may overlap. Let T denote the assertion characterising the overlap; then assertion A denotes $\mathcal{Q}[e/\mathcal{V}]$ without the overlap, and P denotes $\mathcal{Q}[e'/\mathcal{V}]$ without the overlap. The net effect of a successful CAS is then to acquire A and to release P, while T remains with the location invariant across the CAS. Automating the choice of T, A, and P is one of the main challenges of encoding this rule. Finally, if the access mode τ does not permit ownership transfer (that is, fences are needed to perform the transfer), A and P are put under the appropriate modalities.

Encoding. Our encoding of CAS operations uses several techniques presented in earlier sections: see App. E of the TR [34] for details. We represent RMWAcq assertions analogously to our encoding of Acq assertions (see Sec. 3). We use the value of field `acq` (cf. Fig. 7) to distinguish holding some RMWAcq assertion from some Acq assertion. Since RMWAcq assertions are duplicable (cf. Fig. 11), we employ **`wildcard`** permissions for the corresponding `AcqConjunct` predicates.

Our encoding of the proof rule for CAS operations is somewhat involved; we give a high-level description here, and relegate the details to App. E of the TR. We focus on the more-interesting case of a successful CAS here. The key challenge is how to select assertion T to satisfy the premises of the rule. Maximising this overlap is desirable in practice since this reduces the resources to be transferred, and which must interact in some cases with the modalities. Our Viper encoding indirectly *computes* this largest-possible T as follows (see Fig. 12(ii) for an illustration).

We introduce yet another heap ("tmp") in which we inhale the invariant $\mathcal{Q}[e/\mathcal{V}]$ for the value read. Now, we exhale the invariant $\mathcal{Q}[e'/\mathcal{V}]$ for the value written, but adapt the assertions as follows: for each permission in the invariant, we take *the maximum possible* amount from our "tmp" heap; these permissions correspond to T. Any remainder is taken from the current heap (either the real or the "up" heap, depending on τ); these correspond to P. Any permissions remaining in the "tmp" heap after this exhale correspond to the assertion A and are moved (in a way similar to our $\texttt{fence}_{\texttt{acq}}$ encoding in Fig. 10) to either the real or "down" heap (depending on τ).

This combination of techniques results in an automatic support for the proof rule for CAS statements. This completes the core of our Viper encoding, which now handles the complete set of memory access constructs from Fig. 1.

6 Soundness and Completeness

We give a brief overview of the soundness argument for our encoding here, and also discuss where it can be incomplete compared with a manual proof effort; further details are included in App. F of the TR [34].

Soundness. Soundness means that if the Viper encoding of a program and its specification verifies, then there exists a proof of the program and specification using the RSL logics. We can show this property in two main steps. First, we show that the states before and after each encoded statement in the Viper program satisfy several invariants. For example, we never hold permissions to a non-atomic reference's `val` field but not its `init` field. Second, we reproduce a Hoare-style proof outline in the RSL logics. For this purpose, we define a mapping from *states* of the Viper program back to RSL *assertions* and show two properties: (1) When we map the initial and final states of an encoded program statement to RSL assertions, we obtain a provable Hoare triple. (2) Any automatic entailment reasoning performed by Viper coincides with entailments sound in the RSL logics. These two facts together imply that our technique will only verify (encoded) properties for which a proof exists in the RSL logics; i.e. our technique is sound.

Completeness. Completeness means that all programs provable in the RSL logics can be verified via their encoding into Viper. By systematically analysing each rule of these logics, we identify three sources of incompleteness of our encoding: (1) It does not allow one to strengthen the invariant in a Rel assertion; strengthening the requirement on writing does not allow more programs to be verified [36]. (2) For a $\texttt{fence}_{\texttt{acq}}$, our encoding removes *all* assertions from under a $\triangledown$ modality. As explained in Sec. 4, the ability to choose *not* to remove the modality is not useful in practice. (3) The ghost state employed in FSL++ can be defined over a *custom permission structure* (partial commutative monoid), which is not possible in Viper. This is the only incompleteness of our encoding arising in practice; we will discuss an example in Sec. 7.

7 Examples and Evaluation

We evaluated our work with a prototype front-end tool [4], and some additional experiments directly at the Viper level [3]. Our front-end tool accepts a simple input language for C11 programs, closely modelled on the syntax of the RSL logics. It supports all features described in this paper, with the exception of invariant rewriting (*cf.* App. C of the TR [34]) and ghost state (App. D of the TR), which will be simple extensions. We encoded examples which require these

Program	Prototype support	Size (LOC, funcs,loops)	Time (s)	Specs PP	Specs LI	Other Annot.	Coq Annot.
`RSLSpinLock`	✓	7,3,2	10.83	3	1	1	120 [37]
`RSLLockNoSpin`	✓	6,3,1	10.33	3	0	1	84 [22]
`RSLLockNoSpin_err`	✓	6,3,1	9.74	3	0	1	n/a
`RelAcqMsgPass`	✓	15,3,1	10.46	3	0	1	99 [37]
`RelAcqMsgPass_err`	✓	15,3,1	9.57	3	0	1	n/a
`RelAcqDblMsgPassSplit`	✓	21,4,2	10.84	4	0	1	n/a
`RelAcqDblMsgPassSplit_err`	✓	21,4,2	9.86	4	0	1	n/a
`CASModesTest`	✓	23,3,2	18.05	3	0	2	n/a
`CASModesTest_err`	✓	24,3,2	17.50	3	0	2	n/a
`FencesDblMsgPass`	✓	27,4,2	12.32	4	0	3	n/a
`FencesDblMsgPass_err`	✓	27,4,2	10.73	4	0	3	n/a
`FencesDblMsgPassSplit`	✓	24,4,2	12.61	4	0	2	n/a
`FencesDblMsgPassSplit_err`	✓	24,4,2	11.53	4	0	2	n/a
`FencesDblMsgPassAcqRewrite`		24,4,2	15.75	4	0	3	n/a
`RustARCOriginal_err`		10,4,0	37.53	4	0	2	654 [18]
`RustARCStronger`		10,4,0	31.86	4	0	2	n/a
`RelAcqRustARCStronger`		9,4,0	15.75	4	0	2	n/a
`FollyRWSpinlock_err`		24,7,2	28.21	7	2	0	n/a
`FollyRWSpinlockStronger`		26,7,3	21.93	7	3	0	n/a

Fig. 13. The results of our evaluation. Examples including `_err` are expected to generate errors; those with `Stronger` are variants of the original code with less-efficient atomics and a correspondingly different proof. "Time" reports the verification time in seconds, including the generation of the Viper code. Under "Size", we measure lines of code, number of distinct functions/threads, and number of loops. Under "Specs", "PP" stands for the necessary pairs of pre and post-conditions; "LI" stands for loop invariants required. "Other Annot." counts any other annotations needed. For examples that have been verified in Coq, we report the number of manual proof steps (in addition to pre-post pairs) and provide a reference to the proof.

features, additional theories, or custom permission structures manually into Viper, to simulate what an extended version of our prototype will be able to achieve.

Our encoding supports several extra features which we used in our experiments but mention only briefly here: (1) We support the FSL++ rules for *ghost state*: see App. D of the TR. (2) Our encoding handles common spin loop patterns without requiring loop invariant annotations. (3) We support fetch-update instructions (e.g. atomic increments) natively, modelled as a CAS which never fails.

Examples. We took examples from the RSL [37] and FSL [17] papers, along with variants in which we seeded errors, to check that verification fails as expected (and in comparable time). We also encoded the Rust reference-counting (ARC) library [1], which is the main example from FSL++ [18]. The proof there employs a custom permission structure, which is not yet supported by Viper. However, following the suggestion of one of the authors [36], we were able to fully verify two variants of the example, in which some access modes are strengthened, making the code slightly less efficient but enabling a proof using a simpler permission model. For these variants, we required *counting permissions* [10], which we expressed with additional background definitions (see [3] for details, and App. B of the TR [34] for the code). Finally, we tackled seven core functions of a reader-writer-spinlock from the Facebook Folly library [2]. We were able to verify five of them directly.

The other two employ code idioms which seem to be beyond the scope of the RSL logics, at least without sophisticated ghost state. For both functions, we also wrote and verified alternative implementations. The Rust and Facebook examples demonstrate a key advantage of building on top of Viper; both require support for extra theories (counting permissions as well as modulo and bitwise arithmetic), which we were able to encode easily.

Performance. We measured the verification times on an Intel Core i7-4770 CPU (3.40GHz, 16Gb RAM) running Windows 10 Pro and report the average of 5 runs. For those examples supported by our front-end, the times include the generation of the Viper code. As shown in Fig. 13, verification times are reasonable (generally around 10 seconds, and always under a 40 seconds).

Automation. Each function (and thread) must be annotated with an appropriate pre and post-condition, as is standard for modular verification. In addition, some of our examples require loop invariants and other annotations (e.g. on allocation statements). Critically, the number of such annotations is very low. In particular, our annotation overhead is between one and two orders of magnitude lower than the overhead of existing mechanised proofs (using the Coq formalisations for [37, 18] and a recent encoding [22] of RSL into Iris [23]). Such ratios are consistent with other recent Coq-mechanised proofs based on separation logic (e.g. [38]), which suggests that the strong soundness guarantees provided by Coq have a high cost when *applying* the logics. By contrast, once the specifications are provided, our approach is almost entirely automatic.

8 Conclusions and Future Work

We have presented the first encoding of modern program logics for weak memory models into an automated deductive program verifier. The encoding enables programs (with suitable annotations) to be verified automatically by existing back-end tools. We have implemented a front-end verifier and demonstrated that our encoding can be used to verify weak-memory programs efficiently and with low annotation overhead. As future work, we plan to tackle other weak-memory logics such as GPS [35]. Building practical tools that implement such advanced formalisms will provide feedback that inspires further improvements of the logics.

Data Availability Statement and Acknowledgements. The artifact accompanying our submission is available in the TACAS figshare repository [4] at https://doi.org/10.6084/m9.figshare.5900233.

We are grateful to Viktor Vafeiadis and Marko Doko for many explanations of the RSL logics and helpful discussions about our encoding. We thank Christiane Goltz for her work on the prototype tool, and Malte Schwerhoff for implementing additional features. We thank Marco Eilers for his assistance with the online appendix, and Arshavir Ter-Gabrielyan for automating our artifact assembly for various operating systems. We also thank Andrei Dan, Lucas Brutschy and Malte Schwerhoff for feedback on earlier versions of this manuscript.

References

1. ARC (Atomic Reference Counting) Rust library. Available at https://doc.rust-lang.org/std/sync/struct.Arc.html.
2. Facebook Folly reader-writer spinlock implementation. Available at https://github.com/facebook/folly/blob/master/folly/RWSpinLock.h.
3. Online appendix of Viper-encoded examples. Available at http://viper.ethz.ch/onlineappendix-rsl-encoding/.
4. RSL to Viper front-end. figshare. https://doi.org/10.6084/m9.figshare.5900233.
5. P. A. Abdulla, M. F. Atig, A. Bouajjani, and T. P. Ngo. The benefits of duality in verifying concurrent programs under TSO. *CoRR*, abs/1701.08682, 2017.
6. P. A. Abdulla, M. F. Atig, B. Jonsson, and C. Leonardsson. Stateless model checking for power. In *CAV 2016 Proceedings Part II*, pages 134–156, 2016.
7. J. Alglave and P. Cousot. Ogre and pythia: An invariance proof method for weak consistency models. In *POPL 2017*, POPL 2017, pages 3–18, New York, NY, USA, 2017. ACM.
8. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.
9. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.
10. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL '05*, pages 259–270, New York, NY, USA, 2005. ACM.
11. A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against tso. In *ESOP 2013*, ESOP'13, pages 533–553, Berlin, Heidelberg, 2013. Springer-Verlag.
12. J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
13. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 234–245, New York, NY, USA, 2011. ACM.
14. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. *VCC: A Practical System for Verifying Concurrent C*, pages 23–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
15. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In *SEFM*, pages 233–247, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
16. A. Dan, Y. Meshman, M. Vechev, and E. Yahav. Effective abstractions for verification under relaxed memory models. In *VMCAI 2015*, pages 449–466, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
17. M. Doko and V. Vafeiadis. A program logic for C11 memory fences. In *VMCAI*, volume 9583 of *Lecture Notes in Computer Science*, pages 413–430. Springer, 2016.
18. M. Doko and V. Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In *ESOP 2017*, pages 448–475. Springer Berlin Heidelberg, 2017.
19. S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Abstract read permissions: Fractional permissions without the fractions. In *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 315–334. Springer-Verlag, 2013.

20. B. Jacobs. Verifying TSO programs. CW Reports CW660, Department of Computer Science, KU Leuven, May 2014.
21. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, volume 6461 of *LNCS*, pages 304–311. Springer, 2010.
22. J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP 2017*, volume 74 of *LIPIcs*, pages 17:1–17:29. Schloss Dagstuhl–Leibniz, 2017.
23. R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, pages 696–723, New York, NY, USA, 2017. Springer-Verlag New York, Inc.
24. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of LPAR'10*, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.
25. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *ESOP*, volume 5502 of *LNCS*, pages 378–393. Springer-Verlag, 2009.
26. P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In *CAV*, volume 9779 of *LNCS*, pages 405–425. Springer-Verlag, 2016.
27. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *VMCAI*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.
28. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, pages 1–19, London, UK, UK, 2001. Springer-Verlag.
29. 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.
30. J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference—Volume 2*, ACM '72, pages 717–740. ACM, 1972.
31. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE Computer Society Press, 2002.
32. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of PLDI '15*, pages 77–87, New York, NY, USA, 2015. ACM.
33. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.
34. A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs (extended version). *arXiv:1703.06368*, 2018.
35. A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707. ACM, 2014.
36. V. Vafeiadis. Personal communication, December 2016.
37. V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for C11 concurrency. In *OOPSLA*, pages 867–884. ACM, 2013.
38. F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive OS kernels. In *CAV Proceedings Part II*, pages 59–79, Cham, 2016. Springer International Publishing.