Implementing and Verifying Release-Acquire Transactional Memory in C11

SADEGH DALVANDI, University of Surrey, UK
BRIJESH DONGOL, University of Surrey, UK

Transactional memory (TM) is an intensively studied synchronisation paradigm with many proposed implementations in software and hardware, and combinations thereof. However, TM under relaxed memory, e.g., C11 (the 2011 C/C++ standard) is still poorly understood, lacking rigorous foundations that support verifiable implementations. This paper addresses this gap by developing TMS2-RA, a relaxed operational TM specification. We integrate TMS2-RA with RC11 (the repaired C11 memory model that disallows load-buffering) to provide a formal semantics for TM libraries and their clients. We develop a logic, TARO, for *verifying* client programs that use TMS2-RA for synchronisation. We also show how TMS2-RA can be *implemented* by a C11 library, TML-RA, that uses relaxed and release-acquire atomics, yet guarantees the synchronisation properties required by TMS2-RA. We *benchmark* TML-RA and show that it outperforms its sequentially consistent counterpart in the STAMP benchmarks. Finally, we use a simulation-based verification technique to *prove correctness* of TML-RA. Our entire development is supported by the Isabelle/HOL proof assistant.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; *Concurrency*; • **Computing methodologies → Concurrent computing methodologies**.

Additional Key Words and Phrases: Weak Memory, Transactional Memory, C11, Verification, Refinement

ACM Reference Format:
Sadegh Dalvandi and Brijesh Dongol. 2022. Implementing and Verifying Release-Acquire Transactional Memory in C11. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 189 (October 2022), 28 pages. https://doi.org/10.1145/3563352

1 INTRODUCTION

The advent and proliferation of architectures implementing relaxed memory models has resulted in many new challenges in the development of concurrent programs. In the context of the C/C++ relaxed memory model defined by C11[1], over a decade's worth of research has resulted in rigorous semantic foundations [Batty et al. 2016, 2011; Kang et al. 2017; Lahav et al. 2017; Lee et al. 2020; Paviotti et al. 2020], and more recently, logics for reasoning about the correctness of concurrent programs [Dalvandi et al. 2020a, 2022; Doherty et al. 2019; Doko and Vafeiadis 2017; He et al. 2016; Kaiser et al. 2017; Kang et al. 2017; Lahav and Vafeiadis 2015; Vafeiadis and Narayan 2013; Wright et al. 2021]. These works have provided the background necessary to develop high-level abstractions and concurrency libraries over relaxed-memory architectures. Recent works have included reimplementations of concurrent data structures [Dalvandi and Dongol 2021; Dongol et al. 2018b; Emmi and Enea 2019; Krishna et al. 2020; Raad et al. 2019a], including those with relaxed specifications that aim to exploit the additional behaviours allowed by relaxed memory.

[1]C11 refers to the 2011 ISO specification of C/C++.

Authors' addresses: Sadegh Dalvandi, m.dalvandi@surrey.ac.uk, University of Surrey, Guildford, UK; Brijesh Dongol, b.dongol@surrey.ac.uk, University of Surrey, Guildford, UK.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2022 Copyright held by the owner/author(s).
2475-1421/2022/10-ART189
https://doi.org/10.1145/3563352

Our aim for this paper is to implement and verify synchronisation abstractions, fine-tuned for C11, in the form of *transactional memory (TM)* libraries, which provide reusable foundations for high-performance, yet easy to manage concurrency control [Guerraoui and Kapalka 2010; Herlihy and Moss 1993; Shavit and Touitou 1997]. Implementations include those in software (as STM libraries) and hardware (Intel-RTM and Armv9). Other variations include hybrid TM that combine software and hardware TMs and implementations that are natively supported by the compiler (e.g., the C++ TM Lite development). In addition to supporting general-purpose concurrency, TM has also been used to develop transactional concurrent objects and data structures [Assa et al. 2020, 2021; Bronson et al. 2010; Lesani et al. 2022]. Intel's persistent memory development kit (PMDK) [Scargall 2020] extensively promotes the transactional paradigm (though multi-threaded transactions are not directly supported by PMDK's transactions). These prior works have assumed SC transactions, i.e., that transactional access provide the same guarantees as sequentially consistent memory. Our focus is the verification of STMs implemented as a programming language library with *relaxed*, *release*, *acquire* and *release-acquire* accesses providing a pathway towards simplified development of transactional objects (including concurrent data structures) for relaxed memory.

TM implementations provide fine-grained interleaving (for efficiency) that execute with an *illusion of atomicity* (for correctness). A completed transaction may be committed or aborted so that all or none of its effects are externally visible. TM implementations are designed to satisfy a variety of correctness conditions such as (strict) serialisability, opacity, and snapshot isolation, which restrict ordering possibilities of completed transactions. TM has been extensively studied for sequentially consistent (SC) architectures [Lamport 1979], but implementations over relaxed memory are limited.

Prior works on relaxed memory transactions (e.g., [Chong et al. 2018; Dongol et al. 2018a]) have focussed on foundations of *hardware transactions* and their interaction with relaxed memory models, e.g., the expected isolation guarantees, reordering possibilities etc. The work of Chong et al. [2018] also provides for semantics of native C++ transactions. However, native TM support in C++ is still in a state of flux [Spear et al. 2020; Zardoshti et al. 2019] and the underlying designs have changed since the original works by Chong et al. [2018]. Moreover, these semantics are presented in an axiomatic (aka declarative) style, which cannot be used to verify TM implementations, where we require *operational* descriptions of correctness. Therefore, our point of departure is a separate set of works on TM specifications, in particular the TMS2 specification [Doherty et al. 2013], which has been used extensively as a TM specification for standard (i.e., SC) architectures.

More recent works have taken steps towards C++ implementations, including native support of TM within C++ [Zardoshti et al. 2019] and STMs implemented using C++ relaxed memory [Rodriguez and Spear 2020]. However, Zardoshti et al. [2019] do not describe interactions with the C11 relaxed memory model, while Rodriguez and Spear [2020] focus on data race freedom and privatisation guarantees. Neither of these works have a formal semantics, nor are they supported by a verification methodology. (See §7 for a more comprehensive survey of related works.)

Our work addresses several gaps in the current state-of-the-art of transactions for C/C++. We work with RC11, i.e., the *repaired C11* memory model [Lahav et al. 2017]. The RC11 memory model disallows program-order and reads-from cycles, and hence disallows load-buffering behaviour. This restriction greatly simplifies reasoning and variants of RC11 are supported by a number of different logics [Dalvandi et al. 2020a; Dalvandi and Dongol 2021; Dalvandi et al. 2022; Dang et al. 2022; Kaiser et al. 2017; Lahav and Vafeiadis 2015]. Logics that address the full C11 memory model (allowing load buffering) have also been developed, but proofs in these logics are limited to small litmus tests [Svendsen et al. 2018; Wright et al. 2021].

We develop: (i) a reusable *specification* of TM that provides well-defined guarantees to those developing client programs; (ii) techniques for *verifying client programs* in C11 that use such

TM abstractions; (iii) *implementations* of TM in C11, including their rigorous verification; and (iv) *mechanisation* of the verification described above in the theorem prover Isabelle/HOL. We discuss these contributions in more detail below.

Correctness specifications. To enable verification, we start with the *TMS2* specification [Doherty et al. 2013]. TMS2 implies the *TMS1* specification, which is known to be both necessary and sufficient for *observational refinement* (of client programs) [Attiya et al. 2018]. The main difference between TMS1 and TMS2 is that TMS1 allows aborted transactions to observe different serialization orders [Lesani et al. 2012]. In contrast, TMS2, like *opacity* [Guerraoui and Kapalka 2010], ensures strict serializability of the committed transactions and furthermore that aborted transactions are consistent with the serialisation order. Although more restrictive than TMS1, TMS2 has been shown to be a robust correctness condition that is useful in practice, providing a specification for a number of TM implementations under SC [Armstrong and Dongol 2017; Armstrong et al. 2017; Derrick et al. 2018; Doherty et al. 2016]. Under relaxed memory, the TMS2 specification is inadequate since it does not provide any of the *client-side guarantees* required by relaxed memory libraries [Dalvandi and Dongol 2021; Dang et al. 2022; Dongol et al. 2018b; Raad et al. 2019a, 2018]. Such guarantees are required under relaxed memory since writes in one thread may not be propagated to other threads unless the library is properly synchronised (cf. the message passing litmus tests [Alglave et al. 2014]).

Our first contribution is the adaptation of TMS2 to address this issue. In particular, our specification, TMS2-RA, provides a flexible meaning of correctness, allowing a client to specify *relaxed*, *releasing*, *acquiring* and *release-acquiring* transactions (see §3), mimicking the memory annotations of C11 atomics [Batty et al. 2011]. This provides greater flexibility in TM design; we develop a model in which these different types of transactions co-exist within the same TM system.

Client verification. Our second contribution (see §5) is a verification technique for relaxed-memory client programs that use TMS2-RA. In particular, we prove correctness of several variations of the message passing litmus test, synchronised through TMS2-RA transactions, to show that TMS2-RA behaves as expected. In particular, we show how different client-side guarantees are achieved depending on the type of synchronisation guarantee (relaxed, releasing or acquiring) provided by the transaction in question.

Our verification framework includes a new logic, TARO, capable of efficiently reasoning about the *views* of a client programs [Dalvandi et al. 2022; Kaiser et al. 2017]. This means that the correctness of programs can be established using a standard Owicki-Gries reasoning framework [Dalvandi et al. 2020a; Dalvandi and Dongol 2021; Owicki and Gries 1976].

Implementation, benchmarking and verification. Our third contribution is the implementation and full verification of an STM algorithm that uses C11 relaxed/release-acquire atomics and implements TMS2-RA. Our implementation is an adaptation of Dalessandro et al's *Transactional Mutex Lock (TML)* [Dalessandro et al. 2010], which presents a simple mechanism for synchronising transactions optimised for read-heavy workloads. TML is synchronised using a single global lock, and allows multiple concurrent read-only transactions, but at most one writing transaction, i.e., a writing transaction causes all other concurrent transactions to abort.

Interestingly, our adapted algorithm, which we call TML-RA, allows more concurrency than TML by exploiting the parallelism afforded by relaxed and release-acquire C11 atomics. Moreover, a writing transaction does not force other read-only transactions to abort, allowing greater read/write parallelism (see §4). We show that this theoretical speedup manifests in real implementations and TML-RA outperforms its SC counterpart in all STAMP benchmarks (see §4.3).

(a) Unsynchronised MP (b) Synchronised MP

Fig. 1. Message passing (MP) in C11

Furthermore, we use a simulation-based verification method for the C11 memory model [Dalvandi and Dongol 2021] to prove correctness of TML-RA. This proof establishes trace refinement between TML-RA and TMS2-RA, which ensures that all observable behaviours of TML-RA are observable behaviours of TMS2-RA. Thus, if a client program C is proved correct when it uses TMS2-RA, then C will also be correct if we replace calls to TMS2-RA in C by calls to TML-RA.

Mechanisation. Our fourth contribution is the mechanisation of all proofs presented in the paper in the Isabelle/HOL proof assistant (available as supplementary material [Dalvandi and Dongol 2022a]). This includes the operational semantics of C11 integrated with TMS2-RA, soundness of all TARO rules, the use of TARO to prove several client programs that use TMS2-RA, and finally the proof of simulation between TMS2-RA and TML-RA.[2]

Overview. This paper is structured as follows. We describe our requirements for relaxed and release-acquire transactions in §2. We formalise this semantics in §3 via the TMS2-RA specification, and describe its integration with a view-based semantics for RC11 with release-acquire atomics [Dalvandi et al. 2020a]. In §4, we provide an examplar implementation and benchmarking results for TML-RA. In §5, we present our logic for reasoning about release-acquire transactional memory, which provides a method of reasoning about client programs that use the TMS2-RA specification. Finally, in §6, we present a proof of correctness of TML-RA via refinement w.r.t. TMS2-RA.

2 TRANSACTIONAL GUARANTEES IN C11

A TM specification in a relaxed memory setting has two distinct sets of goals. The first set must guarantee the expected behaviours of transactions, e.g., serializability, opacity etc. The second must provide client-side guarantees, e.g., release-acquire synchronisation, observational refinement etc. We consider both in our TMS2-RA specification (see Fig. 4).

2.1 Release-Acquire Synchronisation

Prior to detailing the design choices of TMS2-RA, we recap the basics of release-acquire synchronisation in C11, including a recently developed timestamp-based operational semantics, which is the semantics assumed by TMS2-RA.

The fragment of C11 we focus on is the RC11-RAR fragment. The first "R" denotes the *repairing model* [Lahav et al. 2017], which precludes 'thin-air' behaviour by disallowing memory operations within a thread to be reordered. The "RAR" refers to the fact that the model includes *release-acquire* as well as *relaxed* atomics [Dalvandi et al. 2020b; Doherty et al. 2019]. [3] For the remainder of this paper, we simply write C11 to refer to RC11-RAR.

[2]Our development may be found in [Dalvandi and Dongol 2022a].

[3]Note that extending this model to include other types of C11 synchronisation (e.g., SC fences) and relaxations that allow intra-thread ordering is possible [Wright et al. 2021], but these extended models are not so interesting for the purposes of this paper, and the additional complexity that they induce detracts from our main contributions.

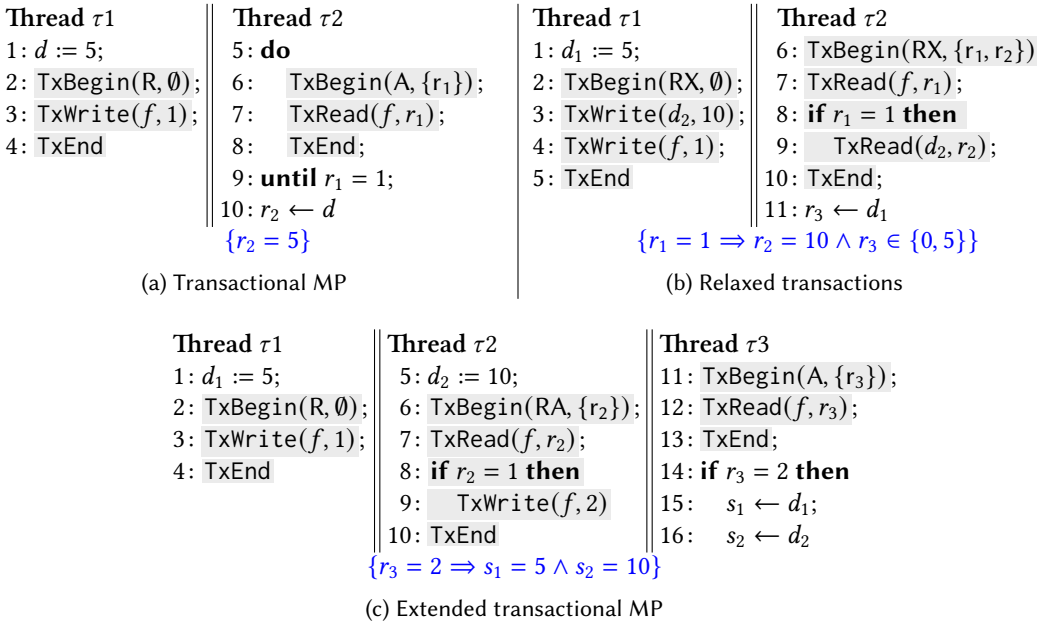


Fig. 2. Transactional memory client interactions

We explain the main ideas behind release-acquire synchronisation using the message passing (MP) litmus test in Figs. 1a and 1b. It comprises two shared variables: d (for data) and f (for a flag), both of which are initially 0. Under SC, the postcondition of the program is $r_2 = 5$ because the loop in thread $\tau 2$ only terminates after f has been updated to 1 in thread $\tau 1$, which in turn happens after d is set to 5. Therefore, the only possible value of d that thread $\tau 2$ can read is 5.

However, in Fig. 1a, all read/write accesses of d and f are *relaxed*, and hence the program can only establish the weaker postcondition $r_2 = 0 \vee r_2 = 5$ since it is possible for thread $\tau 2$ to read 0 for d at line 4. In particular, reading 1 for f does not guarantee that thread $\tau 2$ will read 5 for d.

This anomaly is corrected in Fig. 1b where the highlighted code depicts the necessary changes. In particular, we introduce a *release* annotation (line 2) as well as an *acquire* annotation (line 3), which together induces a *happens-before* relation if the read of f at line 3 reads from the write at line 2 (see [Batty et al. 2011]). This in turn ensures that thread $\tau 2$ sees the most recent write to d at line 1. We explain how relaxed accesses and release-acquire synchronisation is formalised by the operational semantics in §3.1.

2.2 Transactional Message Passing

We now describe the guarantees provided by our transactional model in the context of a client program. Like standard reads and writes in C11, we allow transactions to be combined with a synchronising annotation, which may be one of relaxed (RX), releasing (R), acquiring (A), or release-acquiring (RA). These annotations dictate whether or not a transaction induces a client-side happens before. In particular, client-side happens-before is induced from thread $\tau 1$ to thread $\tau 2$ if (i) a read in transaction t_2 executed by $\tau 2$ reads-from a write in transaction t_1 executed by $\tau 1$, (ii) t_1 contains a release annotation (either R or RA), and (iii) t_2 contains an acquire annotation (either A or RA). We illustrate the implications of these annotations via the examples in Fig. 2, where the highlights

are used to identify the transactions. We assume that a client provides a transaction with a set of registers that it may use when the transaction begins (see §3).

Fig. 2a describes a transactional variation of MP. Thread $\tau 1$ comprises a (non-transactional) relaxed write on d followed by a transactional write of the flag, f. Thread $\tau 2$ contains a transactional read of f within a loop that terminates if $\tau 2$ reads 1 for f. After the loop terminates, $\tau 2$ performs a (non-transactional) relaxed read of d. In this example, like Fig. 1b, the release and acquire annotations induce a happens-before relation from $\tau 1$ to $\tau 2$ and hence ensure that the read of d in $\tau 2$ does not return the stale value, 0.

Fig. 2b describes a program that uses a relaxed transaction. The postcondition of the program considers the case where the transaction in $\tau 1$ occurs before the transaction in $\tau 2$ since the antecedent assumes that $r_1 = 1$, i.e., the read of f at line 7 reads the write of f at line 4. In this example, both transactions are relaxed, and hence, the ordering of transactions above does not induce a happens before from $\tau 1$ to $\tau 2$. Thus, the read of d_1 at line 11 is not guaranteed to see the write of d_1 at line 1, i.e., the final value of r_3 is either 0 or 5. However, since the write and read of d_2 occurs within the transactions of $\tau 1$ and $\tau 2$, respectively, if $r_1 = 1$, then $\tau 2$ is guaranteed to read 10 for d_2.

Finally, Fig. 2c demonstrates a program with an RA transaction. The antecedent of the program's postcondition implies that the transaction in $\tau 3$ occurs after the transaction in $\tau 2$, which in turn occurs after the transaction in $\tau 1$. Here, the transaction annotations ensure that the writes to d_1 and d_2 (at lines 1 and 5) performed by the *client* are seen by the client reads at lines 15 and 16. This is because the transaction in $\tau 2$ (annotated by RA) is guaranteed to synchronise with the transaction in $\tau 1$ (annotated by R), and similarly, the transaction in $\tau 3$ (annotated by A) is guaranteed to synchronise with the transaction in $\tau 2$ (annotated by RA). Note that if the transaction in $\tau 2$ was only releasing, then $\tau 1$ and $\tau 2$ would not synchronise, and the read at line 15 may return either 0 or 5. Yet, the read at line 16 would still be guaranteed to return 10 for d_2 since $\tau 2$ and $\tau 3$ synchronise. If the transaction in $\tau 2$ was only acquiring, then $\tau 2$ and $\tau 3$ would not synchronise. In this case, although $\tau 1$ and $\tau 2$ have synchronised, neither of the reads at lines 15 and 16 are guaranteed to return the new writes at lines 1 and 5.

Deciding a transaction's synchronisation flag ultimately comes down to the needs of a client program, much like `memory_order` parameters on `atomic_compare_exchange` instructions in C11 [cppreference.com 2022]. Client programs that require message passing through transactions would use release-acquire, while others may only require relaxed annotations.

3 RELEASE-ACQUIRE TM SPECIFICATION

With the basic requirements for release-acquire and transactional synchronisation in place, we work towards a formal TM specification. Our specification will be closely tied to an operational semantics for C11 with timestamped writes and per-thread views [Dalvandi et al. 2020a; Dolan et al. 2018; Kaiser et al. 2017; Kang et al. 2017; Podkopaev et al. 2016] (see §3.1). We integrate this model with a TM specification in §3.3.

3.1 View-Based Operational Semantics

As discussed above, in our model, the C11 relaxed memory state is formalised by *timestamped writes*. Instead of mapping each location to a value, the state contains a set of writes $\mathtt{writes} \subseteq \mathit{Write}$, where $\mathit{Write} = \mathit{Loc} \times \mathit{Val} \times \mathit{TS}$ represents a write to a location Loc with value Val and $\mathit{TS} \mathrel{\hat{=}} \mathbb{Q}$ is the set of possible timestamps. If $w \in \mathit{Write}$ and $w = (x, v, q)$, then we let $\mathrm{loc}(w) \mathrel{\hat{=}} x$, $\mathrm{val}(w) \mathrel{\hat{=}} v$, $\mathrm{tst}(w) \mathrel{\hat{=}} q$, be the functions that extract the location, value and timestamp of w, respectively.

A *view* is a mapping from a location to a write of that location, i.e., $\mathit{View} \mathrel{\hat{=}} \mathit{Loc} \to \mathit{Write}$. To define the allowable reads by each thread to each location, the state also records a *thread view* for

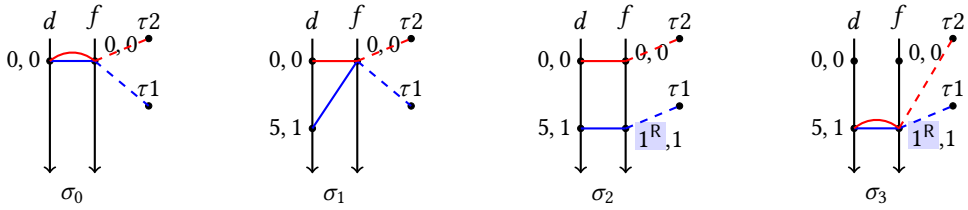


Fig. 3. Synchronised message passing views

each thread defined by a function

$$\texttt{tview} : \mathit{TId} \to \mathit{View}$$

where $\mathit{TId} \triangleq \mathbb{N}$ is the set of thread identifiers. A thread may read from *any* write whose timestamp is no smaller than the thread's current view. Thus, the *observable values* (OV), i.e., the set of values that thread τ can read for location x is

$$OW_\tau(x) \triangleq \left\{ w \in \texttt{writes} \;\middle|\; \mathsf{loc}(w) = x \wedge \mathsf{tst}(w) \geq \mathsf{tst}(\texttt{tview}_\tau(x)) \right\}$$
$$OV_\tau(x) \triangleq \{\mathsf{val}(w) \mid w \in OW_\tau(x)\}$$

A write may be introduced at any timestamp greater than the thread's current view (with a caveat that ensures atomicity of read-modify-writes, see [Dalvandi et al. 2020a; Doherty et al. 2019] for details).

Finally, to formalise release-acquire synchronisation, a state in the timestamp model also includes a notion of a *modification view*,

$$\texttt{mview} : \mathit{Write} \to \mathit{View}$$

which is a function that records the thread view of the executing thread when a new write is introduced to `writes`. In particular, if thread τ introduces a new write w to `writes` and $\texttt{tview}_\tau$ is updated to view in this new state, then `mview` is also updated so that $\texttt{mview}_w = \mathit{view}$ in the new state. This information is used to update thread views in case release-acquire synchronisation occurs.

Formally, when threads synchronise, a new view is calculated using an operator '$\otimes$', which is defined as follows. Given $V_1, V_2 \in \mathit{View}$, we have

$$V_1 \otimes V_2 \triangleq \lambda x.\ \textbf{if } \mathsf{tst}(V_2(x)) \leq \mathsf{tst}(V_1(x)) \textbf{ then } V_1(x) \textbf{ else } V_2(x)$$

which constructs a new view by taking the write with the larger timestamp for each location x.

Example 1 (Synchronised MP)**.** Consider Fig. 3, which depicts a possible execution of the program in Fig. 1b. Each "v, i" represents a "value, timestamp" pair for the location in question. The initial state is σ_0, where the views of threads $\tau 1$ and $\tau 2$ are both the initial writes. State σ_1 occurs after executing line 1, where the view of $\tau 1$ is updated to the new write on d. Similarly, σ_2 occurs after executing line 2. Note that the new write is tagged with a release annotation. Moreover, the operational semantics guarantees that in σ_2, we have $\sigma_2.\texttt{mview}_{(f,1,1)}(d) = (d, 5, 1)$, i.e., the modification view of the write $(f, 1, 1)$ returns $(d, 5, 1)$ for d (since this was the thread view of $\tau 1$ for d when the write at line 2 occurred).

Finally, σ_3 depicts the state after execution of line 3, where the read returns the value 1 for f. In this case, the thread view of $\tau 2$ for f is updated to the new read. More importantly, due to release-acquire annotations the semantics enforces that the thread view of $\tau 2$ for d in σ_3 is *also* updated to the new modification view, i.e., $\sigma_2.\texttt{mview}_{(f,1,1)}(d)$. Thus, after state σ_3, $\tau 2$ will no longer be able to return the stale value 0 for d.

The key difference in execution of the unsynchronised example (Fig. 1a) is that the read at line 3 does not update $\texttt{tview}_{\tau 2}(d)$. Hence, for the state of Fig. 1a analogous to σ_3, the view of $\tau 2$ for d will remain at the initial write, allowing it to return a stale value.

3.2 TMS2

First, we consider the TMS2 specification, which is our TMS2-RA specification *without* any client-side release-acquire guarantees. This is given by the unhighlighted components of Fig. 4, which correspond precisely to the *internal actions* of TMS2 [Doherty et al. 2013].[4] Note that each action of Fig. 4 is atomic and guarded by the conditions defined in **pre**. If all the conditions in **pre** hold the transition is *enabled*, and the corresponding action atomically updates the state according to the assignments and functions in **eff**. If some condition in **pre** does not hold then the transition is *blocked*. We use $\sqcap$ to denote a non-deterministic choice (see [Lynch 1996] for details).

TMS2 is a close operational approximation of *opacity* [Guerraoui and Kapalka 2010]. The differences between TMS2 and opacity are minor [Lesani et al. 2012], and much of the discussion below applies equally to opacity. TMS2 (and opacity) distinguishes between *completed* and *live* transactions, where completed transaction may either be *committed* or *aborted*. TMS2 guarantees the existence of a total order $\prec$ over *all* transactions such that:

(1) if transaction t_1 executes TxEnd before t_2 executes TxBegin, then $t_1 \prec t_2$;
(2) for any transaction t, if $\prec_{\downarrow t}$ is the strict downclosure of t w.r.t. $\prec$ and m is the memory obtained by applying the committed transactions in $\prec_{\downarrow t}$ in order, then
   - all internal reads in t for a variable x are consistent with the last write to x in t, and
   - all external reads of t are consistent with m.

Note that conditions (1) and (2) together imply strict serialisability of the transactions. Condition (2) additionally ensures that no transaction reads from an aborted or live transaction since all external writes can be explained by the prior writes of committed transactions only. Moreover, reads of all transactions (including aborted and live transactions) never return a spurious value, i.e., each non-aborting read can be explained by prior committed transactions.

The existence of the total order mentioned above is guaranteed by the TMS2 specification as follows. Each transaction t comprises a local read set, $\texttt{rdSet}_t$, local write set, $\texttt{wrSet}_t$, and variable, $\texttt{status}_t$ that is used to model control flow within a transaction. If the status of t is NOTSTARTED, t may transition to status READY if a thread τ executes $\texttt{TxBegin}_\tau$. Once ready, τ may execute some number of TxRead and TxWrite operations, or TxEnd, which sets the status of t (the transaction that τ is executing) to COMMITTED. Note that if transaction t is READY, it may transition to status ABORT at any time. Moreover, in some circumstances, t may be forced to abort because all other transitions of t are blocked.

To ensure read/write consistency, TMS2 uses a sequence of memories M, where a memory is a mapping from locations to values. A transaction t records the earliest memory it can read from by setting $\texttt{beginIdx}_t$ to the last index of M when t executed $\texttt{TxBegin}_\tau$. Moreover, each committing writing transaction t constructs a new memory $N = \mathit{last}(M) \oplus \texttt{wrSet}_t$ which is the memory $\mathit{last}(M)$ overwritten with the write set of t. It then appends N to the end of M (see TxEndWR).

We differentiate between internal reads TxReadInt and external reads TxReadExt, by whether the read location x is in the write set of the executing transaction, t. An internal read of x simply returns the value of x in the write set of t. An external read of x non-deterministically picks a memory index i. This read is enabled iff i is a valid index (i.e., is between $\texttt{beginIdx}_t$ and the last memory index, $|M| - 1$) and the read set of t is consistent with M_i (i.e., the memory at index i). In

[4] TMS2-RA, like TMS2 is presented as an I/O automaton [Lynch 1996]. For simplicity, we eschew the external actions, but they can easily be included to formalise the TM interface.

case an external read occurs, the read set is updated and the value read is returned. This means that all external reads in t are validated with respect to *some* memory snapshot between $\texttt{beginIdx}_t$ and the maximum memory index. Note that it is possible for two different reads to validate w.r.t. *different* memory snapshots.

TMS2 prescribes a *lazy* write-back strategy via `TxWrite`, where writes are cached in a local write set until the commit occurs (as described above). However, as we shall see, this does not preclude implementations of TMS2 that use *eager* write-backs, where writes occur in memory at the time of writing (see [Derrick et al. 2018]). In fact, the TML-RA algorithm, our main case study in this paper, *is* such an eager algorithm (see §4).

We split the commit phase into two cases: *read-only* (modelled by `TxEndRO`) and *writing* (modelled by `TxEndWR`). Since all reads are validated at the time of reading, a read-only transaction can simply commit the transaction. On the other hand, the writing transaction must ensure its reads are valid w.r.t. the last memory snapshot. The effect of this transition is to install a new memory snapshot as described above.

The final component of t is a local set $\texttt{regs}_t$ that is used to keep track of the set of registers that the transaction has written to. A client provides the set of registers to be used by each transaction when the transaction begins. These registers are set to a special value $\perp$ when a transaction aborts to ensure that no value read by t is seen outside t.

3.3 TMS2-RA

We now discuss the release-acquire extensions of TMS2-RA, as defined by the I/O automata algorithm in Fig. 4, including the highlighted components. The key extension of TMS2-RA is its ability to synchronise client threads, thus allowing it to cope with the examples in Fig. 2. Formally, this is achieved by ensuring TMS2-RA synchronises the *thread view* of the client whenever transactional release-acquire synchronisation occurs.

We introduce two new local variables in transaction t. Namely, $\texttt{synctype}_t$, which records the type of synchronisation of t, and $\texttt{seenIdxs}_t$, which records the set of all memory indices seen by t that are either releasing or release-acquiring. We also introduce a new thread local variable `txview`, which records the *transaction thread view* of τ. The transaction thread view is similar to *thread view* introduced in §3.1. The difference here is in the definition of *View*. In this context the *View* is a function that maps the threads to memory indexes of M. The transaction view of τ is the smallest memory in M that can be read by any transaction t that was begun by thread τ. We also introduce two global variables. Namely S, which is a sequence recording the type of each committed writing transaction that installs each new memory in M, and V which is a sequence of modification views for each new memory in M. Thus, in TMS2-RA, memory M_i has synchronisation type S_i, and modification view V_i.

The transactional operations of TMS2 are modified as follows. In $\texttt{TxBegin}_t$, we take as input the type of synchronisation transaction t is to perform, and store this value in $\texttt{synctype}_t$. $\texttt{TxBegin}_t$ calls $\texttt{TxBeginV}_t$ with another input m, which is an index to a visible memory M. We also initialise $\texttt{seenIdxs}_t$ to the empty set. In $\texttt{TxReadExt}_t(x, i)$, i.e., a transition for external read of x from memory index i, we record the index i in $\texttt{seenIdxs}_t$ if the memory M_i is releasing or release-acquiring.

When a transaction ends (for both read-only and writing transactions), if the transaction is acquiring or release-acquiring and $\texttt{seenIdxs}_t$ is non-empty, we construct a new view `nv` to be the maximum modification view for each transaction in $\texttt{seenIdxs}_t$ using the function *view*. We use this to synchronise the client thread's view by updating $\texttt{tview}_\tau$ to $\texttt{tview}_\tau \otimes \texttt{nv}$. For a writing transaction, we record this new view of the client in V so that any future transactions that synchronise with this new transaction does so with respect to this view. Finally, once a transaction ends, it updates `txview` to the largest index in `seenIdxs`.

$\mathtt{TxBeginV}_\tau(\mathit{sflag}, m, \mathit{regSet})$
pre $\mathtt{status}_t = \mathtt{NOTSTARTED}$
$\mathtt{txn}_\tau = \bot$
$m \in \mathit{OM}_\tau$
eff $\mathtt{wrSet}_t := \emptyset$
$\mathtt{rdSet}_t := \emptyset$
$\mathtt{beginIdx}_t := m$
$\mathtt{seenIdxs}_t := \emptyset$
$\mathtt{synctype}_t := \mathit{sflag}$
$\mathtt{regs}_t := \mathit{regSet}$
$\mathtt{txn}_\tau := t$
$\mathtt{status}_t := \mathtt{READY}$

$\mathtt{TxWrite}_\tau(x, v)$
pre $\mathtt{status}_t = \mathtt{READY}$
$\mathtt{txn}_\tau = t$
eff $\mathtt{wrSet}_t := \mathtt{wrSet}_t \cup \{x \mapsto v\}$

$\mathtt{TxReadInt}_\tau(x, r)$
pre $\mathtt{status}_t = \mathtt{READY}$
$x \mapsto v \in \mathtt{wrSet}_t$
$r \in \mathtt{regs}_t$
$\mathtt{txn}_\tau = t$
eff $r := v$

$\mathtt{TxReadExt}_\tau(x, i, r)$
pre $\mathtt{status}_t = \mathtt{READY}$
$x \notin \mathbf{dom}(\mathtt{wrSet}_t)$
$\mathtt{beginIdx}_t \le i < |M|$
$\mathtt{rdSet}_t \subseteq M_i$
$\mathtt{txn}_\tau = t$
eff $\mathtt{rdSet}_t := \mathtt{rdSet}_t \cup \{x \mapsto M_i(x)\}$
if $S_i \in \{\mathsf{R}, \mathsf{RA}\}$
then $\mathtt{seenIdxs}_t := \mathtt{seenIdxs}_t \cup \{i\}$
$r := M_i(x)$

$\mathtt{TxRead}_\tau(x, r) =$
$\mathtt{TxReadInt}_\tau(x) \sqcap \bigsqcap_i \mathtt{TxReadExt}_\tau(x, i, r)$

$\mathtt{TxEndRO}_\tau$
pre $\mathtt{status}_t = \mathtt{READY}$
$\mathtt{wrSet}_t = \emptyset$
$\mathtt{txn}_\tau = t$
eff $\mathtt{status}_t := \mathtt{COMMIT}$
if $\mathtt{synctype}_t \in \{\mathsf{A}, \mathsf{RA}\} \wedge \mathtt{seenIdxs}_t \neq \emptyset$
then
let $\mathtt{nv} = \mathit{view}(\mathtt{seenIdxs}_t, \mathsf{V})$ **in**
$\mathtt{tview}_\tau := \mathtt{tview}_\tau \otimes \mathtt{nv}$
$\mathtt{txn}_\tau := \bot$
$\mathtt{txview}_\tau := \max(\mathtt{seenIdxs}_t)$

$\mathtt{TxEndWR}_\tau$
pre $\mathtt{status}_t = \mathtt{READY}$
$\mathtt{wrSet}_t \neq \emptyset$
$\mathtt{rdSet}_t \subseteq \mathit{last}(M)$
$\mathtt{txn}_\tau = t$
eff $M := M \cdot (\mathit{last}(M) \oplus \mathtt{wrSet}_t)$
$\mathtt{status}_t := \mathtt{COMMIT}$
if $\mathtt{synctype}_t \in \{\mathsf{A}, \mathsf{RA}\} \wedge \mathtt{seenIdxs}_t \neq \emptyset$
then
let $\mathtt{nv} = \mathit{view}(\mathtt{seenIdxs}_t, \mathsf{V})$ **in**
$V := V \cdot (\mathtt{tview}_\tau \otimes \mathtt{nv})$
$\mathtt{tview}_\tau := \mathtt{tview}_\tau \otimes \mathtt{nv}$
else $V := V \cdot \mathtt{tview}_\tau$
$S := S \cdot \mathtt{synctype}_t$
$\mathtt{txn}_\tau := \bot$
$\mathtt{txview}_\tau := \max(\mathtt{seenIdxs}_t)$

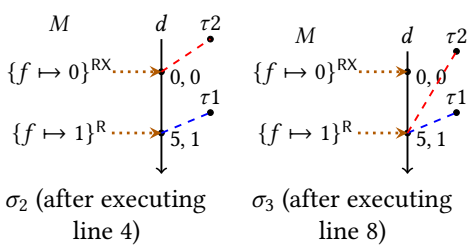
$\mathtt{Abort}_\tau$
pre $\mathtt{status}_t = \mathtt{READY}$
$\mathtt{txn}_\tau = t$
eff $\forall s \in \mathtt{regs}_t \,.\, s := \bot$
$\mathtt{txn}_t := \bot$
$\mathtt{status}_t := \mathtt{ABORT}$

$\mathtt{TXBegin}_\tau(\mathit{sflag}, \mathit{regSet}) = \bigsqcap_m \mathtt{TXBeginV}_\tau(\mathit{sflag}, m, \mathit{regSet})$
$\mathtt{TxEnd}_\tau = \mathtt{TxEndRO}_\tau \sqcap \mathtt{TxEndWR}_\tau$

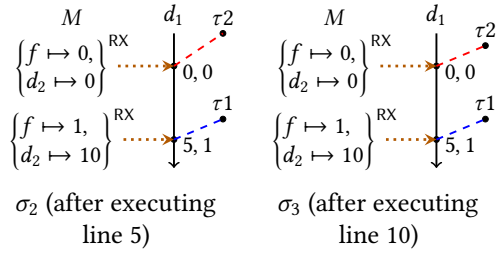
where
$M : \mathit{seq}(\mathit{Loc} \to \mathit{Val})$, initially $M = \langle(\lambda v \in \mathit{Loc}.0)\rangle$
$V : \mathit{seq}(\mathit{Loc} \to \mathit{TS})$, initially $V = \langle(\lambda v \in \mathit{Loc}.0)\rangle$
$\mathit{OM}_\tau = \{n \mid n \ge \mathtt{txview}_\tau \wedge n \le |M| - 1\}$
$S : \mathit{seq}\{\mathsf{RX}, \mathsf{R}, \mathsf{A}, \mathsf{RA}\}$, initially $S = \langle\mathsf{RX}\rangle$
$\mathit{view}(\mathit{Idxs}, \mathit{Vf}) = \lambda l \in \mathit{Loc}.\ \mathit{maxWr}\{\mathit{Vf}_i(l) \mid i \in \mathit{Idxs}\}$

Fig. 4. TMS2-RA specification: highlighted components are extensions necessary for client synchronisation for C11 transactions. We assume that the transactions are executed by thread τ. Moreover, let $Q \cdot a$ be the sequence Q appended with element a and $f \oplus g$ be the function f overridden by function g. Finally, let maxWr be a function that returns the write with the largest timestamp in the given set of writes.

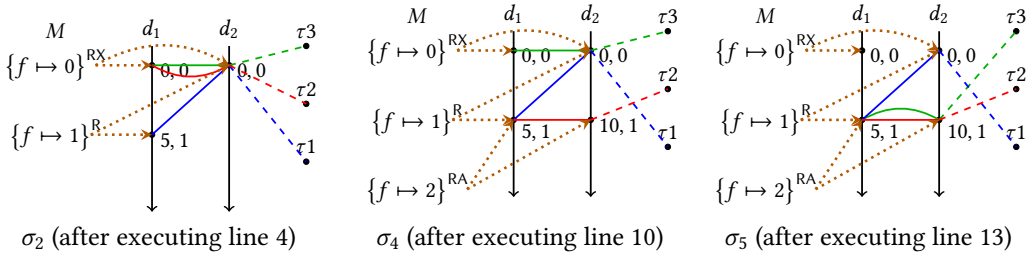
We demonstrate the interaction of TMS2-RA and a client program by considering the views of three possible executions of the programs in Fig. 5. Unlike the trace considered in Fig. 3, we only



(a) Views for transactional MP (Fig. 2a)



(b) Views for relaxed transactions (Fig. 2b)



(c) Views for release-acquire transaction chain (Fig. 2c)

Fig. 5. Views for the transaction-based client program from Fig. 2

show the most critical transitions. The memory sequence M of TMS2-RA is clear from the figures. We represent S by the superscripts on each state of M, and the modification views V by the dotted arrows ($\cdots\!\!\rightarrow$) from each state of M.

Fig. 5a represents part of an execution of the program in Fig. 2a. In the execution depicted, we assume that all of thread $\tau 1$ executes before $\tau 2$. Here, σ_2 is the state after executing line 4, where $\tau 1$ has introduced a new write to d and then executed its (releasing) transaction, introducing a new memory snapshot whose modification view becomes the new write of d (since $\tau 1$'s thread view is at this new write). Then, when $\tau 2$ executes its (acquiring) transaction that reads 1 from f, it *synchronises* with the latest memory snapshot, causing $\tau 2$'s thread view to be the new write of d as well. This is analogous, as required, to the way in which views are updated in C11 (see Fig. 3).

Fig. 5b represents a part execution of the program in Fig. 2b. Again, we assume a complete execution of thread $\tau 1$ followed by $\tau 2$. State σ_2 is the state after executing line 5, where $\tau 1$ has introduced a new write to d_1. Now consider the state σ_3 (the state after execution of line 10), where $\tau 1$ introduces a new memory snapshot in M with annotation RX and modification view pointing to the new write on d_1. When $\tau 2$ continues execution, its transaction must be ordered after the latest memory snapshot, but this will *not* induce a release-acquire synchronisation. This means that $\tau 2$'s view of d_2 will not be updated. However, since $\tau 2$'s transaction occurs after $\tau 1$'s transaction, $\tau 2$ is guaranteed to read 10 for d_2. Note that since the transaction executed by $\tau 2$ is a read-only relaxed transaction, the view of $\tau 2$ of the client variable (d_1) is unchanged. However, $\tau 2$'s view of the transactional memory (not shown in the diagrams) will be updated to the new memory state $\{f \mapsto 1, d_2 \mapsto 10\}$.

Finally, Fig. 5c represents part of an execution of Fig. 2c comprising the complete execution of $\tau 1$, $\tau 2$ then $\tau 3$ in order. State σ_2 represents the state after executing line 4, where the modification view of the newly installed memory is consistent with the view of the executing thread $\tau 1$. Then,

$$\textsc{TxBegin}_\tau(sflag, regSet)\ \frac{\gamma.\mathtt{status}_t = \mathtt{NOTSTARTED} \qquad \gamma.\mathtt{txn}_\tau = \bot \qquad m \in vmems_\tau}{lst, \gamma, \beta \rightsquigarrow_\tau \ lst, \gamma \begin{bmatrix} \mathtt{beginIdx}_t := m, \mathtt{seenIdxs}_t := \emptyset, \mathtt{rdSet}_t := \emptyset, \\ \mathtt{wrSet}_t := \emptyset, \mathtt{synctype}_t := sflag, \mathtt{regs}_t := regSet, \\ \mathtt{status}_t = \mathtt{READY}, \mathtt{txn}_\tau := t \end{bmatrix}, \beta}$$

$$\textsc{TxWrite}_\tau(l, v)\ \frac{\gamma.\mathtt{status}_t = \mathtt{READY} \qquad \gamma.\mathtt{txn}_\tau = t}{lst, \gamma, \beta \rightsquigarrow_\tau \ lst, \gamma \left[\mathtt{wrSet}_t := \gamma.\mathtt{wrSet}_t \cup \{l \mapsto v\}\right], \beta}$$

$$\textsc{TxRead}_\tau(l, r)\ \frac{\begin{array}{c} \gamma.\mathtt{status}_t = \mathtt{READY} \qquad \gamma.\mathtt{txn}_\tau = t \qquad r \in \gamma.\mathtt{regs}_t \\ (l \in \mathbf{dom}(\gamma.\mathtt{wrSet}_t) \vee (\gamma.\mathtt{beginIdx}_t \leq i \ \wedge \gamma.\mathtt{rdSet}_t \subseteq \gamma.M_i)) \\ v = \mathbf{if}\ l \notin \mathbf{dom}(\gamma.\mathtt{wrSet}_t)\ \mathbf{then}\ \gamma.M_i(l)\ \mathbf{else}\ \gamma.\mathtt{wrSet}_t(l) \\ \mathtt{seenIdxs}' = \mathbf{if}\ l \notin \mathbf{dom}(\gamma.\mathtt{wrSet}_t) \wedge \gamma.S_i \in \{\mathsf{R}, \mathsf{RA}\} \\ \qquad \mathbf{then}\ \gamma.\mathtt{seenIdxs}_t \cup \{i\}\ \mathbf{else}\ \gamma.\mathtt{seenIdxs}_t \\ \mathtt{rdSet}' = \mathbf{if}\ l \notin \mathbf{dom}(\gamma.\mathtt{wrSet}_t)\ \mathbf{then}\ \gamma.\mathtt{rdSet}_t \cup \{l \mapsto v\}\ \mathbf{else}\ \gamma.\mathtt{rdSet}_t \end{array}}{lst, \gamma, \beta \rightsquigarrow_\tau \ lst[r := v], \gamma \left[\mathtt{rdSet}_t := \mathtt{rdSet}', \mathtt{seenIdxs}_t := \mathtt{seenIdxs}'\right], \beta}$$

$$\textsc{TxEndRO}_\tau\ \frac{\begin{array}{c} \gamma.\mathtt{status}_t = \mathtt{READY} \qquad \gamma.\mathtt{txn}_\tau = t \qquad \gamma.\mathtt{wrSet}_t = \emptyset \\ \mathtt{tview}' = \mathbf{if}\ \gamma.\mathtt{rdSet}_t \neq \emptyset \wedge \gamma.\mathtt{synctype}_t \in \{\mathsf{A}, \mathsf{RA}\} \wedge \gamma.\mathtt{seenIdxs}_t \neq \emptyset \\ \qquad \mathbf{then}\ \beta.\mathtt{tview}_\tau \otimes view(\gamma.\mathtt{seenIdxs}_t, \gamma.V)\ \mathbf{else}\ \beta.\mathtt{tview}_\tau \end{array}}{lst, \gamma, \beta \rightsquigarrow_\tau \ lst, \gamma \begin{bmatrix} \mathtt{status}_t := \mathtt{COMMITTED}, \\ \mathtt{txview}_\tau := \max(\gamma.\mathtt{seenIdxs}_t) \end{bmatrix}, \beta[\mathtt{tview}_\tau := \mathtt{tview}']}$$

$$\textsc{TxEndWR}_\tau\ \frac{\begin{array}{c} \gamma.\mathtt{status}_t = \mathtt{READY} \qquad \gamma.\mathtt{txn}_\tau = t \qquad \gamma.\mathtt{wrSet}_t \neq \emptyset \qquad i = |\gamma.M| \\ S' = \gamma.\mathtt{synctype}_t \qquad mem' = (last(\gamma.M) \oplus \gamma.\mathtt{wrSet}_\tau) \\ \mathtt{tview}' = \mathbf{if}\ \gamma.\mathtt{rdSet}_t \neq \emptyset \wedge \gamma.\mathtt{synctype}_t \in \{\mathsf{A}, \mathsf{RA}\} \wedge \gamma.\mathtt{seenIdxs}_t \neq \emptyset \\ \qquad \mathbf{then}\ \beta.\mathtt{tview}_\tau \otimes view(\gamma.\mathtt{seenIdxs}_t, \gamma.V)\ \mathbf{else}\ \beta.\mathtt{tview}_\tau \end{array}}{lst, \gamma, \beta \rightsquigarrow_\tau \ lst, \gamma \begin{bmatrix} \mathtt{status}_t := \mathtt{COMMITTED}, M_i := mem' \\ S_i := S', \mathsf{V}_i := \mathtt{tview}' \\ \mathtt{txview}_\tau := \max(\gamma.\mathtt{seenIdxs}_t) \end{bmatrix}, \beta[\mathtt{tview}_\tau := \mathtt{tview}']}$$

$$\textsc{TxAbort}_\tau\ \frac{\gamma.\mathtt{status}_t = \mathtt{READY} \qquad \gamma.\mathtt{txn}_\tau = t \qquad lst' = \lambda r \in Reg.\ \mathbf{if}\ r \in \gamma.\mathtt{regs}_t\ \mathbf{then}\ \bot\ \mathbf{else}\ lst(r)}{lst, \gamma, \beta \rightsquigarrow_\tau \ lst', \gamma \left[\mathtt{status}_t := \mathtt{ABORTED}\right], \beta}$$

Fig. 6. Operational semantics for TMS2-RA

in σ_4 (the state after execution of line 10), we have a new write on d_2 with value 10 and a further new snapshot that synchronises with the snapshot $\{f \mapsto 1\}^{\mathsf{R}}$ causing the thread view of $\tau 2$ and the modification view of the new snapshot to be updated to the last writes of d_1 and d_2. Next, in σ_5, when $\tau 3$ executes its transaction, this transaction is guaranteed to synchronise with $\{f \mapsto 2\}^{\mathsf{RA}}$, causing $\tau 3$'s view to be updated to the latest writes of d_1 and d_2, which is inherited from the modification view of $\{f \mapsto 2\}^{\mathsf{RA}}$.

3.4 Modular Operational Semantics

To reason about clients that use abstract TMS2-RA transactions in a modular fashion, we use configurations that are triples (lst, γ, β), where $lst : Reg \to Val$ denotes the local register state, γ is

Init: glb := 0

TxBegin(*regSet*)
*B*1: regs := *regSet*;
*B*2: hasRead := *false*;
*B*3: **do** loc $\leftarrow^{A}$ glb
*B*4: **until** *even*(loc)

TxWrite(x, v)
*W*1: **if** *even*(loc) **then**
*W*2: $r_1 \leftarrow$ **CAS**^{RA}(glb, loc, loc + 1);
*W*3: **if** $\neg r_1$ **then**
*W*4: $\forall s \in$ regs. $s := \bot$; **return**; // ABORT
*W*5: **else** loc := loc + 1;
*W*6: $x :=^{R} v$; // WRITE OK

TxEnd
*E*1: **if** *odd*(loc) **then**
*E*2: glb $:=^{R}$ loc + 1;

TMRead(x, r)
*R*1: **if** $r \in$ regs **then**
*R*2: $r \leftarrow^{A} x$;
*R*3: **if** $\neg$hasRead $\wedge$ *even*(loc) **then**
*R*4: $r_1 \leftarrow$ **CAS**^{RA}(glb, loc, loc)
*R*5: **if** r_1 **then**
*R*6: hasRead := *true*;
*R*7: **return**; // READ OK
*R*8: **else**
*R*9: $r_1 \leftarrow$ glb;
*R*10: **if** r_1 = loc **then**
*R*11: **return**; // READ OK
*R*12: $\forall s \in$ regs. $s := \bot$; // ABORT

Fig. 7. TML-RA: A release-acquire transactional mutex lock. For simplicity, the thread id is omitted

the TMS2-RA state (which includes all transactional variables described in Fig. 4) and β is the C11 state of the client (see [Dalvandi et al. 2020a; Dalvandi and Dongol 2022b; Dalvandi et al. 2022]).

The transition relation for transactional operations is given in Fig. 6. These follow the automata-style description given in Fig. 4, but make state components that are affected by each transition more precise. The most interesting aspect of these rules is the interaction between a releasing writing transaction and subsequent committing reading transaction.

Note that each releasing writing transaction sets S_i (where i is last index in M at the time of writing) to either R or RA. Additionally, the view of the thread at the time of writing is recorded in V_i. A later transaction with an acquiring annotation calculates a new view using the function *view* as defined in Fig. 4 and updates, among other components, the executing thread's view in β. This means that, as expected, if there is a release-acquire synchronisation through a transactional memory library, then the client's view will be updated to match the synchronisation that occurs.

4 A C11 STM IMPLEMENTATION

In this section, we develop a release-acquire version of a transactional mutex lock, that we call TML-RA, based on an SC implementation by Dalessandro et al [Dalessandro et al. 2010]. Our algorithm is provided in Fig. 7, where the highlights indicate the fragments of code that we have introduced or modified. The grey highlights represent code additional to Dalessandro et al's original implementation, and the blue highlights represent the necessary release-acquire synchronisation.

We first discuss the core features of TML (§4.1), then discuss the extensions introduced in TML-RA to optimise for C11 release-acquire synchronisation (§4.2). We present the benchmarking results for both algorithms in §4.3. In §6, we present a proof that TML-RA implements TMS2-RA, i.e., any observation a client program makes when it uses TML-RA is a possible observation when it uses TMS2-RA.

4.1 TML

TML is synchronised using a single global counter `glb`, initialised to 0, where `glb` is even iff no writing transaction is currently executing.

A transaction begins by taking a snapshot of `glb` in local variable `loc` and only begins if the value read is even.

A write operation checks that `loc` is even and if so, it attempts to increment `glb` using the **CAS** at line $W2$. If this **CAS** succeeds, it increments `loc` (line $W5$), then immediately updates the location x (line $W6$). If the **CAS** fails, the transaction aborts. Note that if `loc` is odd then the current transaction "owns" the lock, meaning that lines $W2$-$W5$ can be bypassed.

A read operation (ignoring lines $R3$-$R7$ for now) reads the given location into the given register r (line $R2$). At lines $R9$ and $R10$ it checks that `glb` is consistent with `loc`. If so, the read succeeds, otherwise, the transaction aborts.

A transaction ends by checking whether the current transaction is a writing transaction. This can be determined by checking whether `loc` is odd since a writing transaction must have incremented `glb` via the **CAS** at line $W2$ and `loc` via the write at line $W5$ making both their values odd. Therefore, a writing transaction must increment `glb` to make it even again.

4.2 TML-RA

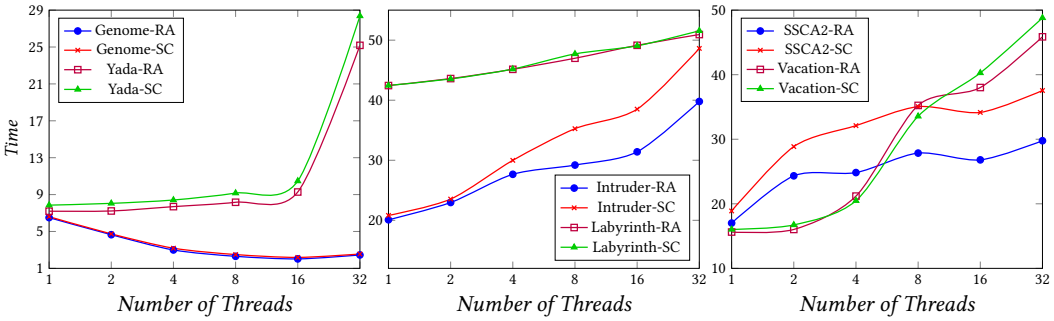
We now describe the necessary modifications to TML and the synchronisation induced by TMS2-RA. We assume that transactions in TML-RA are all release-acquiring and hence we omit the transaction annotation in `TxBegin`.

We assume all accesses to shared variables are either relaxed (e.g., the read at line $R9$), releasing (e.g., the write at line $E2$), acquiring (e.g., the read at line $B2$) or release-acquiring (e.g., the **CAS** at line $R4$). Additionally, we introduce a new local variable `hasRead`, initially set to *false* and a code path $R3$-$R7$, which is followed if a transaction performs a read without having previously performed a read or a write. We explain the purpose of this code path in more detail below.

Transaction synchronisation. Recall that TMS2-RA requires that transactions are consistent w.r.t. a single memory snapshot and that external reads of a transaction synchronise with *some* memory snapshot. This may not occur in a relaxed memory context without adequate synchronisation. In particular, a writing transaction must perform a releasing write to `glb` at line $E2$ so that if a later transaction reads from this write, it synchronises with *all* of the writes performed by the writing transaction. To ensure this, we require the read of `glb` at line $B3$ as well as the **CAS** operations at lines $W2$ and $R4$ to be acquiring. Note that this also guarantees release-acquire *client synchronisation*.

The second key synchronisation is between $W6$ performed by a writing transaction t_w and $R2$ performed by a (different) reading transaction t_r. Suppose that both t_w and t_r are live. If t_r happens to read the write written at $W6$, it must now abort because t_r's snapshot of `glb` will be inconsistent with the latest value of `glb` installed by the t_w. The release-acquire synchronisation between $W6$ and $R2$ ensures that this will happen, i.e., t_r will see the new `glb` written by t_w, causing the test at $R10$ to fail and t_r to abort.

Causal linearizability. The design of TML-RA ensures that all transactions, including read-only transactions are *causally linearizable* [Doherty et al. 2018], which is a condition that additionally guarantees *compositionality* (or *locality* [Herlihy and Wing 1990; Sela et al. 2021]) of concurrent objects. This notion of compositionality is that of Herlihy and Wing [Herlihy and Wing 1990]. In particular, under SC memory, given a history comprising several concurrent objects, if the history restricted to each object is linearizable, then the history as a whole is linearizable. In a relaxed memory setting, Doherty et al [Doherty et al. 2018] have shown that linearizability alone is



(a) Genome and Yada (b) Intruder and Labyrinth (c) SSCA2 and Vacation

Fig. 8. Results of STAMP benchmarks for TML-RA and TML-SC

insufficient to guarantee compositionality, and it is necessary to induce a "happens-before" relation when a specification induces a particular linearization.

The happens-before required by causal linearizability is naturally achieved for writing transactions via the **CAS** at line $W2$. For a read-only transaction, we introduce the **CAS** at line $R4$, which installs a new write to `glb` without changing its value. All transactions that follow the **CAS** at line $R4$ will be *causally ordered* after the reading transaction. Such a **CAS** must only be performed once, thus we introduce a local variable `hasRead`, which is set to true if the **CAS** succeeds so that later reads from the same transaction can avoid the code path from $R3$-$R7$.

Note that the conditions necessary to guarantee causal linearizability (and hence compositionality) could have been introduced at the level of TMS2-RA. However, there are questions about whether the notion of compositionality introduced by Herlihy and Wing [Herlihy and Wing 1990] are appropriate in a relaxed memory context [Raad et al. 2019a]. Therefore we leave out the causal linearizability conditions in TMS2-RA to avoid over-constraining the specification.

4.3 Benchmarking

We implemented two versions of the TML algorithm: TML-RA (see Fig. 7) and TML-SC (the SC counterpart [Dalessandro et al. 2010]) and benchmarked both using the STAMP benchmarking suite [Minh et al. 2008]. Each experiment was repeated 20 times to rule out external loads on the test machine and an average of these times was taken. The results of the six benchmarks that we ran with STAMP are presented in Fig. 8. TML-RA is equivalent to or outperforms TML-SC in almost all cases, with a maximum improvement of 20%. On average, TML-RA performs 8.2% better than TML-SC.

Unsurprisingly, since TML optimises read-heavy workloads, its performance degrades under high write contention, and this is consistent with prior results [Dalessandro et al. 2010]. However, it is interesting that the degradation of TML-RA is not as severe as TML-SC for the Intruder and SSCA2 benchmarks.

TML-RA theoretically allows more parallelism than TML-SC since a read-only transaction t_r is not forced to abort if a writing transaction t_w executes after t_r's first read operation - t_r must only aborts if it sees t_w's *glb* update, or one of t_w's writes. Both Intruder and SSCA2 have a large number of short transactions; SSCA2 additionally has small read/write sets [Minh et al. 2008]. Here, TML-RA may be able to exploit the theoretical parallelism. In the single-threaded case, TML-RA executes far fewer heavyweight CASs.

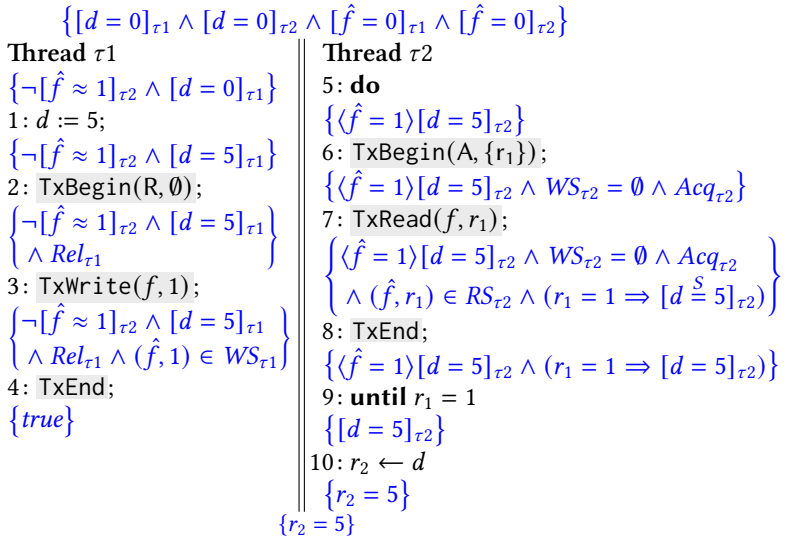


Fig. 9. Proof outline for transactional MP from Fig. 2a

As with prior results, we see that for the read-heavy benchmark Genome, the performance of both TML-RA and TML-SC improves as the number of threads increases.

5 TARO: A LOGIC FOR RELEASE-ACQUIRE TM

The development of view-based operational semantics for various fragments of C11 [Dalvandi et al. 2020a; Kaiser et al. 2017; Kang et al. 2017] has provided foundations for several logics for reasoning about C11 programs. These include separation logics [Kaiser et al. 2017; Svendsen et al. 2018] and extensions to Owicki-Gries reasoning [Dalvandi et al. 2020a, 2022; Lahav and Vafeiadis 2015; Wright et al. 2021]. Our point of departure is the Owicki-Gries encoding for RC11 RAR [Dalvandi et al. 2020a], which is the fragment of C11 that we focus on in this paper.[5]

A key benefit of the logic in [Dalvandi et al. 2020a] is that it enables reuse of *standard* Owicki-Gries proof decomposition rules and straightforward mechanisation in Isabelle/HOL [Dalvandi et al. 2020b, 2022]. As we shall see, we maintain these benefits in the context of C11 with release-acquire transactions. Our reasoning framework, called TARO, like Dalvandi et al [Dalvandi et al. 2020a; Dalvandi and Dongol 2021] uses view-based assertions to abstractly describe the system state, allowing reasoning about the *current view* of a thread, and *view transfer* from one thread to another through release-acquire synchronisation. TARO introduces additional assertions to enable reasoning about transactional views.

5.1 View-Based Assertions

In this section, we discuss the assertions and proof rules of TARO abstractly. The proof rules can be used to reason syntactically about a program without having to understand the low-level operational semantics of the C11 model. Our operational semantics is an extension of prior works [Dalvandi et al. 2020a; Kaiser et al. 2017; Kang et al. 2017] that include an encoding of TMS2-RA.

[5]These frameworks are based on models that assume top-level parallelism only. Therefore, our framework similarly re assumes top-level paralellism. This model can be extended to support dynamic parallelism, but such extensions are uninteresting for the purposes of this paper.

To motivate TARO, consider the proof outline in Fig. 9 for the transactional message passing program from Fig. 2a. We use '$\hat{}$' to distinguish transactional locations in a proof. For the program in Fig. 9, we have a transactional location $\hat{f}$.

5.1.1 View assertions. The proof outline contains three assertions from [Dalvandi et al. 2020a] describing the *views* that each thread may have of the system state. Recall (§3.1), that we can define the set of values that a thread can see in each state using the function OV.

- A *definite value* assertion, denoted $[x = v]_\tau$, holds iff thread τ sees the last write to location x and this write has value v. Thus, $[x = v]_\tau \Rightarrow OV_\tau(x) = \{v\}$.
- A *possible value* assertion, denoted $[x \approx v]_\tau$, which holds iff when τ can see a write to x with value v. $[x \approx v]_\tau$ is shorthand for $v \in OV_\tau(x)$.
- A *conditional value* assertion, denoted $\langle y = u\rangle[x = v]_\tau$, which holds iff an acquiring read of y by τ that returns the value u is guaranteed to induce a release-acquire synchronisation so that $[x = v]_\tau$ holds after this read.

Example 2. Consider the third state, i.e., σ_2 in Fig. 3. There, we have $[d = 5]_{\tau 1} \wedge [f = 1]_{\tau 1}$ as well as $[f \approx 0]_{\tau 2} \wedge [f \approx 1]_{\tau 2}$. Moreover, we have $\langle f = 1\rangle[d = 5]_{\tau 2}$.

We ask the interested reader to consult [Dalvandi et al. 2020a, 2022] for further details of these assertions.

5.1.2 Transactional assertions. As alluded to above, TARO introduces several new assertions to describe the transactional state. These assertions are, in general, local to the transaction being executed, and hence, stable under the execution of other threads. Fig. 9 contains the following transaction local assertions:

- Rel_τ, which holds iff τ is executing a releasing or release-acquiring transaction.
- Acq_τ, which holds iff τ is executing an acquiring or release-acquiring transaction.
- $(\hat{x}, v) \in WS_\tau$ (and $(\hat{x}, v) \in RS_\tau$), which holds iff τ is executing a transaction whose write set (resp. read set) contains a write to (resp. read of) $\hat{x}$ with value v.
- $[x \overset{S}{=} v]_\tau$, which holds iff τ is executing a transaction such that committing this transaction results in the definite value assertion $[x = v]_\tau$ (see above).

In addition, we include a number of assertions that can be used to verify client programs that use TMS2-RA (See §5.1). The assertion language presented here is heavily inspired by the view-based assertion language presented in [Dalvandi et al. 2020a].

A memory i is visible to a transaction executed by a thread τ iff i is greater than the transaction thread view of τ ($\texttt{txview}_\tau$) and is less than the maximum index of the memory ($|M| - 1$). We define the set of visible memories OM_t to be:

$$OM_\tau = \{n \mid n \geq \texttt{txview}_\tau \wedge n \leq |M| - 1\}$$

- A *transactional definite observation* assertion, denoted $[\hat{x} = v]_t$, holds iff for all memory versions i, where i is greater than or equal to $\texttt{beginIdx}_t$, the value of $M_i(x)$ is v. Formally, for a transactional state γ:

$$[\hat{x} = v]_\tau(\gamma) \;\hat{=}\; \forall i \in \gamma.OM_\tau.\ \gamma.M_i(\hat{x}) = v$$

  These are lifted to client-object states (γ, β) in the normal manner, e.g., $[\hat{x} = v]_\tau(\gamma, \beta) = [\hat{x} = v]_\tau(\gamma)$
- A *transactional possible observation* assertion, denoted $[\hat{x} \approx v]_\tau$, holds iff there exists a memory version i that has value v for $\hat{x}$. Formally:

$$[\hat{x} \approx v]_\tau(\gamma) \;\hat{=}\; \exists i \in \gamma.OM_\tau.\ \gamma.M_i(\hat{x}) = v$$

- A *transactional conditional observation* assertion, denoted $\langle \hat{y} = u \rangle [x = v]_\tau$, holds iff an acquiring transactional read of y by τ that returns a value u is guaranteed to induce a release-acquire synchronisation so that $[x \overset{S}{=} v]_\tau$ holds in the client state after the reading transaction successfully commits. Formally:

$$\langle \hat{y} = u \rangle [x = v]_\tau(\gamma, \beta) \;\hat{=}\; \forall i \in \gamma.OM_\tau.\ \gamma.M_i(\hat{y}) = u \Rightarrow \gamma.V_i(x) = \beta.last(x) \wedge \mathtt{val}(\beta.last(x)) = v \wedge \gamma.S_i$$

  where $last(x)$ is the last write to x in the modification order. It is important to note that this assertion is over two states: transaction state γ and client state β, explaining transfer of information across two threads using the transactional memory. In particular, the transactional view V_i for the memory index i must see the last write to x in the client state β. This means that the thread that committed the transaction writing the value u to $\hat{y}$ did so when it saw the last write to x.

Example 3. Returning to our transactional MP example (Fig. 9), the precondition of line 1 contains assertions $\neg[\hat{f} \approx 1]_{\tau 2}$ and $[d = 0]_{\tau 1}$, which ensure that, prior to executing line 1, thread $\tau 2$ *cannot* see the value 1 for $\hat{f}$ and thread $\tau 1$ *must* see the value 0 for d, respectively. In the postcondition of line 2, $[d = 0]_{\tau 1}$ changes to $[d = 5]_{\tau 1}$ since $\tau 1$ performs a write to d with value 5. The other view-based assertions in $\tau 1$ are similar. We explain the transactional assertions involving *Rel* and *WS* below.

Now consider the assertions in thread $\tau 2$. The precondition of line 6 (which is also the precondition of line 5) contains a conditional value assertion $\langle \hat{f} = 1 \rangle [d = 5]_{\tau 2}$. This assertion ensures that, if $\tau 2$ reads the value 1 for $\hat{f}$ via an acquiring (or release-acquiring) transaction, and this transaction successfuly commits, then its view is guaranteed to be updated so that $[d = 5]_{\tau 2}$ holds. In a transactional setting, we establish this fact in three steps.

(1) After executing line 7, we use $\langle \hat{f} = 1 \rangle [d = 5]_{\tau 2}$ to establish that $r_1 = 1 \Rightarrow [d \overset{S}{=} 5]_{\tau 2}$ holds. Note that r_2 stores the value 1 returned by a transactional read of $\hat{f}$. Thus, $\langle \hat{f} = 1 \rangle [d = 5]_{\tau 2}$ is transformed into an implication after the execution of line 7. The assertion $[d \overset{S}{=} 5]_{\tau 2}$ is a new assertion introduced in TARO, which states that if the transaction executed by $\tau 2$ commits, then $[d = 5]_{\tau 2}$ holds in the post-state.
(2) If the transaction sucessfully commits (line 8), we use $r_1 = 1 \Rightarrow [d \overset{S}{=} 5]_{\tau 2}$ to establish $r_1 = 1 \Rightarrow [d = 5]_{\tau 2}$ in the postcondition. Recall that all registers used by a transaction are set to $\perp$ when a transaction aborts, so if $\tau 2$ reaches line 9 by aborting the transaction, then this assertion is trivially true.
(3) We use $r_1 = 1 \Rightarrow [d = 5]_{\tau 2}$ to establish $[d = 5]_{\tau 2}$ after the **do-until** loop, using the guard $r_1 = 1$ at line 9.

Finally, we use $[d = 5]_{\tau 2}$ in the precondition of line 10 to establish the postcondition $r_2 = 5$. This is because $[d = 5]_{\tau 2}$ guarantees that the only value $\tau 2$ can read for d is 5.

5.2 TARO: Transactional Owicki-Gries Reasoning

Now that we have introduced the assertions used by TARO, we now review the Owicki-Gries proof obligations. As discussed above, the use of view-based assertions allows us to use the standard Owicki-Gries theory. Regardless, we review the theory in the context of our language, which supports (abstract) TM operations. Formally, we model programs as a labelled transition system, given by the syntax in Fig. 10.

A command (of type *ACom*) is either a local assignment $r := Exp$, a *store* to a shared location $x :=^{[\mathrm{R}]} Exp$, a *load* from a shared location $r \leftarrow^{[\mathrm{A}]} x$, a compare-and-swap $r \leftarrow \mathbf{CAS}^{[Sync]}(x, u, v)$,

$$u, v \in Val \triangleq \mathbb{N} \qquad x, y, \ldots \in Loc \qquad r, r_1, r_2 \ldots \in Reg \qquad \tau, \tau 1, \tau 2, \ldots \in TId \triangleq \mathbb{N} \qquad i, j, k, \ldots \in Label$$
$$e \in Exp ::= v \mid r \mid e{+}e \mid \cdots$$
$$B \in BExp ::= true \mid B \wedge B \mid \cdots$$
$$Sync ::= \mathsf{RX} \mid \mathsf{R} \mid \mathsf{A}$$
$$\alpha \in ACom ::= r := Exp \mid x :=^{[\mathsf{R}]} Exp \mid r \leftarrow^{[\mathsf{A}]} x \mid r \leftarrow \mathbf{CAS}^{[Sync]}(x, u, v) \mid \mathtt{TxBegin}(Sync, 2^{Reg}) \mid \mathtt{TxRead}(x, r) \mid \mathtt{TxWrite}(x, v) \mid \mathtt{TxEnd}$$
$$ls \in LCom ::= \alpha \ \mathbf{goto}\ j \mid \mathbf{if}\ B\ \mathbf{goto}\ j\ \mathbf{elseto}\ k$$
$$\Pi \in Prog \triangleq TId \times Label \to LCom$$

Fig. 10. Language syntax

or a transactional operation. The annotations RX, R and A are optional, as indicated by the brackets '[' and ']'. Thus, for example, both $x := e$ and $x :=^{\mathsf{R}} e$ are valid load commands; the former is relaxed and the latter is releasing. A **CAS** may be annotated to be relaxed, or release and/or acquire. Note that a **CAS** returns a boolean to indicate whether or not the compare-and-swap has been successful. $\mathtt{TxBegin}([Sync], 2^{Reg})$, $\mathtt{TxRead}(x, r)$, $\mathtt{TxWrite}(x, v)$ and $\mathtt{TxEnd}$ are transactional operations, as defined by the TMS2-RA automata in Fig. 4.

We use a program counter variable $pc : TId \to Label$ to model control flow, and model a program Π as a function mapping each pair (τ, i) of thread identifier and label to the *labelled statement* (in *LCom*) to be executed. A labelled statement may be (i) a plain statement of the form α **goto** j, comprising an atomic statement α to be executed and the label j of the next statement; or (ii) a conditional statement of the form **if** B **goto** j **elseto** k to accommodate branching, which proceeds to label j if B holds and to k, otherwise. We assume a designated label, $\iota \in Label$, representing the *initial label*; i.e., each thread begins execution with $pc(\tau) = \iota$. Similarly, $\zeta \in Label$ represents the *final label*.

We let *Assertion* be the set of *assertions* that use view-based expressions. We model program annotations via an *annotation function*, $ann \in Ann = TId \times Label \to Assertion$, associating each program point (τ, i) with its associated assertion. A *proof outline* is a tuple (in, ann, fin), where $in, fin \in Assertion$ are the initial and final assertions.

Definition 1 (Validity)**.** A proof outline (in, ann, fin) is *valid* for a program Π iff each of the following holds:

Initialisation For all $\tau \in TId$, $in \Rightarrow ann(\tau, \iota)$.
Finalisation $(\forall \tau \in TId.\ ann(\tau, \zeta)) \Rightarrow fin$
Local correctness For all $\tau \in TId$ and $i \in Label$, either:
- $\Pi(\tau, i) = \alpha$ **goto** j and $\{ann(\tau, i)\}\ \alpha\ \{ann(\tau, j)\}$; or
- $\Pi(\tau, i) =$ **if** B **goto** j **elseto** k and both $ann(\tau, i) \wedge B \Rightarrow ann(\tau, j)$ and $ann(\tau, i) \wedge \neg B \Rightarrow ann(\tau, k)$ hold.

Stability For all $\tau_1, \tau_2 \in TId$ such that $\tau_1 \neq \tau_2$ and $i_1, i_2 \in Label$ if $\Pi(\tau_1, i_1) = \alpha$ **goto** j, then $\{ann(\tau_2, i_2) \wedge ann(\tau_1, i_1)\}\ \alpha\ \{ann(\tau_2, i_2)\}$

Intuitively, Initialisation (resp. Finalisation) ensures that the initial (resp. final) assertion of each thread holds at the beginning (resp. end); Local correctness establishes validity for each thread; and Stability ensures that each (local) thread annotation is *interference-free* under the execution of other threads [Owicki and Gries 1976].

To support Owicki-Gries reasoning, we have proved a number of high-level rules, extending those of Dalvandi et al. [2020a]; Dalvandi and Dongol [2021] to cope with transactional assertions from §5.1 and the transactional commands. For instance, the following rules are used in the proof

of transactional message passing. A number of other rules are provided as part of our Isabelle/HOL development.

Lemma 1. *Suppose $\tau 1 \neq \tau 2$. Then each of the following holds:*

$\{true\}\ \texttt{TxWrite}_\tau(x,v)\ \{(\hat{x},v) \in WS_\tau\}$

$\{(x,u) \in WS_{\tau 1} \wedge Rel_{\tau 1} \wedge [\hat{x} \not\approx u]_{\tau 2} \wedge [y = v]_{\tau 1}\}\ \texttt{TxEnd}_{\tau 2}\ \{\langle \hat{x} = u \rangle [y = v]_{\tau 2}\}$

$\{(x,\_) \notin WS_\tau \wedge Acq_\tau \wedge \langle \hat{x} = u \rangle [y = v]_\tau\}\ \texttt{TxRead}_\tau(x,r)\ \left\{(\hat{x},r) \in RS_\tau \wedge (r = u \implies [y \overset{S}{=} v]_\tau)\right\}$

$\left\{(\hat{x},r) \in RS_\tau \wedge (r = u \implies [y \overset{S}{=} v]_\tau)\right\}\ \texttt{TxEnd}_\tau\ \{r = u \implies [y = m]_\tau\}$

The rules in Lemma 1 have been verified in Isabelle/HOL w.r.t. the operational semantics. Once proved, they can be used to show validity of proof outlines such as those in Fig. 9 without having to consult the operational semantics.

Theorem 1. *The proof outline in Fig. 9 is valid.*

This theorem has been verified in Isabelle/HOL, and it makes extensive use of generic proof rules such as the ones proved in Lemma 1. In particular, given such lemmas, like in previous works [Bila et al. 2022; Dalvandi et al. 2020a; Dalvandi and Dongol 2021; Dalvandi et al. 2022], Isabelle/HOL is automatically able to find and apply the appropriate proof rule using the built-in `sledgehammer` tool [Böhme and Nipkow 2010]. This automation has been key to scaling mechanised verification of proof outlines in view-based logics. For example, the proofs of TML-RA (see §6) requires verification of complex invariants and proof outlines, and these proofs make use of the proof rules developed in prior work [Dalvandi et al. 2020a]. Similarly, TARO can be applied to verify more complex programs that use transactions, for instance if one were to develop transactional data structures. Interestingly, because transactions provide isolation guarantees, many of the proofs are simplified since the stability checks for in-flight transactions become trivial.

The proof outlines for the programs in Figs. 2b and 2c are provided in [Dalvandi and Dongol 2022b].

6 PROVING CORRECTNESS OF TML-RA

We now turn to the question of correctness of TML-RA with respect to the TMS2-RA specification.

6.1 Refinement and Simulation for Weak Memory

Since we have an operational semantics with an interleaving semantics over weak memory states, the development of our refinement theory closely follows the standard approach under SC [de Roever and Engelhardt 1998]. Suppose P is a program with initialisation **Init**. An *execution* of P is defined by a possibly infinite sequence $\Delta_0\ \Delta_1\ \Delta_2\ \ldots$ such that

(1) each Δ_i is a 4-tuple $(P_i, ls_i, \gamma_i, \beta_i)$ comprising a program to be executed, local state, global library state and global client state, and
(2) $(P_0, ls_0, \gamma_0, \beta_0) = (P, ls_{\mathbf{Init}}, \gamma_{\mathbf{Init}}, \beta_{\mathbf{Init}})$, and
(3) for each i, we have $\Delta_i \Longrightarrow \Delta_{i+1}$, where $\Longrightarrow$ is the transition relation of the program (as defined by the operational semantics).

Let $LVar_P$ be the set of local variables corresponding to a program P. If P is a client, a *client trace* corresponding to an execution $\Delta_0\ \Delta_1\ \Delta_2\ \ldots$ is a sequence $ct \in \Sigma_P^*$ such that $ct_i = (\pi_2(\Delta_i)_{|P}, \pi_4(\Delta_i))$, where π_n is a projection function that extracts the nth component of a given tuple and $ls_{|P}$ restricts the given local state ls to the variables in $LVar_P$. Thus, each ct_i is the global client state component of Δ_i. After such a projection, the concrete implementation may contain (finite or infinite)

stuttering [de Roever and Engelhardt 1998], i.e., consecutive states in which the client state is unchanged. We let $rem\_stut(ct)$ be the function that removes all stuttering from the trace ct, i.e., each consecutively repeating state is replaced by a single instance of that state. We let $Tr_{SF}(P)$ denote the set of *stutter-free traces* of a program P, i.e., the stutter-free traces generated from the set of all executions of P.

Below we refer to the client that uses the abstract specification as the *abstract client* and the client that uses the implementation as the *concrete client*. The notion of contextual refinement that we develop ensures that a client is not able to distinguish the use of a concrete implementation in place of an abstract specification. In other words, each thread of the concrete client should only be able to observe the writes (and updates) in the client state (i.e., γ component) that the thread could already observe in a corresponding of the client state of the abstract client. First we define trace refinement for weak memory states.

Definition 2 (State and Trace Refinement)**.** We say a concrete client state (ls, β_C) is a *refinement* of an abstract client state (als, β_A), denoted $(ls, \beta_C) \leq (als, \beta_A)$ iff $ls = als$ and for all threads τ and $x \in GVar$, we have $\beta_C.OW_\tau(x) \subseteq \beta_A.OW_\tau(x)$. We say a concrete client trace ct is a *refinement* of an abstract client trace at, denoted $ct \leq at$, iff $ct_i \leq at_i$ for all i.

This now leads to a natural trace-based definition of contextual refinement.

Definition 3 (Program Refinement)**.** A concrete program P_C is a *refinement* of an abstract program P_A, denoted $P_C \leq P_A$, iff for any (stutter-free) client trace $ct \in Tr_{SF}(P_C)$ there exists a (stutter-free) client trace $at \in Tr_{SF}(P_A)$ such that $ct \leq at$.

Finally, we obtain a notion of contextual refinement for abstract objects. We let $P[O]$ be the client program calling operations from object O. Note that O may be an abstract object, in which case execution of each method call follows the abstract object semantics, or a concrete implementation.

Definition 4 (Contextual refinement)**.** We say a concrete object CO is a *contextual refinement* of an abstract object AO iff for any client program P, we have $P[CO] \leq P[AO]$.

Here, we use a *simulation-based* proof method, which is a standard technique from the literature that establishes refinement between TMS2-RA and TML-RA. The difference in a relaxed memory setting is that the refinement relation is between more complex configurations of the form (ls, γ, β), where ls describes the local state, γ is the client state and β is a state of the TM in question. In particular, a *simulation relation*, R, relates triples $\Gamma_A \hat{=} (als, \gamma_A, \beta_A)$ of the abstract system with triples $\Gamma_C \hat{=} (ls, \gamma_C, \beta_C)$ of the concrete system.

The definition below assumes a reflexive relation $\gamma_C \xrightarrow{\texttt{tview}_\tau} \gamma'_C$ for each thread τ that arbitrarily advances the thread view of τ (for one or more locations).

Definition 5 (Forward simulation)**.** For an abstract object AO and a concrete object CO, for a client program P, we say $R(\Gamma_A, \Gamma_C) \hat{=} R_V((als, \beta_A), (ls, \beta_C)) \wedge R_O((als_{|AO}, \gamma_A), (ls_{|CO}, \gamma_C))$ is a forward simulation between A and C iff each of the following holds:

Client observation.

$R_V((als, \beta_A), (ls, \beta_C)) = als_{|P} = ls_{|P} \wedge (\forall \tau \in TId, x \in Loc.\ \ \beta_A.\texttt{tview}(t, x) \leq \beta_C.\texttt{tview}(t, x))$

Thread view stability. For any thread τ,

$R_O((als_{|AO}, \gamma_A), (ls_{|CO}, \gamma_C)) \wedge (\gamma_C \xrightarrow{\texttt{tview}_\tau} \gamma'_C) \Rightarrow R_O((als_{|AO}, \gamma_A), (ls_{|CO}, \gamma'_C))$

Initialisation. For any concrete initial state Γ^0_C, there exists an abstract initial state Γ^0_A such that $R(\Gamma^0_A, \Gamma^0_C)$.

Preservation. For any concrete states Γ_C, Γ'_C such that C can take an atomic transition from Γ_C to Γ'_C, if Γ_A is an abstract state such that $R(\Gamma_A, \Gamma_C)$, then either

- $R(\Gamma_A, \Gamma'_C)$, or (stuttering step)
- there exists a transition of A from Γ_A to some state Γ'_A such that $R(\Gamma'_A, \Gamma'_C)$. (non-stuttering step)

Initialisation and *preservation* are standard components of a forward simulation. *Client observation* is necessary in a relaxed memory context to ensure that the client-side observations of the concrete system are possible observations of the abstract system. In particular, if an abstract object specifies a particular client-side synchronisation, then this synchronisation must also be present in the concrete implementation (see [Dalvandi and Dongol 2021]). *Thread view stability* guarantees that the R_O component of the refinement relation is preserved when the thread view in the library is shifted forward, e.g., due to synchronisation within a client.

Note that Definition 5 only guarantees preservation of safety. To additionally preserve liveness, further progress guarantees are required in an implementation [Dongol and Groves 2016; Gotsman and Yang 2011]. We leave liveness preservation through refinement for future work since notions of fairness and progress of weak memory models is still at the early stages [Lahav et al. 2021].

Theorem 2. *If R is a forward simulation between AO and CO, then for any client P we have $P[CO] \leq P[AO]$.*

6.2 Forward Simulation for TML-RA

Perhaps the most technically challenging aspect of this paper is the proof of Theorem 3 below, which ensures the correctness of TML-RA w.r.t. TMS2-RA.

This section describes the simulation relation used to prove refinement between TML-RA and TMS2-RA. Validity of the forward simulation itself has been verified using Isabelle/HOL. The refinement relation

$$R((als, \gamma_A, \beta_A), (ls, \gamma_C, \beta_C)) \;\hat{=}\; R_V((als, \beta_A), (ls, \beta_C)) \wedge (1) \wedge (\forall t.\ (2) \wedge (3) \wedge (4) \wedge (5) \wedge (6) \wedge (7) \wedge (8) \wedge (9))$$

The first conjunct (1) in the refinement relation R states that the value of the last write to glb divided by 2 ($\mathtt{wc}(n) \mathrel{\hat{=}} n \div 2$) is equal to the last version of history written to M.

$$\mathtt{wc}(\gamma_C.\mathtt{lastval}(glb)) = |\gamma_A.M| \tag{1}$$

The next conjunct, (2), states that the last value written to any location l in γ_C is either the value of l in the last abstract memory index or in the write set of the executing transaction

$$\forall l.\ l \neq glb \Rightarrow \gamma_C.\mathtt{lastval}(l) \in \{\gamma_A.M_{|\gamma_A.M|}(l), \gamma_A.\mathtt{wrSet}_t(l)\} \tag{2}$$

where $\mathtt{lastval}(x)$ is a function that returns the value of the last write written to a location x.

The next conjunct (3) is an in-flight simulation relation (i.e. the transaction has begun and is not committed or aborted) and states that for all threads τ if transaction t ($\gamma_A.\mathtt{txn}_\tau = t$) is in-flight, the value of $\mathtt{wc}(\gamma_C.loc_\tau)$ will be greater than or equal to $\mathtt{beginIdx}_t$ and the read set of t will be consistent with memory verison $\mathtt{wc}(\gamma_C.loc_\tau)$:

$$\gamma_A.\mathtt{beginIdx}_t \leq \mathtt{wc}(ls.loc_t) \wedge \gamma_A.\mathtt{rdSet}_t \subseteq \gamma_A.M_{\mathtt{wc}(ls.loc_t)} \tag{3}$$

Conjunct (4) states that if the value of $ls.loc_t$ is even then write set of γ_A must be empty:

$$\mathtt{even}(ls.loc_t) \Rightarrow \gamma_A.\mathtt{wrSet}_t = \emptyset \tag{4}$$

Also if a transaction t that has already written to a location then the write set of γ_A is not empty:

$$ls.\mathtt{hasWritten}_t \Rightarrow \gamma_A.\mathtt{wrSet}_t \neq \emptyset \tag{5}$$

If a transaction has not read any location yet in the concrete state, then the read set of the abstract state should be empty:

$$\neg ls.\mathtt{hasRead}_t \Rightarrow \gamma_A.\mathtt{rdSet}_t = \emptyset \tag{6}$$

If there is a write in the write set of the abstract state, then the value should match the value of the last write written to that location by the concrete implementation:

$$\forall l \in \mathbf{dom}(\gamma_A.\mathtt{wrSet}_t).\ \gamma_A.\mathtt{wrSet}_t(l) = \gamma_C.\mathtt{lastval}(l) \tag{7}$$

The value of a visible write of thread τ to variable *glb* divided by two is a visible memory by thread τ of the abstract state:

$$\forall w \in \gamma_C.OW_\tau(glb).\ \mathtt{wc}(\mathtt{val}(w)) \in \gamma_A.vmems_\tau \tag{8}$$

All seen memory indices by the abstract transaction t are less the value of thread view of *glb* for thread τ divided by 2:

$$\forall i \in \gamma_A.\mathtt{seenIdxs}_t.\ i \leq \mathtt{wc}(\mathtt{val}(\gamma_C.\mathtt{tview}_\tau(glb))) \tag{9}$$

Theorem 3. *R is a forward simulation between TMS2-RA and TML-RA.*

Proof. This theorem has been verified in Isabelle/HOL (see [Dalvandi and Dongol 2022a]). □

7 RELATED WORK

Verifying C11 programs. There are now several different approaches to program verification that support different aspects of the C11 relaxed memory model using pen-and-paper proofs (e.g., [Alglave and Cousot 2017; Doko and Vafeiadis 2017; Lahav and Vafeiadis 2015; Turon et al. 2014]), model checking (e.g., [Abdulla et al. 2019; Kokologiannakis et al. 2019]), specialised tools (e.g., [Krishna et al. 2020; Summers and Müller 2018; Svendsen et al. 2018; Tassarotti et al. 2015]), and generalist theorem provers (e.g., [Dalvandi et al. 2020a]). These cover a variety of (fragments of) memory models and proceed via exhaustive state space exploration, separation logics, or Hoare-style calculi. A related approach to TARO that uses a view-based semantics for persistent x86-TSO has been developed by Bila et al. [2022].

Another series of works has focussed on semantics that support the *relaxed dependencies* that are allowed by C11 [Jagadeesan et al. 2020; Kang et al. 2017; Lee et al. 2020; Paviotti et al. 2020]. These have been followed more recently by logics and verification over this semantics [Svendsen et al. 2018; Wright et al. 2021]. However, relaxed dependencies produce high levels of non-determinism, making verification significantly more complex. We consider a verification framework that supports relaxed dependencies and STMs to be a topic for future research.

More recent works include *robustness* of C11-style programs, which aims to show "adequate synchronisation" so that the relaxed memory executions reduce to executions under stronger memory models [Margalit and Lahav 2021]. Such reductions, although automatic, are limited to finite state systems, and a small number of threads. Furthermore, it is currently unclear how they would handle client-library synchronisation or relaxed (non-SC) specifications.

Correctness conditions under relaxed memory. Following the extensive literature on the semantics of relaxed memory architectures, a natural next question has been the development of library abstractions for relaxed memory. One aim has been to ensure *observational refinement* and *compositionality* of the implemented objects. A series of works have considered reformulations of *linearizability* [Doherty et al. 2018; Dongol et al. 2018b; Raad et al. 2019a] by presenting suitable weakenings fine-tuned to the underlying memory model. This includes extensions of linearizability, e.g., so that it is defined in terms of axiomatic (aka declarative) relaxed memory models [Dongol

et al. 2018b; Raad et al. 2019a] and those that are based on the more abstract concept of execution structures [Doherty et al. 2018]. Recent works have covered verification of relaxed memory concurrent data structures that have been developed to satisfy the conditions described above [Dalvandi and Dongol 2021; Krishna et al. 2020; Raad et al. 2019a], but none of these cover transactions.

Khyzha and Lahav [2022] have recently developed notions of abstraction for crash resilient libraries, providing correctness conditions (extending linearizability) that ensure contextual refinement for concurrent objects executed over the PSC (persistent sequential consistency) model. They do so by exposing the internal synchronisation mechanisms that are used to implement an object in the history (in addition to the invocations and responses). Our work differs since we consider transactional memory libraries as opposed to concurrent objects, use a different memory model and focus on verification of contextual refinement directly. Nevertheless, in future work, it would be interesting to see if their methods provide an alternative method for specifying concurrent object and transactional memory libraries in C11.

Several papers have revisited transaction semantics in the context of relaxed memory models [Chong et al. 2018; Dongol et al. 2018a, 2019; Raad et al. 2019b]. Raad et al have considered relaxed memory and *snapshot isolation* [Raad et al. 2018, 2019b], which is a weaker condition than serializability (and hence opacity and TMS2). The question of whether snapshot isolation can be fully exploited by implementations in a relaxed memory setting remains a topic of future research, with most transactional implementations aiming to support at least serializability [Zardoshti et al. 2019]. Dongol et al. [2018a] and Chong et al. [2018] have provided *axiomatic* a transactional semantics integrated with relaxed memory models, focussing on hardware memory models and hardware transactions. Chong et al. [2018] additionally propose a model for C11 transactions, but these models are focused on transactions within the compiler, as opposed to STMs. Finally, the axiomatic models proposed in these earlier works [Chong et al. 2018; Dongol et al. 2018a] are not suitable for operational verification, e.g., as supported by TARO, where we require an operational semantics as provided by TMS2-RA.

Another set of works has focussed on distributed (relaxed) transactions [Beillahi et al. 2021a,b; Xiong et al. 2020]. Although there are analogues between transactions in distributed systems and relaxed memory, constraints such as replication consistency and session order are not factors in shared memory, and hence the underlying issues are fundamentally different. Xiong et al. [2020] describe a taxonomy of distributed transactional models supported by an operational semantics. It would be interesting to investigate whether TARO can be adapted to cope with client-object systems in their models.

Relaxed memory TM implementations. There is a set of recent works on implementing TM algorithms in C11 [Spear et al. 2020; Zardoshti et al. 2019]. The focus here has been real-world implementability of STMs via compiler support. Since the focus is on benchmarks and real-world workflows, these works neither consider a formal semantics nor provide a verification framework. Our work can thus be seen as providing a formal basis to support to these efforts. In particular, we show how the serialisability specifications assumed by Spear et al. [2020]; Zardoshti et al. [2019] can be relaxed, without impacting correctness, while improving performance.

8 CONCLUSIONS

In this paper, we have presented a new approach to release-acquire transactions for RC11 RAR (a fragment of C11 that supports relaxed as well as release-acquire atomics). We have developed a new TM specification, TMS2-RA, that extends TMS2 to a relaxed memory context by describing the interactions between transactions and their clients. We implement TMS2-RA by TML-RA, which is

an adaptation of an existing eager algorithm, TML. We show that TML-RA outperforms TML-SC using the STAMP benchmarks.

Our second set of contributions covers the verification of release-acquire TM implementations. We focus on proofs at two levels: (i) correctness of *client programs* that use TMS2-RA, and (ii) correctness of *implementations* of TMS2-RA. For (i), we have developed a logic, TARO, extending [Dalvandi and Dongol 2021], and used this logic to prove that TMS2-RA does indeed guarantee the desired client-side synchronisation properties. For (ii), we have applied a simulation method, similar to [Dalvandi and Dongol 2021] and proved a forward simulation between TML-RA and TMS2-RA. All proofs for (i) and (ii) as well as all meta-level soundness results are fully mechanised in the Isabelle/HOL proof assistant, providing a high level of assurance to our results.

Our motivation for using TML as the main implementation case study was to start with a simple algorithm with an existing proof in SC [Derrick et al. 2018]. TML performs a global synchronisation through a CAS on a single location, which degrades performance on write-heavy workloads. For improved scalability, there are more sophisticated algorithms like TL2 [Dice et al. 2006] that offer per-location locking as well as hybrid TM implementations [Matveev and Shavit 2015] that combine hardware and software TM. TMS2 is known to be a sufficient abstraction for hybrid TMs in SC [Armstrong and Dongol 2017], so it is likely that TMS2-RA also provides a basis for developing and verifying relaxed and release-acquire versions of these more sophisticated algorithms. We leave such studies for future work.

ACKNOWLEDGMENTS

The authors would also like to thank the Eleni Vafeiadi Bila and anonymous referees for their valuable comments and helpful suggestions. Dalvandi and Dongol are supported by EPSRC Grant EP/R032556/1. Dongol is additionally supported by EPSRC Grant EP/V038915/1, EPSRC Grant EP/R025134/2, ARC Grant DP190102142 and VeTSS.

REFERENCES

P. A. Abdulla, J. Arora, M. F. Atig, and S. N. Krishna. 2019. Verification of programs under the release-acquire semantics. In *PLDI*, K. S. McKinley and K. Fisher (Eds.). ACM, 1117–1132. https://doi.org/10.1145/3314221.3314649

J. Alglave and P. Cousot. 2017. Ogre and Pythia: an invariance proof method for weak consistency models. In *POPL*, G. Castagna and A. D. Gordon (Eds.). ACM, 3–18.

J. Alglave, L. Maranget, and M. Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74.

A. Armstrong and B. Dongol. 2017. Modularising Opacity Verification for Hybrid Transactional Memory. In *FORTE (LNCS, Vol. 10321)*, A. Bouajjani and A. Silva (Eds.). Springer, 33–49.

A. Armstrong, B. Dongol, and S. Doherty. 2017. Proving Opacity via Linearizability: A Sound and Complete Method. In *FORTE (LNCS, Vol. 10321)*, A. Bouajjani and A. Silva (Eds.). Springer, 50–66.

G. Assa, H. Meir, G. Golan-Gueta, I. Keidar, and A. Spiegelman. 2020. Nesting and composition in transactional data structure libraries. In *PPoPP '20*, R. Gupta and X. Shen (Eds.). ACM, 405–406. https://doi.org/10.1145/3332466.3374514

G. Assa, H. Meir, G. Golan-Gueta, I. Keidar, and A. Spiegelman. 2021. Using Nesting to Push the Limits of Transactional Data Structure Libraries. In *OPODIS (LIPIcs, Vol. 217)*, Q. Bramas, V. Gramoli, and A. Milani (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:17. https://doi.org/10.4230/LIPIcs.OPODIS.2021.30

H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. 2018. Characterizing Transactional Memory Consistency Conditions Using Observational Refinement. *J. ACM* 65, 1 (2018), 2:1–2:44.

M. Batty, A. F. Donaldson, and J. Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *POPL*. ACM, 634–648.

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. 2011. Mathematizing C++ concurrency. In *POPL*, T. Ball and M. Sagiv (Eds.). ACM, 55–66.

S. M. Beillahi, A. Bouajjani, and C. Enea. 2021a. Checking Robustness Between Weak Transactional Consistency Models. In *ESOP (LNCS, Vol. 12648)*, N. Yoshida (Ed.). Springer, 87–117. https://doi.org/10.1007/978-3-030-72019-3_4

S. M. Beillahi, A. Bouajjani, and C. Enea. 2021b. Robustness Against Transactional Causal Consistency. *Log. Methods Comput. Sci.* 17, 1 (2021). https://lmcs.episciences.org/7149

E. Vafeiadi Bila, B. Dongol, O. Lahav, A. Raad, and J. Wickerson. 2022. View-Based Owicki-Gries Reasoning for Persistent x86-TSO. In *ESOP (Lecture Notes in Computer Science, Vol. 13240)*, I. Sergey (Ed.). Springer, 234–261. https://doi.org/10.1007/978-3-030-99336-8_9

S. Böhme and T. Nipkow. 2010. Sledgehammer: Judgement Day. In *IJCAR (LNCS, Vol. 6173)*. Springer, 107–121.

N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. 2010. Transactional predication: high-performance concurrent sets and maps for STM. In *PODC*, A. W. Richa and R. Guerraoui (Eds.). ACM, 6–15. https://doi.org/10.1145/1835698.1835703

N. Chong, T. Sorensen, and J. Wickerson. 2018. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *PLDI*, J. S. Foster and D. Grossman (Eds.). ACM, 211–225. https://doi.org/10.1145/3192366.3192373

cppreference.com. 2022. std::atomic_compare_exchange. https://en.cppreference.com/w/cpp/atomic/atomic_compare_exchange Accessed 18 July, 2022.

L. Dalessandro, D. Dice, M. L. Scott, N. Shavit, and M. F. Spear. 2010. Transactional Mutex Locks. In *Euro-Par (2) (LNCS, Vol. 6272)*. Springer, 2–13.

S. Dalvandi, S. Doherty, B. Dongol, and H. Wehrheim. 2020a. Owicki-Gries Reasoning for C11 RAR. In *ECOOP (LIPIcs, Vol. 166)*, R. Hirschfeld and T. Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.11

S. Dalvandi, S. Doherty, B. Dongol, and H. Wehrheim. 2020b. Owicki-Gries Reasoning for C11 RAR (Artifact). *Dagstuhl Artifacts Ser.* 6, 2 (2020), 15:1–15:2. https://doi.org/10.4230/DARTS.6.2.15

S. Dalvandi and B. Dongol. 2021. Verifying C11-style weak memory libraries. In *PPoPP*, J. Lee and E. Petrank (Eds.). ACM, 451–453. https://doi.org/10.1145/3437801.3441619

S. Dalvandi and B. Dongol. 2022a. *Implementing and Verifying Release-Acquire Transactional Memory (Artifact).* https://doi.org/10.5281/zenodo.6899919

S. Dalvandi and B. Dongol. 2022b. Implementing and Verifying Release-Acquire Transactional Memory (Extended Version). https://doi.org/10.48550/ARXIV.2208.00315

S. Dalvandi, B. Dongol, S. Doherty, and H. Wehrheim. 2022. Integrating Owicki-Gries for C11-Style Memory Models into Isabelle/HOL. *J. Autom. Reason.* 66, 1 (2022), 141–171. https://doi.org/10.1007/s10817-021-09610-2

H.-H. Dang, J. Jung, J. Choi, D.-T. Nguyen, W. Mansky, J. Kang, and D. Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *PLDI*, R. Jhala and I. Dillig (Eds.). ACM, 792–808. https://doi.org/10.1145/3519939.3523451

W. P. de Roever and K. Engelhardt. 1998. *Data Refinement: Model-oriented Proof Theories and their Comparison.* Cambridge Tracts in Theoretical Computer Science, Vol. 46. Cambridge University Press.

J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, O. Travkin, and H. Wehrheim. 2018. Mechanized proofs of opacity: a comparison of two techniques. *Formal Aspects Comput.* 30, 5 (2018), 597–625. https://doi.org/10.1007/s00165-017-0433-3

D. Dice, O. Shalev, and N. Shavit. 2006. Transactional Locking II. In *DISC (Lecture Notes in Computer Science, Vol. 4167)*, S. Dolev (Ed.). Springer, 194–208. https://doi.org/10.1007/11864219_14

S. Doherty, B. Dongol, J. Derrick, G. Schellhorn, and H. Wehrheim. 2016. Proving Opacity of a Pessimistic STM. In *OPODIS (LIPIcs, Vol. 70)*, P. Fatourou, E. Jiménez, and F. Pedone (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 35:1–35:17.

S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick. 2018. Making Linearizability Compositional for Partially Ordered Executions. In *iFM (LNCS, Vol. 11023)*, C. A. Furia and K. Winter (Eds.). Springer, 110–129. https://doi.org/10.1007/978-3-319-98938-9_7

S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick. 2019. Verifying C11 programs operationally. In *PPoPP*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 355–365.

S. Doherty, L. Groves, V. Luchangco, and M. Moir. 2013. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.* 25, 5 (2013), 769–799.

M. Doko and V. Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *ESOP*. 448–475.

S. Dolan, KC Sivaramakrishnan, and A. Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *PLDI* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 242–255.

B. Dongol and L. Groves. 2016. Contextual Trace Refinement for Concurrent Objects: Safety and Progress. In *ICFEM (Lecture Notes in Computer Science, Vol. 10009)*, K. Ogata, M. Lawford, and S. Liu (Eds.). 261–278. https://doi.org/10.1007/978-3-319-47846-3_17

B. Dongol, R. Jagadeesan, and J. Riely. 2018a. Transactions in relaxed memory architectures. *PACMPL* 2, POPL (2018), 18:1–18:29.

B. Dongol, R. Jagadeesan, and J. Riely. 2019. Modular transactions: bounding mixed races in space and time. In *PPoPP*, J. K. Hollingsworth and I. Keidar (Eds.). ACM, 82–93. https://doi.org/10.1145/3293883.3295708

B. Dongol, R. Jagadeesan, J. Riely, and A. Armstrong. 2018b. On abstraction and compositionality for weak-memory linearisability. In *VMCAI (LNCS, Vol. 10747)*. Springer, 183–204.

M. Emmi and C. Enea. 2019. Weak-consistency specification via visibility relaxation. *Proc. ACM Program. Lang.* 3, POPL (2019), 60:1–60:28. https://doi.org/10.1145/3290373

A. Gotsman and H. Yang. 2011. Liveness-Preserving Atomicity Abstraction. In *ICALP (Lecture Notes in Computer Science, Vol. 6756)*, L. Aceto, M. Henzinger, and J. Sgall (Eds.). Springer, 453–465. https://doi.org/10.1007/978-3-642-22012-8_36

R. Guerraoui and M. Kapalka. 2010. *Principles of Transactional Memory.* Morgan & Claypool Publishers.

M. He, V. Vafeiadis, S. Qin, and J. F. Ferreira. 2016. Reasoning about Fences and Relaxed Atomics. In *PDP*. IEEE Computer Society, 520–527.

M. Herlihy and J. E. B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, A. J. Smith (Ed.). ACM, 289–300.

M. Herlihy and J. M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS* 12, 3 (1990), 463–492.

R. Jagadeesan, A. Jeffrey, and J. Riely. 2020. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 194:1–194:30. https://doi.org/10.1145/3428262

J.-O. Kaiser, H.-H. Dang, D. D., O. Lahav, and V. Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP (LIPIcs, Vol. 74)*, P. Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 17:1–17:29.

J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*. ACM, 175–189.

A. Khyzha and O. Lahav. 2022. Abstraction for Crash-Resilient Objects. In *ESOP (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 262–289. https://doi.org/10.1007/978-3-030-99336-8_10

M. Kokologiannakis, A. Raad, and V. Vafeiadis. 2019. Model checking for weakly consistent libraries. In *PLDI*, K. S. McKinley and K. Fisher (Eds.). ACM, 96–110. https://doi.org/10.1145/3314221.3314609

S. Krishna, M. Emmi, C. Enea, and D. Jovanovic. 2020. Verifying Visibility-Based Weak Consistency. In *ESOP (LNCS, Vol. 12075)*, P. Müller (Ed.). Springer, 280–307. https://doi.org/10.1007/978-3-030-44914-8_11

O. Lahav, E. Namakonov, J. Oberhauser, A. Podkopaev, and V. Vafeiadis. 2021. Making weak memory models fair. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485475

O. Lahav and V. Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *ICALP (LNCS, Vol. 9135)*, M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann (Eds.). Springer, 311–323.

O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*. ACM, 618–632.

L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

S.-H. Lee, M. Cho, A. Podkopaev, S. Chakraborty, C.-K. Hur, O. Lahav, and V. Vafeiadis. 2020. Promising 2.0: global optimizations in relaxed memory concurrency. In *PLDI*, A. F. Donaldson and E. Torlak (Eds.). ACM, 362–376. https://doi.org/10.1145/3385412.3386010

M. Lesani, V. Luchangco, and M. Moir. 2012. Putting Opacity in Its Place. In *WTTM*.

M. Lesani, L. Xia, A. Kaseorg, C. J. Bell, A. Chlipala, B. C. Pierce, and S. Zdancewic. 2022. C4: verified transactional objects. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–31. https://doi.org/10.1145/3527324

N. A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann.

R. Margalit and O. Lahav. 2021. Verifying Observational Robustness against a C11-Style Memory Model. *Proc. ACM Program. Lang.* 5, POPL, Article 4 (Jan. 2021), 33 pages. https://doi.org/10.1145/3434285

A. Matveev and N. Shavit. 2015. Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. In *ASPLOS*, Ö. Ö., K. Ebcioglu, and S. Dwarkadas (Eds.). ACM, 59–71. https://doi.org/10.1145/2694344.2694393

C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*, D. Christie, A. Lee, O. Mutlu, and B. G. Zorn (Eds.). IEEE Computer Society, 35–46. https://doi.org/10.1109/IISWC.2008.4636089

S. S. Owicki and D. Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Inf.* 6 (1976), 319–340.

M. Paviotti, S. Cooksey, A. Paradis, D. Wright, S. Owens, and M. Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *ESOP (LNCS, Vol. 12075)*, P. Müller (Ed.). Springer, 599–625. https://doi.org/10.1007/978-3-030-44914-8_22

A. Podkopaev, I. Sergey, and A. Nanevski. 2016. Operational Aspects of C/C++ Concurrency. *CoRR* abs/1606.01400 (2016). arXiv:1606.01400

A. Raad, M. Doko, L. Rozic, O. Lahav, and V. Vafeiadis. 2019a. On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.* 3, POPL (2019), 68:1–68:31. https://doi.org/10.1145/3290381

A. Raad, O. Lahav, and V. Vafeiadis. 2018. On Parallel Snapshot Isolation and Release/Acquire Consistency. In *ESOP (LNCS, Vol. 10801)*, A. Ahmed (Ed.). Springer, 940–967. https://doi.org/10.1007/978-3-319-89884-1_33

A. Raad, O. Lahav, and V. Vafeiadis. 2019b. On the Semantics of Snapshot Isolation. In *VMCAI (LNCS, Vol. 11388)*, C. Enea and R. Piskac (Eds.). Springer, 1–23. https://doi.org/10.1007/978-3-030-11245-5_1

M. Rodriguez and M. F. Spear. 2020. Brief Announcement: On Implementing Software Transactional Memory in the C++ Memory Model. In *PODC*, Y. Emek and C. Cachin (Eds.). ACM, 224–226. https://doi.org/10.1145/3382734.3405746

S. Scargall. 2020. *Programming Persistent Memory: A Comprehensive Guide for Developers.* APress. https://doi.org/10.1007/978-1-4842-4932-1_8

G. Sela, M. Herlihy, and E. Petrank. 2021. Brief Announcement: Linearizability: A Typo. In *PODC*, A. Miller, K. Censor-Hillel, and J. H. Korhonen (Eds.). ACM, 561–564. https://doi.org/10.1145/3465084.3467944

N. Shavit and D. Touitou. 1997. Software Transactional Memory. *Distributed Computing* 10, 2 (1997), 99–116.

M. Spear, H. Boehm, V. Luchangco, M. L. Scott, and M. Wong. 2020. *Transactional Memory Lite Support in C++.* Technical Report. isocpp.

A. J. Summers and P. Müller. 2018. Automating Deductive Verification for Weak-Memory Programs. In *TACAS (LNCS, Vol. 10805)*, D. Beyer and M. Huisman (Eds.). Springer, 190–209.

K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, and V. Vafeiadis. 2018. A Separation Logic for a Promising Semantics. In *ESOP (LNCS, Vol. 10801)*, A. Ahmed (Ed.). Springer, 357–384.

J. Tassarotti, D. Dreyer, and V. Vafeiadis. 2015. Verifying read-copy-update in a logic for weak memory. In *PLDI*, D. Grove and S. Blackburn (Eds.). ACM, 110–120. https://doi.org/10.1145/2737924.2737992

A. Turon, V. Vafeiadis, and D. Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, A. P. Black and T. D. Millstein (Eds.). ACM, 691–707.

V. Vafeiadis and C. Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*. 867–884.

D. Wright, M. Batty, and B. Dongol. 2021. Owicki-Gries Reasoning for C11 Programs with Relaxed Dependencies. 13047 (2021), 237–254. https://doi.org/10.1007/978-3-030-90870-6_13

S. Xiong, A. Cerone, A. Raad, and P. Gardner. 2020. Data Consistency in Transactional Storage Systems: A Centralised Semantics. In *ECOOP (LIPIcs, Vol. 166)*, R. Hirschfeld and T. Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:31. https://doi.org/10.4230/LIPIcs.ECOOP.2020.21

P. Zardoshti, T. Zhou, P. Balaji, M. L. Scott, and M. F. Spear. 2019. Simplifying Transactional Memory Support in C++. *ACM Trans. Archit. Code Optim.* 16, 3 (2019), 25:1–25:24. https://doi.org/10.1145/3328796