Compositional Verification of Relaxed-Memory Program Transformations

MIKE DODDS, MARK BATTY, and ALEXEY GOTSMAN

This paper is about verifying program transformations on an axiomatic relaxed memory model of the kind used in C/C++ and Java. Relaxed models present particular challenges for verifying program transformations, because they generate many additional modes of interaction between code and context. For a block of code being transformed, we define a denotation from its behaviour in a set of representative contexts. Our denotation summarises interactions of the code block with the rest of the program both through local and global variables, and through subtle synchronisation effects due to relaxed memory. We can then prove that a transformation does not introduce new program behaviours by comparing the denotations of the code block before and after. Our approach is compositional: by examining only representative contexts, transformations are verified for any context. It is also fully abstract, meaning any valid transformation can be verified. We cover several tricky aspects of C/C++-style memory models, including release-acquire operations, sequentially consistent fences, and non-atomics. We also define a variant of our denotation that is finite at the cost of losing full abstraction. Based on this variant, we have implemented a prototype verification tool and applied it to automatically prove and disprove a range of compiler optimisations.

1 INTRODUCTION

Context and objectives. Any program defines a collection of observable behaviours: a sorting algorithm maps unsorted to sorted sequences, and a paint program responds to mouse clicks by updating a rendering. It is often desirable to transform a program without introducing new observable behaviours – for example, in a compiler optimisation or programmer refactoring. Such transformations are called *observational refinements*, and they ensure that properties of the original program will carry over to the transformed version. It is also desirable for transformations to be *compositional*, meaning that they can be applied to a block of code irrespective of the surrounding program context. Compositional transformations are particularly useful for automated systems such as compilers, where they are known as *peephole optimisations*.

The semantics of the language is highly significant in determining which transformations are valid, because it determines the ways that a block of code being transformed can interact with its context and thereby affect the observable behaviour of the whole program. Our work applies to a relaxed memory concurrent setting. Thus, the context of a code-block includes both code sequentially before and after the block, and code that runs in parallel. Relaxed memory means that different threads can observe different, apparently contradictory orders of events – such behaviour is permitted by programming languages to reflect CPU-level relaxations and to allow compiler optimisations.

We focus on *axiomatic* memory models of the type used in C/C++ and Java. In axiomatic models, program executions are represented by structures of memory actions and relations on them, and program semantics is defined by a set of axioms constraining these structures. Reasoning about the correctness of program transformations on such memory models is very challenging, and indeed, compiler optimisations have been repeatedly shown unsound with respect to models they were intended to support (Vafeiadis et al. 2015; Ševčík and Aspinall 2008). The fundamental difficulty is that axiomatic models are defined in a global, non-compositional way, making it very challenging to reason compositionally about the single code-block being transformed.

Publication date: January 2017.

Approach. Suppose we have a code-block B, embedded into an unknown program context. We define a *denotation* for the code-block which summarises its behaviour in a restricted representative context. The denotation consists of a set of *histories* which track interactions across the boundary between the code-block and its context, but abstract from internal structure of the code-block. We can then validate a transformation from code-block B to B' by comparing their denotations. This approach is compositional: it requires reasoning only about the code-blocks and representative contexts; the validity of the transformation in an arbitrary context will follow. It is also *fully abstract*, meaning that it can verify any valid transformation: considering only representative contexts and histories does not lose generality.

We also define a variant of our denotation that is *finite* at the cost of losing full abstraction. We achieve this by further restricting the form of contexts one needs to consider in exchange for tracking more information in histories. For example, it is unnecessary to consider executions where two context operations read from the same write.

Using this finite denotation, we implement a prototype verification tool, Stellite. Our tool converts an input transformation into a model in the Alloy language (Jackson 2012), and then checks that the transformation is valid using the Alloy* solver (Milicevic et al. 2015). Our tool can prove or disprove a range of introduction, elimination, and exchange compiler optimisations. Many of these were verified by hand in previous work; our tool verifies them automatically.

Contributions. Our contribution is twofold. First, we define the first fully abstract denotational semantics for an axiomatic relaxed model. Previous proposals in this space targeted either non-relaxed sequential consistency (Brookes 1996) or much more restrictive operational relaxed models (Burckhardt et al. 2010; Jagadeesan et al. 2012; Poetzl and Kroening 2016). Second, we show it is feasible to automatically verify relaxed-memory program transformations. Previous techniques required laborious proofs by hand or in a proof assistant (Vafeiadis et al. 2015; Vafeiadis and Zappa Nardelli 2011; Ševčík and Aspinall 2008; Ševčík et al. 2011, 2013). Our target model is derived from the C/C++ 2011 standard (The C++ Standards Committee 2011). However, our aim is not to handle C/C++ per se (especially as the model is in flux in several respects; see §3.1). Rather we target the simplest axiomatic model rich enough to demonstrate our approach.

2 OBSERVATION AND TRANSFORMATION

Observational refinement. The notion of *observation* is crucial when determining how different programs are related. For example, observations might be I/O behaviour or writes to special variables. Given program executions X_1 and X_2, we write $X_1 \preccurlyeq_{\mathsf{ex}} X_2$ if the observations in X_1 are replicated in X_2. Lifting this notion, a program P_1 *observationally refines* another P_2 if every observable behaviour of one could also occur with the other – we write this $P_1 \preccurlyeq_{\mathsf{pr}} P_2$. More formally, let $[\![-]\!]$ be the map from programs to sets of executions. Then we define $\preccurlyeq_{\mathsf{pr}}$ as:

$$P_1 \preccurlyeq_{\mathsf{pr}} P_2 \quad \overset{\Delta}{\iff} \quad \forall X_1 \in [\![P_1]\!].\ \exists X_2 \in [\![P_2]\!].\ X_1 \preccurlyeq_{\mathsf{ex}} X_2 \tag{1}$$

Compositional transformation. Many common program transformations are *compositional*: they modify a sequential fragment of the program without examining the rest of the program. We call the former the *code-block* and the latter its *context*. Contexts can include sequential code before and after the block, and concurrent code that runs in parallel with it. Code-blocks are sequential, i.e. they do not feature internal concurrency. A context C and code-block B can be composed to give a whole program $C(B)$.

A transformation $B_2 \rightsquigarrow B_1$ replaces some instance of the code-block B_2 with B_1. To validate such transformation, we must establish whether *every* whole program containing B_1 observationally refines the same program with B_2 substituted. If this holds, we say that B_1 observationally refines B_2, written $B_1 \preccurlyeq_{\mathsf{bl}} B_2$, defined by lifting $\preccurlyeq_{\mathsf{pr}}$ as

```
      store(x,0); store(y,0);
store(x,1);      || store(y,1);
v1 := load(y);   || v2 := load(x);
```

```
      store(f,0); store(x,0);
store(x,1);  || b := load(f);
store(f,1);  || if (b == 1)
             ||   r := load(x);
```

Fig. 1. *Left:* store-buffering (SB) example. *Right:* message-passing (MP) example.

follows:

$$B_1 \preccurlyeq_{\mathsf{bl}} B_2 \overset{\Delta}{\iff} \forall C.\ C(B_1) \preccurlyeq_{\mathsf{pr}} C(B_2) \tag{2}$$

If $B_1 \preccurlyeq_{\mathsf{bl}} B_2$ holds, then the compiler can replace block B_2 with block B_1 irrespective of the whole program, i.e. $B_2 \rightsquigarrow B_1$ is a valid transformation. Thus, deciding $B_1 \preccurlyeq_{\mathsf{bl}} B_2$ is the core problem in validating compositional transformations.

The language semantics is highly significant in determining observational refinement. For example, the code blocks B_1: `store(x,2); store(x,5)` and B_2: `store(x,5)` are observationally equivalent in a sequential setting, but in a concurrent setting the intermediate state, `x` = 2, can be observed in B_2 but not B_1. In a relaxed-memory setting there is no global state seen by all threads, which further complicates the notion of observation.

Compositional verification. To establish $B_1 \preccurlyeq_{\mathsf{bl}} B_2$, it is difficult to examine all possible syntactic contexts. Our approach is to construct a *denotation* for each code-block – a simplified, ideally finite, summary of possible interactions between the block and its context. We then define a *refinement relation* on denotations and use it to establish observational refinement. We write $B_1 \sqsubseteq B_2$ when the denotation of B_1 refines that of B_2.

Refinement on denotations should be *adequate*, i.e., it should validly approximate observational refinement: $B_1 \sqsubseteq B_2 \implies B_1 \preccurlyeq_{\mathsf{bl}} B_2$. Hence, if $B_1 \sqsubseteq B_2$, then $B_2 \rightsquigarrow B_1$ is a valid transformation. It is also desirable for the denotation to be *fully abstract*: $B_1 \preccurlyeq_{\mathsf{bl}} B_2 \implies B_1 \sqsubseteq B_2$. This means any valid transformation can be verified by comparing denotations. Below we define several versions of $\sqsubseteq$ with different properties.

3 TARGET LANGUAGE AND CORE MEMORY MODEL

We now describe our target language. Our language's memory model is derived from the C/C++ 2011 standard (henceforth '*C11*'), as formalized by Batty et al. (2011); The C++ Standards Committee (2011). However, we simplify our model in several ways; see end of section for details. In C11 terms, our model covers release-acquire and non-atomic operations, and sequentially consistent fences. To simplify the presentation, at first we omit non-atomics, and extend our approach to cover them in §7. Thus, all operations in this section correspond to C11's release-acquire.

Relaxed memory primer. In a sequentially consistent concurrent system, there is a total temporal order on loads and stores, and loads take the value of the most recent store; in particular, they cannot read overwritten values, or values written in the future. A *relaxed* (or *weak*) memory model weakens this total order, allowing behaviours forbidden under sequential consistency. Two standard examples of relaxed behaviour are *store buffering* and *message passing*, shown in Figure 1.

In most relaxed models `v1 = v2 = 0` is a possible post-state for SB. This cannot occur on a sequentially consistent system: if `v1` = 0 then `store(y,1)` must be ordered after the load of `y`, which would order `store(x,1)` before the load of `x`, forcing it to assign `v2` = 1. In some relaxed models, `b` = 1 ∧ `r` = 0 is a possible post-state for MP. This is undesirable if, for example, `x` is a complex data-structure and `f` is a flag indicating it has been safely created.

Language syntax. Programs in the language we consider manipulate *thread-local variables* $l, l_1, l_2 \ldots \in \mathsf{LVar}$ and *global variables* $x, y, \ldots \in \mathsf{GVar}$, coming from disjoint sets LVar and GVar. Each variable stores a value from a finite set Val and is initialised to $0 \in \mathsf{Val}$. Constants are encoded by special read-only thread-local variables. We assume that each thread uses the same set of thread-local variable names LVar. The syntax of the programming language is as follows:

$$\begin{aligned} C &::= l := E \mid \mathtt{store}(x,l) \mid l := \mathtt{load}(x) \mid l := \mathtt{LL}(x) \mid l' := \mathtt{SC}(x,l) \mid \mathtt{fence} \mid \\ &\quad\;\; C_1 \parallel C_2 \mid C_1; C_2 \mid \mathtt{if}\,(l)\,\{C_1\}\,\mathtt{else}\,\{C_2\} \mid \{-\} \\ E &::= l \mid l_1 = l_2 \mid l_1 \neq l_2 \mid \ldots \end{aligned}$$

Many of the constructs are standard. $\mathtt{LL}(x)$ and $\mathtt{SC}(x,l)$ are *load-link* and *store-conditional*, which are basic concurrency operations available on many platforms (e.g., Power and ARM). A load-link $\mathtt{LL}(x)$ behaves as a standard load of global variable x. However, if it is followed by a store-conditional $\mathtt{SC}(x,l)$, the store fails and returns false if there are intervening writes to the same location. Otherwise the store-conditional writes l and returns true. The `fence` command is a *sequentially consistent fence*: interleaving such fences between all statements in a program guarantees sequentially consistent behaviour. We do not include *compare-and-swap* (CAS) command in our language because LL-SC is more general (Anderson and Moir 1995). Hardware-level LL-SC is used to implement C11 CAS on Power and ARM. Our language does not include loops because in this paper we do not consider infinite computations (see §3.1 for discussion). As a result, loops can be represented by their finite unrollings. Our `load` commands write into a local variable. In our examples, we sometimes write 'bare' loads without a local variable write.

The construct $\{-\}$ represents a block-shaped hole in the program. To simplify our presentation, we assume at most one hole appears on each control-flow path.[1] The set Prog of *whole programs* consists of programs without holes, while the set Contx of *contexts* consists of programs. The set Block of *code-blocks* are whole programs without parallel composition. We often write $P \in \mathsf{Prog}$ for a whole program, $B \in \mathsf{Block}$ for a code-block, and $C \in \mathsf{Contx}$ for a context. Given a context C and a code-block B, the composition $C(B)$ is C with its hole syntactically replaced by B. For example:

C: `load(x); {-}; store(y,l1)`, B: `store(x,2)` $\longrightarrow$ $C(B)$: `load(x); store(x,2); store(y,l1)`

We restrict Prog, Contx and Block syntactically: each SC must be preceded by LL at the same location, with no intervening SC, and we forbid LL-SC pairs from spanning parallel compositions, and from spanning the block/context boundary.

Memory model structure. The semantics of a whole program P is given by a set $[\![P]\!]$ of *executions*, which consist of *actions*, representing memory events on global variables, and several relations on these. Actions are tuples in the set $\mathsf{Action} \triangleq \mathsf{ActID} \times \mathsf{Kind} \times \mathsf{Option}(\mathsf{GVar}) \times \mathsf{Val}^*$. In an action $(a, k, z, b) \in \mathsf{Action}$: $a \in \mathsf{ActID}$ is the unique action identifier; $k \in \mathsf{Kind}$ is the kind of action – we use load, store, LL, SC, and the failed variant SC_f in the semantics, and will introduce further kinds as needed; $z \in \mathsf{Option}(\mathsf{GVar})$ is an option type consisting of either a single global variable $\mathsf{Just}(x)$ or None; and $b \in \mathsf{Val}^*$ is the vector of values (actions with multiple values are used in §4).

Given an action v, we use $\mathsf{gvar}(v)$ and $\mathsf{rval}(v)$ as selectors for the different fields. We often write actions so as to elide action identifiers and the option type. For example, $\mathsf{load}(x, 3)$ stands for $\exists i.\ (i, \mathsf{load}, \mathsf{Just}(x), [3])$. We also sometimes elide values. We call load and LL actions *reads*, and store and successful SC actions *writes*. Given a set of actions $\mathcal{A}$, we write, e.g., $\mathsf{reads}(\mathcal{A})$ to identify read actions in $\mathcal{A}$. Below, we range over all actions by u, v; read actions by r; write actions by w; and LL, SC actions by ll and sc respectively.

[1] Transformations that apply to multiple blocks at once can be simulated by using the fact our approach is compositional. This means that transformations can be applied in sequence using different divisions of the program into code-block and context.

$$
\begin{aligned}
\langle l := \mathtt{load}(x), \sigma\rangle &\triangleq \{(\{\mathsf{load}(x,a)\}, \emptyset, \sigma[l \mapsto a]) \mid a \in \mathsf{Val}\} \\
\langle \mathtt{store}(x,l), \sigma\rangle &\triangleq \{(\{\mathsf{store}(x,a)\}, \emptyset, \sigma) \mid \sigma(l) = a\} \\
\langle l := \mathtt{LL}(x), \sigma\rangle &\triangleq \{(\{\mathsf{LL}(x,a)\}, \emptyset, \sigma[l \mapsto a]) \mid a \in \mathsf{Val}\} \\
\langle l' := \mathtt{SC}(x,l), \sigma\rangle &\triangleq \{\{(\{\mathsf{SC}(x,a)\}, \emptyset, \sigma[l' \mapsto 1]) \mid \sigma(l) = a\} \cup \{(\{\mathsf{SC}_f(x)\}, \emptyset, \sigma[l' \mapsto 0])\} \\
\langle \mathtt{fence}, \sigma\rangle &\triangleq \{(\{ll, sc\}, \{(ll, sc)\}, \sigma) \mid ll = \mathsf{LL}(fen, 0) \wedge sc = \mathsf{SC}(fen, 0)\} \\
\langle C_1 \parallel C_2, \sigma\rangle &\triangleq \{(\mathcal{A}_1 \uplus \mathcal{A}_2, \mathsf{sb}_1 \cup \mathsf{sb}_2, \sigma) \mid (\mathcal{A}_1, \mathsf{sb}_1, \sigma_1) \in \langle C_1, \sigma\rangle \wedge (\mathcal{A}_2, \mathsf{sb}_2, \sigma_2) \in \langle C_2, \sigma\rangle\} \\
\langle C_1; C_2, \sigma\rangle &\triangleq \{(\mathcal{A}_1 \uplus \mathcal{A}_2, \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup (\mathcal{A}_1 \times \mathcal{A}_2), \sigma_2) \mid (\mathcal{A}_1, \mathsf{sb}_1, \sigma_1) \in \langle C_1, \sigma\rangle \wedge (\mathcal{A}_2, \mathsf{sb}_2, \sigma_2) \in \langle C_2, \sigma_1\rangle\} \\
\langle \mathsf{if}(l)\ \{C_1\}\ \mathsf{else}\ \{C_2\}, \sigma\rangle &\triangleq \begin{cases} \langle C_2, \sigma\rangle, & \text{if } \sigma(l) = 0 \\ \langle C_1, \sigma\rangle, & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 2. The thread-local semantics of our target language. We write $\mathcal{A}_1 \uplus \mathcal{A}_2$ for a union that is defined only when actions in $\mathcal{A}_1$ and $\mathcal{A}_2$ use disjoint sets of identifiers. We omit identifiers from actions to avoid clutter.

The semantics of a program $P \in \mathsf{Prog}$ is defined in two stages. First, a *thread-local semantics* of P produces a set $\langle P\rangle$ of *pre-executions* $(\mathcal{A}, \mathsf{sb}) \in \mathsf{PreExec}$. A pre-execution contains a finite set of memory actions $\mathcal{A} \in \mathsf{Action}$ that could be produced by the program. It has a transitive and irreflexive *sequence-before* relation $\mathsf{sb} \subseteq \mathcal{A} \times \mathcal{A}$, which defines the sequential order imposed by the program syntax.

For example two sequential statements in the same thread produce actions ordered in sb. The thread-local semantics takes into account control flow in P's threads and operations on local variables. However, it does not constrain the behaviour of global variables: the values threads read from them are chosen arbitrarily. This is addressed by extending pre-executions with extra relations, and filtering these executions using *validity axioms*.

Thread-local semantics. The thread-local semantics is defined formally in Figure 2. The semantics of a program $P \in \mathsf{Prog}$ is defined using function $\langle -, -\rangle \colon \mathsf{Prog} \times \mathsf{VMap} \to \mathcal{P}(\mathsf{PreExec} \times \mathsf{VMap})$. The values of local variables are tracked by a map $\sigma \in \mathsf{VMap} \triangleq \mathsf{LVar} \to \mathsf{Val}$. Given a program and an input local variable map, the function produces a set of pre-executions paired with an output variable map, representing the values of local variables at the end of the execution. Let σ_0 map every local variable to 0. Then $\langle P\rangle$, the thread-local semantics of a program P, is defined as

$$\langle P\rangle \quad \triangleq \quad \{(\mathcal{A}, \mathsf{sb}) \mid \exists \sigma'.\ (\mathcal{A}, \mathsf{sb}, \sigma') \in \langle P, \sigma_0\rangle\}$$

The significant property of the thread-local semantics is that it does not restrict the behaviour of global variables. For this reason, note that the clause for `load` in Figure 2 leaves the value a unrestricted. We take a simplified approach to local variables at thread creation: the initial variable map σ is copied to both threads in $C_1 \parallel C_2$, and the original map is restored when they complete. We follow Lahav et al. (2016) by encoding the `fence` command by a successful LL-SC pair to a distinguished variable $fen \in \mathsf{GVar}$ that is not otherwise read or written.

Execution structure. The semantics of a program P is a set $[\![P]\!]$ of *executions* $X = (\mathcal{A}, \mathsf{sb}, \mathsf{at}, \mathsf{rf}, \mathsf{mo}, \mathsf{hb}) \in \mathsf{Exec}$, where $(\mathcal{A}, \mathsf{sb})$ is a pre-execution and $\mathsf{at}, \mathsf{rf}, \mathsf{mo}, \mathsf{hb} \subseteq \mathcal{A} \times \mathcal{A}$. Given an execution X we sometimes write $\mathcal{A}(X), \mathsf{sb}(X), \ldots$ as selectors for the appropriate set or relation. The relations have the following purposes.

- *Atomicity* ($\mathsf{at} \subseteq \mathsf{sb}$) is an injective function from LL actions to SC actions that associates each successful store-conditional to the preceding load-link action on the same location, representing a matching LL-SC pair. This definition is made possible by the strong syntactic restrictions on the use of LL-SC.

- *Reads-from* (rf) is an injective map from reads to writes. A read and write action are related $w \xrightarrow{\mathsf{rf}} r$ if r takes its value from w.
- *Modification order* (mo) is an irreflexive, total order on write actions to each distinct variable. This is a per-variable order in which *all* threads observe writes to the variable; two threads cannot observe these writes in different orders.
- *Happens-before* (hb) is analogous to global temporal order – but unlike the sequentially consistent notion of time, it is partial. Happens-before is defined as $(\mathsf{sb} \cup \mathsf{rf})^+$: therefore statements ordered in the program syntax are ordered in time, as are reads with the writes they observe.

Validity axioms. The semantics $[\![P]\!]$ of a program P is the set of executions $X \in \mathsf{Exec}$ compatible with the thread-local semantics and the *validity axioms*, denoted $\mathsf{valid}(X)$:

$$[\![P]\!] \triangleq \{X \mid (\mathcal{A}(X), \mathsf{sb}(X)) \in \langle P \rangle \wedge \mathsf{valid}(X)\} \tag{3}$$

The validity axioms on an execution $(\mathcal{A}, \mathsf{sb}, \mathsf{at}, \mathsf{rf}, \mathsf{mo}, \mathsf{hb})$ are:

- HBdef: $\mathsf{hb} = (\mathsf{sb} \cup \mathsf{rf})^+$ *and* hb is acyclic.

  This axiom defines hb and enforces the intuitive property that there are no cycles in the temporal order. It also prevents an action reading from its hb-future: as rf is included in hb, this would result in a cycle.
- HBvsMO: $\neg\exists w_1, w_2.\ \begin{tikzcd} w_1 \arrow[r, bend left, "\mathsf{hb}"] & w_2 \arrow[l, bend left, "\mathsf{mo}"] \end{tikzcd}$

  This axiom requires that the order in which writes to a location become visible to threads cannot contradict the temporal order. But take note that writes may be ordered in mo but not hb.
- Coherence: $\neg\exists w_1, w_2, r.\ \begin{tikzcd} w_1 \arrow[r, "\mathsf{mo}"] \arrow[rr, bend right, "\mathsf{rf}"'] & w_2 \arrow[r, "\mathsf{hb}"] & r \end{tikzcd}$

  This axiom generalises the sequentially consistent prohibition on reading overwritten values. If two writes are ordered in mo, then intuitively the second overwrites the first. A read that follows some write in hb or mo cannot read from writes earlier in mo – these earlier writes have been overwritten. However, unlike in sequential consistency, hb is partial, so there may be multiple writes that an action can legally read.
- RFval: $\forall r.\ (\neg\exists w'.\ w' \xrightarrow{\mathsf{rf}} r) \implies (\mathsf{rval}(r) = 0 \wedge (\neg\exists w.\ w \xrightarrow{\mathsf{hb}} r \wedge \mathsf{gvar}(w) = \mathsf{gvar}(r)))$

  Most reads must take their value from a write, represented by an rf edge. However, the RFval axiom allows the rf edge to be omitted if the read takes the initial value 0 and there is no hb-earlier write to the same location. Intuitively, an hb-earlier write would supersede the initial value in a similar way to Coherence.
- Atom: $\neg\exists w_1, w_2, ll, sc.\ \begin{tikzcd} w_1 \arrow[r, "\mathsf{mo}"] \arrow[d, "\mathsf{rf}"'] & w_2 \arrow[d, "\mathsf{mo}"] \\ ll \arrow[r, "\mathsf{at}"] & sc \end{tikzcd}$

  This axiom is adapted from Lahav et al. (2016). For an LL-SC pair ll and sc, it ensures that there is no mo-intervening write w_2 that would invalidate the store.

Our model forbids the problematic relaxed behaviour of the message-passing (MP) program in Figure 1 that yields $\mathsf{b} = 1 \wedge \mathsf{r} = 0$. Figure 3 shows an (invalid) execution that would exhibit this behaviour. To avoid clutter, here and in the following we omit hb edges obtained by transitivity and local variable values. This execution is allowed by the thread-local semantics of the MP program, but it is ruled out by the Coherence validity axiom. As hb is transitively closed, there is a derived hb edge $\mathsf{store}(\mathsf{x}, 1) \xrightarrow{\mathsf{hb}} \mathsf{load}(\mathsf{x}, 0)$, which forms a Coherence violation. Thus, this is not an execution of the MP program. Indeed, any execution ending in $\mathsf{load}(\mathsf{x}, 0)$ is forbidden for the same reason, meaning that the undesirable MP relaxed behaviour cannot occur.

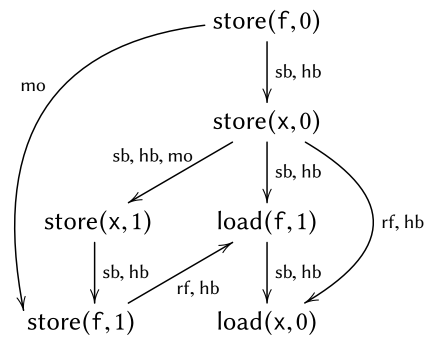


Fig. 3. Invalid execution for the MP example program.

Relaxed observations. Finally, we define a notion of observational refinement suitable for our relaxed model. We assume a subset of *observable* global variables, $\mathsf{OVar} \subseteq \mathsf{GVar}$, which can only be accessed by the context and not by the code-block. We consider the actions and the hb relation on these variables to be the observations. We write $X|_{\mathsf{OVar}}$ for the projection of X's action set and relations to OVar, and use this to define $\preccurlyeq_{\mathsf{ex}}$ for our model:

$$X \preccurlyeq_{\mathsf{ex}} Y \quad \stackrel{\Delta}{\iff} \quad \mathcal{A}(X|_{\mathsf{OVar}}) = \mathcal{A}(Y|_{\mathsf{OVar}}) \;\wedge\; \mathsf{hb}(Y|_{\mathsf{OVar}}) \subseteq \mathsf{hb}(X|_{\mathsf{OVar}})$$

This is lifted to programs and blocks as in §2, def. (1) and (2). Note that in the more abstract execution, actions on observable variables must be the same, but hb can be weaker. This is because we interpret hb as a constraint on time order: two actions that are unordered in hb could have occurred in either order, or in parallel. Thus, weakening hb allows more observable behaviours (see §2).

3.1 Differences from C11

Our language's memory model is derived from the C11 formalization in Batty et al. (2011), with a number of simplifications. We chose C11 because it demonstrates most of the important features of axiomatic language models. However, we do not target the precise C11 model: rather we target an abstracted model that is rich enough to demonstrate our approach. Relaxed language semantics is still a very active topic of research, and several C11 features are known to be significantly flawed, with multiple competing fixes proposed. Some of our differences from Batty et al. (2011) are intended to avoid such problematic features so that we can cleanly demonstrate our approach.

In C11 terms, our model covers release-acquire and non-atomic operations (the latter addressed in §7), and sequentially consistent fences. We deviate from C11 in the following ways:

- We omit *sequentially consistent* accesses because their semantics is known to be flawed in C11 (Lahav et al. 2017). We do handle sequentially consistent fences, but these are stronger than those of C11: the stronger semantics we use was proposed in Lahav et al. (2016) and proved sound under existing compilation strategies to common multiprocessors.
- We omit *relaxed* (RLX) accesses to avoid well-known problems with thin-air values (Batty et al. 2015). There are multiple recent competing proposals for fixing these problems, e.g. Jeffrey and Riely (2016); Kang et al. (2017); Pichon-Pharabod and Sewell (2016).
- We do not consider infinite computations, because the semantics of infinite computations in C11-style axiomatic models remains undecided in the literature (Batty et al. 2015). However, our proofs do not depend on the assumption that execution contexts are finite.

- Our language is based on shared variables, not dynamically allocated addressable memory, so for example we cannot write `y:=*x; z:=*y`. This simplifies our theory by allowing us to fix the variables accessed by a code-block up-front. We believe our results can be extended to support addressable memory, because C11-style models grant no special status to pointers; we elaborate on this in §4.
- We add LL-SC atomic instructions to our language in addition to C11's standard CAS. This increases the observational power of a context and is necessary for full abstraction in the presence of non-atomics; see §8. LL-SC is available as a hardware instruction on many platforms supporting C11, such as Power and ARM. However, we do not propose adding LL-SC to C11: rather, it supports an interesting result in relaxed memory model theory. Our adequacy results do not depend on LL-SC.

4 DENOTATIONS OF CODE-BLOCKS

We construct the denotation for a code-block in two steps: (1) generate the *block-local* executions under a set of special cut-down contexts; (2) from each of these executions, extract a summary of interactions between the code-block and the context called a *history*.

Block-local executions. The block-local executions of a block $B \in \mathsf{Block}$ omit context structure such as syntax and actions on variables not accessed in the block. Instead the context is represented by special actions call and ret, a set $\mathcal{A}_B$, and relations R_B and S_B, each covering an aspect of the interaction of the block and an arbitrary unrestricted context.

- *Local variables.* A context can include code that precedes and follows the block on the same thread, with interaction through local variables, but – due to syntactic restriction – not through LL/SC atomic regions. We capture this with special action $\mathsf{call}(\sigma)$ at the start of the block, and $\mathsf{ret}(\sigma')$ at the end, where $\sigma, \sigma' \colon \mathsf{LVar} \to \mathsf{Val}$ record the values of local variables at these points. Assume that variables in LVar are ordered: $l_1, l_2, \ldots, l_n$. Then $\mathsf{call}(\sigma)$ is encoded by the action $(i, \mathsf{call}, \mathsf{None}, [\sigma(l_1), \ldots\ \sigma(l_n)])$, with fresh identifier i. We encode ret in the same way.
- *Global variable actions.* The context can also interact with the block through concurrent reads and writes to global variables. These interactions are represented by set $\mathcal{A}_B$ of actions added to the ones generated by the thread-local semantics of the block. This set only contains actions on the variables VS_B that B can access (VS_B can be constructed syntactically).
- *Context happens-before.* The context can generate hb edges between its actions – to get adequacy (§2), we track these with a relation R_B over actions in $\mathcal{A}_B$, call and ret:

$$R_B \subseteq (\mathcal{A}_B \times \mathcal{A}_B) \cup (\mathcal{A}_B \times \{\mathsf{call}\}) \cup (\{\mathsf{ret}\} \times \mathcal{A}_B) \tag{4}$$

  The context can generate hb edges between actions directly if they are on the same thread, or indirectly through inter-thread reads. Likewise call / ret may be related to context actions on the same or different threads.
- *Context atomicity.* The context can generate at edges between its actions that we capture in the relation $S_B \subseteq \mathcal{A}_B \times \mathcal{A}_B$. We require this relation to be an injective function from LL to SC actions. We consider only cases where LL/SC pairs do not cross block boundaries, so we need not consider boundary-crossing at edges.

Together, call, ret, $\mathcal{A}_B$, R_B, and S_B represent a limited context, stripped of syntax, relations sb, moand rf, and actions on global variables other than VS_B. When constructing block-local executions, we represent all possible interactions by quantifying over all possible choices of σ, σ', $\mathcal{A}_B$, R_B and S_B. The set $[\![B, \mathcal{A}_B, R_B, S_B]\!]$ contains all executions of B under this special limited context. Formally, an execution $X = (\mathcal{A}, \mathsf{sb}, \mathsf{at}, \mathsf{rf}, \mathsf{mo}, \mathsf{hb})$ is in this set if:

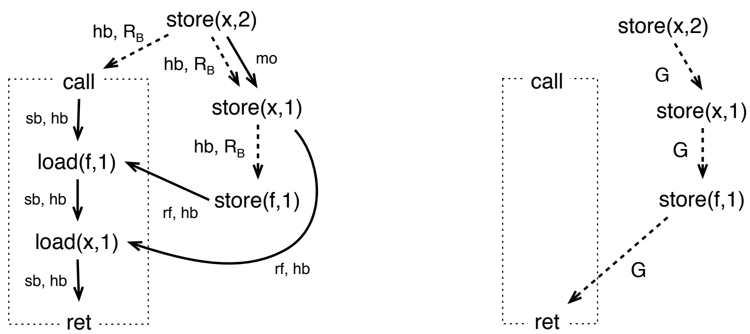


Fig. 4. *Left:* block-local execution. *Right:* corresponding history.

(1) $\mathcal{A}_B \subseteq \mathcal{A}$ and there exist variable maps σ, σ' such that $\{\mathsf{call}(\sigma), \mathsf{ret}(\sigma')\} \subseteq \mathcal{A}$. That is, the call, return, and extra context actions are included in the execution.
(2) There exists a set $\mathcal{A}_l$ and relation sb_l such that (i) $(\mathcal{A}_l, \mathsf{sb}_l, \sigma') \in \langle B, \sigma \rangle$; (ii) $\mathcal{A}_l = \mathcal{A} \setminus (\mathcal{A}_B \cup \{\mathsf{call}, \mathsf{ret}\})$; (iii) $\mathsf{sb}_l = \mathsf{sb} \setminus \{(\mathsf{call}, u), (u, \mathsf{ret}) \mid u \in \mathcal{A}_l\}$. That is, actions from the code-block satisfy the thread-local semantics, beginning with map σ and deriving map σ'. All actions arising from the block are between call and ret in sb.
(3) X satisfies the validity axioms, but with modified axioms HBdef′ and Atom′. We define HBdef′ as: $\mathsf{hb} = (\mathsf{sb} \cup \mathsf{rf} \cup R_B)^+$ and hb is acyclic. That is, context relation R_B is added to hb. Atom′ is defined analogously with S_B added to at.

We say that $\mathcal{A}_B$, R_B and S_B are *consistent with B* if they act over variables in the set VS_B. In the rest of the paper we only consider consistent choices of $\mathcal{A}_B$, R_B, S_B. The *block-local executions* of B are then all executions $X \in [\![B, \mathcal{A}_B, R_B, S_B]\!]$.[2]

Example block-local execution. The left of Figure 4 shows a block-local execution for the code-block

$$\texttt{l1 := load(f); l2 := load(x)} \tag{5}$$

Here the set VS_B of accessed global variables is $\{\mathtt{f}, \mathtt{x}\}$, As before, we omit local variables to avoid clutter. the context action set $\mathcal{A}_B$ consists of the three stores, and R_B is denoted by dotted edges.

In this execution, both $\mathcal{A}_B$ and R_B affect the behaviour of the code-block. The following path is generated by R_B and the load of $\mathtt{f} = 1$:

$$\mathsf{store}(\mathtt{x}, 2) \xrightarrow{\mathsf{mo}} \mathsf{store}(\mathtt{x}, 1) \xrightarrow{R_B} \mathsf{store}(\mathtt{f}, 1) \xrightarrow{\mathsf{rf}} \mathsf{load}(\mathtt{f}, 1) \xrightarrow{\mathsf{sb}} \mathsf{load}(\mathtt{x}, 1)$$

Because hb includes sb, rf, and R_B, there is a transitive edge $\mathsf{store}(\mathtt{x}, 1) \xrightarrow{\mathsf{hb}} \mathsf{load}(\mathtt{x}, 1)$. The edge $\mathsf{store}(\mathtt{x}, 2) \xrightarrow{\mathsf{mo}} \mathsf{store}(\mathtt{x}, 1)$ is forced because the HBvsMO axiom prohibits mo from contradicting hb. Consequently, the Coherence axiom forces the code-block to read $\mathtt{x} = 1$.

[2] This definition relies on the fact that our language supports a fixed set of global variables, not dynamically allocated addressable memory (see §3.1). We believe that in the future our results can be extended to support dynamic memory. For this, the block-local construction would need to quantify over actions on all possible memory locations, not just the static variable set VS_B. The rest of our theory would remain the same, because C11-style models grant no special status to pointer values. Cutting down to a finite denotation, as in §5 below, would require some extra abstraction over memory – for example, a separation logic domain such as (Distefano et al. 2006).

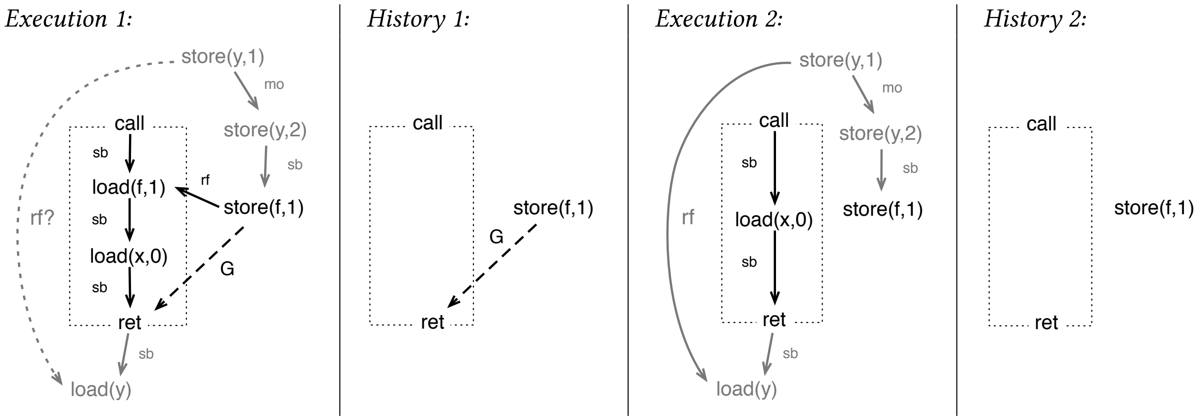


Fig. 5. Executions and histories illustrating the guarantee relation.

Histories. From any block-local execution X, its *history* summarises the interactions between the code-block and the context. Informally, the history records hb over context actions, call, and ret. More formally the history, written $\mathsf{hist}(X)$, is a pair $(\mathcal{A}, G)$ consisting of an action set A and *guarantee relation* $G \subseteq \mathcal{A} \times \mathcal{A}$. We write $\mathsf{contx}(X)$ to denote actions in $\mathcal{A}(X)$ outside the code-block, and define the history as follows:

- The action set $\mathcal{A}$ is the projection of X's action set to call, ret, and $\mathsf{contx}(X)$.
- The guarantee relation G is the projection of $\mathsf{hb}(X)$ to

$$(\mathsf{contx}(X) \times \mathsf{contx}(X)) \cup (\mathsf{contx}(X) \times \{\mathsf{ret}\}) \cup (\{\mathsf{call}\} \times \mathsf{contx}(X)) \qquad (6)$$

The guarantee summarises the code-block's effect on its context: it suffices to only track hb and ignore other relations. Note the guarantee definition is similar to the context relation R_B, definition (4). The difference is that call and ret are switched: this is because the guarantee represents hb edges generated by the code-block, while R_B represents the edges generated by the context. The right of Figure 4 shows the history corresponding to the block-local execution on the left.

To see the interactions captured by the guarantee, compare the block given in def. (5) with the block `l2:=load(x)`. These blocks have differing effects on the following syntactic context:

```
store(y,1); store(y,2); store(f,1)   ||   {-}; l3:=load(y)
```

For the two-load block embedded into this context, $\texttt{l1} = 1 \wedge \texttt{l3} = 1$ is not a possible post-state. For the single-load block, this post-state is permitted.[3]

In Figure 5, we give executions for both blocks embedded into this context. We draw the context actions that are not included into the history in grey. In these executions, the code block determines whether the load of `y` can read value 1 (represented by the edge labelled 'rf?'). In the first execution, the context load of `y` cannot read 1 because there is the path $\mathsf{store}(\texttt{y}, 1) \xrightarrow{\mathsf{mo}} \mathsf{store}(\texttt{y}, 2) \xrightarrow{\mathsf{hb}} \mathsf{load}(\texttt{y})$ which would contradict the Coherence axiom. In the second execution there is no such path and the load is permitted to read 1.

Our abstraction theorem hides the operations inside the block, but we must nonetheless record these kinds of hb effects on the context. In Execution 1, the Coherence violation is still visible if we only consider context operations, call, ret, and the guarantee G – i.e. the history. In Execution 2, the fact that the read is permitted is likewise visible from examining the history. Thus the guarantee, combined with the local variable post-states, capture the effect of the block on the context without recording the actions inside the block.

[3]We choose these post-states for exposition purposes – in fact these blocks are also distinguishable through local variable `l1` alone.

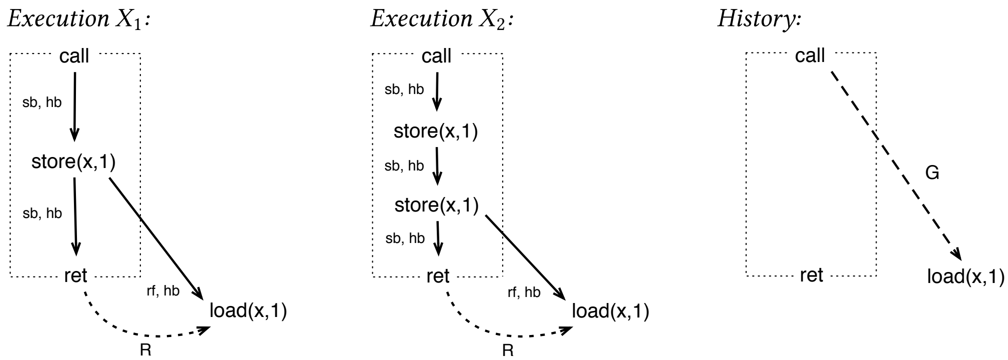


Fig. 6. History comparison for an example program transformation.

Comparing denotations. The denotation of a code-block B is the set of histories of block-local executions of B under each possible context, i.e. the set

$$\{\mathsf{hist}(X) \mid \exists \mathcal{A}_B, R_B, S_B.\, X \in [\![B, \mathcal{A}_B, R_B, S_B]\!]\}$$

To compare the denotations of two code-blocks, we first define a *refinement relation* on histories: $(\mathcal{A}_1, G_1) \sqsubseteq_{\mathsf{h}} (\mathcal{A}_2, G_2)$ holds iff $\mathcal{A}_1 = \mathcal{A}_2 \wedge G_2 \subseteq G_1$. The history $(\mathcal{A}_2, G_2)$ places fewer restrictions on the context than $(\mathcal{A}_1, G_1)$ – a weaker guarantee corresponds to more observable behaviours. For example in Figure 5, *History 1* $\sqsubseteq_{\mathsf{h}}$ *History 2* but not vice versa, which reflects the fact that History 1 rules out the read pattern discussed above.

We write $B_1 \sqsubseteq_{\mathsf{q}} B_2$ to state that the denotation of B_1 *refines* that of B_2. The subscript 'q' stands for the fact we *quantify* over both $\mathcal{A}$ and R_B. We define $\sqsubseteq_{\mathsf{q}}$ by lifting $\sqsubseteq_{\mathsf{h}}$:

$$B_1 \sqsubseteq_{\mathsf{q}} B_2 \overset{\Delta}{\iff} \forall \mathcal{A}, R, S.\, \forall X_1 \in [\![B_1, \mathcal{A}, R, S]\!].\, \exists X_2 \in [\![B_2, \mathcal{A}, R, S]\!].\, \mathsf{hist}(X_1) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X_2) \tag{7}$$

In other words, two code-blocks are related $B_1 \sqsubseteq_{\mathsf{q}} B_2$ if for every block-local execution of B_1, there is a corresponding execution of B_2 with a related history. Note that the corresponding history must be constructed under the same cut-down context $\mathcal{A}, R, S$.

Theorem 1 (Adequacy of $\sqsubseteq_{\mathsf{q}}$). $B_1 \sqsubseteq_{\mathsf{q}} B_2 \implies B_1 \preccurlyeq_{\mathsf{bl}} B_2$. (Proved in §B.)

Theorem 2 (Full abstraction of $\sqsubseteq_{\mathsf{q}}$). $B_1 \preccurlyeq_{\mathsf{bl}} B_2 \implies B_1 \sqsubseteq_{\mathsf{q}} B_2$. (Discussed in §8, proved in §E.)

As a corollary, a program transformation $B_2 \rightsquigarrow B_1$ is valid if and only if $B_1 \sqsubseteq_{\mathsf{q}} B_2$ holds.

Example transformation. We now apply our approach to a simple program transformation:

$$B_2\texttt{: store(x,l1); store(x,l1)} \rightsquigarrow B_1\texttt{: store(x,l1)}$$

To verify this transformation, we must show that $B_1 \sqsubseteq_{\mathsf{q}} B_2$. In Figure 6, we illustrate the necessary reasoning for a single block-local execution $X_1 \in [\![B_1, \mathcal{A}, R, S]\!]$, with a context action set $\mathcal{A}$ consisting of a single load `x = 1`, a context relation R relating ret to the laod, and an empty S relation. This choice of R forces the context load to read from the store in the block. We can exhibit an execution $X_2 \in [\![B_2, \mathcal{A}, R, S]\!]$ with a matching history by making the context load read from the final store in the block.

5 A FINITE DENOTATION

The approach above simplifies contexts by removing syntax and non-hb structure, but there are still infinitely many $\mathcal{A}/R/S$ contexts for any code-block. To solve this, we modify our denotation. This gives us finiteness in some cases, meaning we can automatically check transformations (see §6). However the 'cut' approach is no longer fully abstract. We modify our denotation as follows:

- We eliminate redundant block-local executions from the denotation by only considering those executions X that satisfy a predicate $\mathsf{cut}(X)$. Intuitively, the denotation must consider each *pattern* of context behaviour, but it need not consider every execution.
- We remove the quantification over context relation R from definition (7) by fixing it as $\emptyset$. In exchange, we extend the history with an extra component called a *deny*.

Before defining these steps in detail, we give the structure of our modified refinement $\sqsubseteq_{\mathsf{c}}$. In the definition, $\mathsf{hist}_{\mathsf{E}}(X)$ stands for the *extended history* of an execution X, and $\sqsubseteq_{\mathsf{E}}$ for refinement on extended histories.

$$B_1 \sqsubseteq_{\mathsf{c}} B_2 \;\stackrel{\Delta}{\Longleftrightarrow}\; \forall \mathcal{A}, S.\, \forall X_1 \in [\![B_1, \mathcal{A}, \emptyset, S]\!].\, \mathsf{cut}(X_1) \implies \exists X_2 \in [\![B_2, \mathcal{A}, \emptyset, S]\!].\, \mathsf{hist}_{\mathsf{E}}(X_1) \sqsubseteq_{\mathsf{E}} \mathsf{hist}_{\mathsf{E}}(X_2) \tag{8}$$

Theorem 3 (Adequacy of $\sqsubseteq_{\mathsf{c}}$). $B_1 \sqsubseteq_{\mathsf{c}} B_2 \implies B_1 \preccurlyeq_{\mathsf{bl}} B_2$. (Proved in §D.)

By *finiteness*, we mean that a code-block has a finite number of block-local executions satisfying cut. Because block-local executions are derived from pre-executions in the thread-local semantics, finiteness only holds if the set of the latter is finite. Note that we assume a finite domain of values in Val.

Theorem 4 (Finiteness). *If for any σ the set $\langle B, \sigma \rangle$ is finite, then so is* $\{X \mid \exists \mathcal{A}.\, X \in [\![B, \mathcal{A}, \emptyset, S]\!] \wedge \mathsf{cut}(X)\}$.

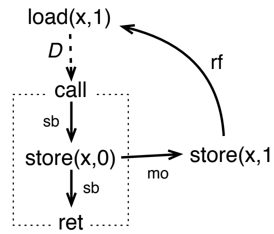
Cutting predicate. We first identify the actions in a block-local execution that are *visible*, meaning they directly affect the behaviour of the block. We write $\mathsf{code}(X)$ for the set of actions in X generated by the code-block. Visible actions belong to $\mathsf{code}(X)$, are read from $\mathsf{code}(X)$, or are read by $\mathsf{code}(X)$. In other words,

$$\mathsf{vis}(X) \;\stackrel{\Delta}{=}\; \mathsf{code}(X) \cup \{u \mid \exists v \in \mathsf{code}(X).\, u \xrightarrow{\mathsf{rf}} v \vee v \xrightarrow{\mathsf{rf}} u\}$$

We first define the predicate $\mathsf{cut}'(X)$ below. The overall predicate $\mathsf{cut}(X)$ extends this in order to keep LL-SC pairs together: it requires that, if $\mathsf{cut}'()$ permits one half of an LL-SC, the other is also permitted implicitly (for brevity we omit the formal definition of $\mathsf{cut}()$ in terms of $\mathsf{cut}'()$). The predicate $\mathsf{cut}'(X)$ is the conjunction of cutR for reads, and cutW for writes.

$$\begin{aligned}
\mathsf{cut}'(X) &\;\stackrel{\Delta}{\Longleftrightarrow}\; \mathsf{cutR}(X) \wedge \mathsf{cutW}(X) \\
\mathsf{cutR}(X) &\;\stackrel{\Delta}{\Longleftrightarrow}\; \mathsf{reads}(X) \subseteq \mathsf{vis}(X) \wedge \forall r_1, r_2 \in \mathsf{contx}(X).\, (r_1 \neq r_2 \Rightarrow \neg\exists w.\, w \xrightarrow{\mathsf{rf}} r_1 \wedge w \xrightarrow{\mathsf{rf}} r_2) \\
\mathsf{cutW}(X) &\;\stackrel{\Delta}{\Longleftrightarrow}\; \forall w_1, w_2 \in (\mathsf{contx}(X) \setminus \mathsf{vis}(X)).\, w_1 \xrightarrow{\mathsf{mo}} w_2 \Rightarrow \exists w_3 \in \mathsf{vis}(X).\, w_1 \xrightarrow{\mathsf{mo}} w_3 \xrightarrow{\mathsf{mo}} w_2
\end{aligned}$$

The predicate cutR requires that all reads are visible and that pairs of reads must read from distinct writes. In particular, this rules out multiple context reads all reading from the same write. Unlike reads, cutW permits writes that are not visible. However any two non-visible writes to a location must be separated in mo by a visible write. This still achieves finiteness (Theorem 4) because for a given pre-execution B, any two non-visible writes must be distinguished by a visible write, limiting their number. Preserving some of non-visible writes is required for our proof of adequacy for $\sqsubseteq_{\mathsf{c}}$ (Theorem 3).



Extended history (hist_E*).* The definition of $\sqsubseteq_\mathsf{c}$ removes the context relation R, which records the hb edges enforced by the context, and replaces it with a history component which records the hb edges that *cannot* be enforced due to the execution structure.

For example, consider the block-local execution to the right[4]. This individual execution represents a set of larger execution contexts, but it cannot be embedded into a context that generates the dashed edge D as a hb – to do so would violate the HBvsMO axiom. We represent such 'forbidden' edges D by a separate history component called a *deny*.

The *extended history* of an execution X, written $\mathsf{hist}_\mathsf{E}(X)$ is a triple $(\mathcal{A}, G, D)$, consisting of the familiar notions of action set $\mathcal{A}$ and guarantee $G \subseteq \mathcal{A} \times \mathcal{A}$, together with deny $D \subseteq \mathcal{A} \times \mathcal{A}$ as defined below:

$$D \stackrel{\Delta}{=} \{(u,v) \mid \mathsf{HBvsMO\text{-}d}(u,v) \vee \mathsf{Cohere\text{-}d}(u,v) \vee \mathsf{RFval\text{-}d}(u,v)\} \ \cap \\ \Big((\mathsf{contx}(X) \times \mathsf{contx}(X)) \cup (\mathsf{contx}(X) \times \{\mathsf{call}\}) \cup (\{\mathsf{ret}\} \times \mathsf{contx}(X))\Big)$$

Each of the predicates HBvsMO-d, Cohere-d, and RFval-d generates the deny for one validity axiom. In the diagrammatic definitions below, dashed edges represent the deny edge, and hb^* is the reflexive-transitive closure of hb:

$$\mathsf{HBvsMO\text{-}d}(u,v): \quad \exists w_1, w_2.\ \begin{tikzcd} w_1 \arrow[r, "\mathsf{hb}^*"] & u \arrow[r, dotted, "D"] & v \arrow[r, "\mathsf{hb}^*"] & w_2 \arrow[lll, bend left, "\mathsf{mo}"] \end{tikzcd}$$

$$\mathsf{Coherence\text{-}d}(u,v): \quad \begin{tikzcd} w_1 \arrow[r, "\mathsf{mo}"] \arrow[rrrr, bend right, "\mathsf{rf}"'] & w_2 \arrow[r, "\mathsf{hb}^*"] & u \arrow[r, dotted, "D"] & v \arrow[r, "\mathsf{hb}^*"] & r \end{tikzcd}$$

$$\mathsf{RFval\text{-}d}(u,v): \quad \exists w, r.\ \mathsf{gvar}(w) = \mathsf{gvar}(r) \wedge \neg\exists w'.\ w' \xrightarrow{\mathsf{rf}} r \wedge \ w \xrightarrow{\mathsf{hb}^*} u \overset{D}{\dashrightarrow} v \xrightarrow{\mathsf{hb}^*} r$$

One can think of a deny edge as an 'almost' violation of an axiom. For example, if $\mathsf{HBvsMO\text{-}d}(u,v)$ holds, then the context cannot generate an extra hb-edge $u \xrightarrow{\mathsf{hb}} v$ – to do so would violate HBvsMO.

Because deny edges represent constraints on the context, weakening the deny places fewer constraints, allowing more behaviours, so we compare them with relational inclusion:

$$(\mathcal{A}_2, G_2, D_2) \sqsubseteq_\mathsf{E} (\mathcal{A}_2, G_2, D_2) \ \stackrel{\Delta}{\iff} \ \mathcal{A}_1 = \mathcal{A}_2 \wedge G_2 \subseteq G_1 \wedge D_2 \subseteq D_1$$

This refinement on extended histories is used to define our refinement relation on blocks, $\sqsubseteq_\mathsf{c}$, def. (8).

Counter-example to full abstraction. Finiteness has a cost: $\sqsubseteq_\mathsf{c}$ is not fully abstract. To see this, consider blocks, B_1: `skip` and B_2: `load`(x). It is easy to see that $B_1 \sqsubseteq_\mathsf{q} B_2$ holds: the new load can read from either a hb-earlier write action, or the initialisation if none exists. Neither case introduces an extra guarantee edge.

However, $B_1 \sqsubseteq_\mathsf{c} B_2$ does not hold. If the context contains a write W, then the load can either read from it or the initialisation. The former generates a hb-edge in the history, while the latter generates a deny from RFval-d – thus history inclusion does not hold. In the quantified version of our approach, the context relation R distinguishes which origin for the load is allowed, which avoids the deny and thus avoids this problem.

[4]We use this execution for illustration, but in fact cut() would not allow the context load.

Transformation	Valid?	Time (s)
Introduction:		
$\texttt{skip} \rightsquigarrow \texttt{fence}$	✓	76
$\texttt{skip} \rightsquigarrow \texttt{load}(x)$	✓	429
$\texttt{skip} \rightsquigarrow l := \texttt{load}(x)$	x	18
$l := \texttt{load}(x) \rightsquigarrow l := \texttt{load}(x); \texttt{store}(x, l)$	x	72
$l := \texttt{load}(x) \rightsquigarrow l := \texttt{load}(y); l := \texttt{load}(x)$	✓	∞
$l := \texttt{load}(x) \rightsquigarrow l := \texttt{load}(x); l := \texttt{load}(x)$	✓	19851
$\texttt{store}(x, l) \rightsquigarrow \texttt{store}(x, l); \texttt{store}(x, l)$	x	136
$\texttt{fence} \rightsquigarrow \texttt{fence}; \texttt{fence}$	✓	248
Elimination:		
$\texttt{fence} \rightsquigarrow \texttt{skip}$	x	15
$l := \texttt{load}(x) \rightsquigarrow \texttt{skip}$	x	17
$l := \texttt{load}(x); \texttt{store}(x, l) \rightsquigarrow l := \texttt{load}(x)$	x	64
$l := \texttt{load}(x); l := \texttt{load}(x) \rightsquigarrow l := \texttt{load}(x)$	✓	2378
$\texttt{store}(x, l); l := \texttt{load}(x) \rightsquigarrow \texttt{store}(x, l)$	✓	9118
$\texttt{store}(x, m); \texttt{store}(x, l) \rightsquigarrow \texttt{store}(x, l)$	✓	24696
$\texttt{fence}; \texttt{fence} \rightsquigarrow \texttt{fence}$	✓	382
Exchange:		
$\texttt{fence}; l := \texttt{load}(x) \rightsquigarrow l := \texttt{load}(x); \texttt{fence}$	x	26
$\texttt{fence}; \texttt{store}(x, l) \rightsquigarrow \texttt{store}(x, l); \texttt{fence}$	x	50
$l := \texttt{load}(x); \texttt{fence} \rightsquigarrow \texttt{fence}; l := \texttt{load}(x)$	x	79
$\texttt{store}(x, l); \texttt{fence} \rightsquigarrow \texttt{fence}; \texttt{store}(x, l)$	x	145
$l := \texttt{load}(x); \texttt{store}(y, m) \rightsquigarrow \texttt{store}(y, m); l := \texttt{load}(x)$	x	28
$m := \texttt{load}(y); l := \texttt{load}(x) \rightsquigarrow l := \texttt{load}(x); m := \texttt{load}(y)$	x	118
$\texttt{store}(y, m); l := \texttt{load}(x) \rightsquigarrow l := \texttt{load}(x); \texttt{store}(y, m)$	✓	∞
$\texttt{store}(y, m); \texttt{store}(x, l) \rightsquigarrow \texttt{store}(x, l); \texttt{store}(y, m)$	x	641

Fig. 7. Results for our checking tool for a range of examples. ✓ and x denote whether the transformation satisfies $\sqsubseteq_c$. A time of ∞ denotes that Stellite timed out after 8 hours. Timings were produced on a 2.3GHz AMD Opteron processor (32 cores), with 128GB of physical memory, running Linux 3.13.0-88 and Java 1.8.0_91.

6 PROTOTYPE VERIFICATION TOOL

Stellite is our prototype tool that verifies transformations using the Alloy* model checker (Jackson 2012; Milicevic et al. 2015). Our tool converts an input transformation $B_2 \rightsquigarrow B_1$ into an Alloy* model encoding $B_1 \sqsubseteq_c B_2$. If the tool reports success, then the transformation is verified for unboundedly large syntactic contexts and executions.

An Alloy model consist of a collection of predicates on relations, and an instance of the model is a set of relations that satisfy the predicates. As previously noted in Wickerson et al. (2017), there is therefore a natural fit between Alloy models and axiomatic memory models. We use the higher-order Alloy* solver of Milicevic et al. (2015) because the standard Alloy solver cannot support the existential quantification on histories in the definition of $\sqsubseteq_c$.

The Alloy* solver is parameterised by the maximum size of the model it will examine. However, Stellite itself is not a bounded model checker. Our finiteness theorem for $\sqsubseteq_c$ (Theorem 4) means there is a bound on the size of cut-down context that needs to be considered to verify any given transformation.

Given a query $B_1 \sqsubseteq_c B_2$, the required context bound grows in proportion to the the number of internal actions on distinct locations in B_1. In our experiments we ran the tool with a model bound of 10, sufficient to give soundness for all the transformations we consider. Note that most of our example transformations do not require such a large bound, and execution times improve if it is reduced.

If a counter-example is discovered, the problematic execution and history can be viewed using the Alloy model visualiser, which has a similar appearance to the execution diagrams in this paper. As $\sqsubseteq_c$ is not fully abstract, this counter-example could of course be spurious.

Stellite currently supports transformations with atomic reads, writes, and fences. It does not yet support non-atomic accesses (see §7), LL-SC, or branching control-flow. We believe supporting the above features would not present fundamental difficulties, since the structure of the Alloy encoding would be similar. Despite the above limitations, our prototype demonstrates that our cut-down denotation can be used for automatic verification of important program transformations.

Experimental results. We have tested our tool on a range of different transformations. A table of experimental results is given in Figure 7. Many of our examples are derived from Vafeiadis et al. (2015) – we cover all their examples that fit into our tool's input language. Transformations of the sort that we check have led to real-world bugs in GCC (Morisset et al. 2013) and LLVM (Chakraborty and Vafeiadis 2016). Note that some transformations are invalid because of their effect on local variables, e.g. $\texttt{skip} \rightsquigarrow l := \texttt{load}(x)$. The closely related transformation $\texttt{skip} \rightsquigarrow \texttt{load}(x)$ throws away the result of the read, and is consequently valid.

Our tool takes significant time to verify some of the above examples, and two of the transformations cause the tool to time out. This is due to the complexity and non-determinism of the C11 model. In particular, our execution times are comparable to existing C++ model *simulators* such as cppmem when they run on a few lines of code (Batty et al. 2013). However, our tool is a sound transformation verifier, rather than a simulator, and thus solves a more difficult problem: transformations are verified for unboundedly large syntactic contexts and executions, rather than for a single execution.

When our tool times out, this of course does not establish validity for the transformation. However, as with bounded model checking, our experience is counter-examples are found at shallow positions in the search space.

7 TRANSFORMATIONS WITH NON-ATOMICS

We now extend our approach to *non-atomic* (i.e. unsynchronised) accesses. For C11 non-atomics, any concurrent read-write or write-write pair of actions to the same location is a *data race*, which causes the whole program to have undefined behaviour. Non-atomics are intended to enable sequential compiler optimisations that would otherwise be unsound in a concurrent context.

Memory model with non-atomics. Non-atomic loads and stores are added to the model by introducing new commands $\texttt{store}_{\mathsf{NA}}(x, l)$ and $l := \texttt{load}_{\mathsf{NA}}(x)$ and the corresponding kinds of actions: $\mathsf{store}_{\mathsf{NA}}, \mathsf{load}_{\mathsf{NA}} \in \mathsf{Kind}$. NA is set of all actions of these kinds. We partition global variables so that they are either only accessed by non-atomics, or by atomics. We do not support non-atomic LL-SC operations. Two new validity axioms ensure that non-atomics read from writes that happen before them, but not from stale writes:

- RFHBNA: $\forall w, r \in \mathsf{NA}.\ w \xrightarrow{\mathsf{rf}} r \implies w \xrightarrow{\mathsf{hb}} r$
- CoherNA: $\neg\exists w_1, w_2, r \in \mathsf{NA}.\ \ w_1 \xrightarrow{\mathsf{hb}} w_2 \xrightarrow{\mathsf{hb}} r \wedge w_1 \xrightarrow{\mathsf{rf}} r$

Modification order (mo) does not cover non-atomic accesses, and we change the definition of happens-before (hb), so that non-atomic loads do not add edges to it:

- HBdef: $\mathsf{hb} = (\mathsf{sb} \cup (\mathsf{rf} \cap \{(w, r) \mid w, r \notin \mathsf{NA}\}))^+$

Consider the MP program given in Figure 1. In Figure 3 we showed an execution that is forbidden by the Coherence axiom. This execution is forbidden because store of `f` to 1 is related to the load by a happens-before edge. If all of the actions in this program were instead made non-atomic, then no happens-before would be created, and by RFHBNA the loads of `f` and `x` would be forced to read the initialisation.

The most significant change to the model is the introduction of a *safety axiom*, data-race freedom (DRF). This forbids non-atomic read-write and write-write pairs that are unordered in hb:

$$\text{DRF:}\quad \forall u,v \in \mathcal{A}.\begin{pmatrix}\exists x.\, u \neq v \wedge u = (\mathsf{store}(x,\_)) \wedge \\ v \in \{(\mathsf{load}(x,\_)),(\mathsf{store}(x,\_))\}\end{pmatrix} \implies (u \xrightarrow{\mathsf{hb}} v \vee v \xrightarrow{\mathsf{hb}} u \vee u,v \notin \mathsf{NA})$$

We write $\mathsf{safe}(X)$ if an execution satisfies this axiom. Returning to MP, we see that there is a race between the load on each thread and the store on the other.

Let $[\![P]\!]^{\mathsf{NA}}_v$ be defined same way as $[\![P]\!]$ is in §3, def. (3). However, we add axioms RFHBNA and CoherNA and substitute the changed axiom HBdef. Then the semantics $[\![P]\!]$ of a program with non-atomics is:

$$[\![P]\!] \quad \stackrel{\Delta}{=} \quad \mathit{if}\ \forall X \in [\![P]\!]^{\mathsf{NA}}_v.\, \mathsf{safe}(X)\ \ \mathit{then}\ \ [\![P]\!]^{\mathsf{NA}}_v\ \ \mathit{else}\ \ \top$$

The undefined behaviour $\top$ subsumes all others, so any program observationally refines a racy program. Hence we modify our notion of observational refinement on whole programs:

$$P_1 \preccurlyeq^{\mathsf{NA}}_{\mathsf{pr}} P_2 \ \stackrel{\Delta}{\iff}\ (\mathsf{safe}(P_2) \implies (\mathsf{safe}(P_1) \wedge P_1 \preccurlyeq_{\mathsf{pr}} P_2))$$

This always holds when P_2 is unsafe; otherwise, it requires P_1 to preserve safety and observations to match. We define observational refinement on blocks, $\preccurlyeq^{\mathsf{NA}}_{\mathsf{bl}}$, by lifting $\preccurlyeq^{\mathsf{NA}}_{\mathsf{pr}}$ as per §2, def. (2).

Denotation with non-atomics. We define $\sqsubseteq^{\mathsf{NA}}_{\mathsf{q}}$, a refinement relation sensitive to non-atomics. We first introduce the *downclosure* $X^{\downarrow}$, the set of $(\mathsf{hb} \cup \mathsf{rf})^+$ prefixes of an execution X:

$$X^{\downarrow} \ \stackrel{\Delta}{=}\ \{X' \mid \exists \mathcal{A}.\, X' = X|_{\mathcal{A}} \wedge \forall (u,v) \in (\mathsf{hb}(X) \cup \mathsf{rf}(X))^+.\, (v \in \mathcal{A} \Rightarrow u \in \mathcal{A})\}$$

Here $X|_{\mathcal{A}}$ is the projection of the execution X to actions in $\mathcal{A}$. We lift the downclosure to sets of executions in the standard way. Now we define $B_1 \sqsubseteq^{\mathsf{NA}}_{\mathsf{q}} B_2$ as follows:

$$\begin{aligned} B_1 \sqsubseteq^{\mathsf{NA}}_{\mathsf{q}} B_2 \ \stackrel{\Delta}{\iff}\ & \forall \mathcal{A},R,S.\, \forall X_1 \in [\![B_1,\mathcal{A},R,S]\!]^{\mathsf{NA}}_v.\, \exists X_2 \in [\![B_2,\mathcal{A},R,S]\!]^{\mathsf{NA}}_v. \\ & (\mathsf{safe}(X_2) \implies \mathsf{safe}(X_1) \wedge \mathsf{hist}(X_1) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X_2)) \ \wedge \\ & (\neg\mathsf{safe}(X_2) \implies \exists X_2' \in (X_2)^{\downarrow}.\, \exists X_1' \in (X_1)^{\downarrow}.\, \neg\mathsf{safe}(X_2') \wedge \mathsf{hist}(X_1') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X_2')) \end{aligned}$$

In this definition, we case-split on whether the execution X_2 of block B_2 we chose is safe or unsafe.

If X_2 is safe, then the situation corresponds to $\sqsubseteq_{\mathsf{q}}$ (§4, def. (7)). In fact, if B_2 is certain to be safe, for example because it has no non-atomic accesses, the above definition is equivalent to $\sqsubseteq_{\mathsf{q}}$.

If X_2 is unsafe, then we can consider prefixes of histories, not full histories. Recall that X_2 represents the block portion of an execution of an unknown program $C(B_2)$. If we can ensure that $C(B_2)$ is unsafe, then all other programs observationally refine it, and the transformation will be sound by default. However, the context C is unknown – it is represented by the history of X_1. The context is only certain to exercise block-local execution X_2 if it is consistent with the history of X_1.

To show that the unsafety will occur in $C(B_2)$, we require an unsafe prefix of X_2 with a related history to a prefix of X_1. In other words, X_2 will behave consistently with X_1 *until it becomes unsafe.* This ensures that the unsafety in X_2 will in fact occur. After X_2 becomes unsafe, the two blocks can behave entirely differently, so we need not show that the complete histories of X_1 and X_2 are related.

Note that the prefixing in our definition of $\sqsubseteq^{\mathsf{NA}}_{\mathsf{q}}$ is required for full abstraction—but it would be adequate to always require *complete* executions with related histories.

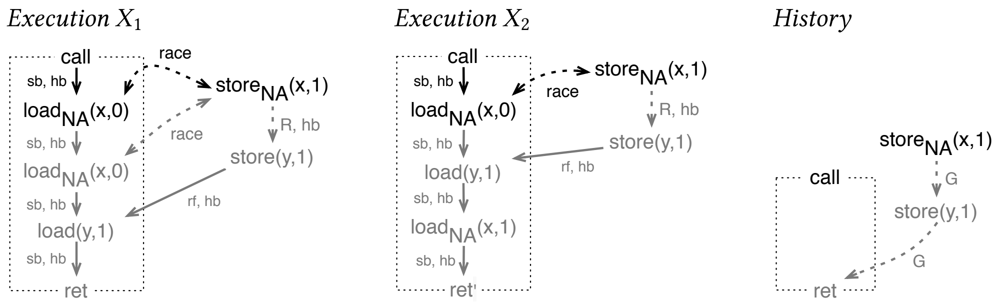


Fig. 8. History comparison for an NA-based program transformation

Theorem 5 (Adequacy of $\sqsubseteq_q^{NA}$). $B_1 \sqsubseteq_q^{NA} B_2 \implies B_1 \preccurlyeq_{bl}^{NA} B_2$. (Proved in §B.)

Theorem 6 (Full abstraction of $\sqsubseteq_q^{NA}$). $B_1 \preccurlyeq_{bl}^{NA} B_2 \implies B_1 \sqsubseteq_q^{NA} B_2$. (Discussed in §8, proved in §E.)

Validating a transformation. Consider the following *anti-roach-motel* transformation:

$$\begin{aligned} & B_2: \texttt{l1} := \texttt{load}_{\mathsf{NA}}(\texttt{x});\ \texttt{l2} := \texttt{load}(\texttt{y});\ \texttt{l3} := \texttt{load}_{\mathsf{NA}}(\texttt{x}) \\ \rightsquigarrow\quad & B_1: \texttt{l1} := \texttt{load}_{\mathsf{NA}}(\texttt{x});\ \texttt{l3} := \texttt{load}_{\mathsf{NA}}(\texttt{x});\ \texttt{l2} := \texttt{load}(\texttt{y}) \end{aligned}$$

To verify the transformation, we must establish that $B_1 \sqsubseteq_q^{NA} B_2$. In Figure 8 we illustrate the necessary reasoning for a single block-local execution $X_1 \in [\![B_1, \mathcal{A}, R, S]\!]$, with a context action set $\mathcal{A}$ consisting of a non-atomic store of x = 1 and an atomic store of y = 1, and a context relation R relating the store of x to the store of y: Note the data races between loads and a store over x.

To prove the result, we exhibit a corresponding unsafe execution $X_2 \in [\![B_2, \mathcal{A}, R, S]\!]_v$. The histories of the *complete* executions X_1 and X_2 differ in their return action. In X_2 the load of y takes the value of the context store, so CoherNA forces the second load of x to read from the context store of x. This changes the values of local variables recorded in ret′. However, because X_2 is unsafe, we can select a prefix X_2' which includes the race (we denote in grey the parts that we do not include). Similarly, we can select a prefix X_1' of X_1. We have that $\mathsf{hist}(X_1') = \mathsf{hist}(X_2')$ (shown in the figure), even though the histories $\mathsf{hist}(X_1)$ and $\mathsf{hist}(X_2)$ do not correspond.

Finite denotation with NA. We have also defined a finite variant of $\sqsubseteq_q^{NA}$, using the cutting strategy described in §5. For space reasons, we leave the details to §C.

8 FULL ABSTRACTION

The key idea of our proofs of full abstraction (Theorems 2 and 6) is to construct a special syntactic context that is sensitive to one particular history. Namely, given an execution X produced from a block B, this context C_X guarantees: (1) that X is the block portion of an execution of $C_X(B)$; and (2) for any block B', if $C_X(B')$ has a different block history from X, then this is visible in different observable behaviour. Therefore for any blocks that are distinguished by different histories, our construction can produce a program with different observable behaviour, establishing full abstraction.

Full abstraction and LL-SC. We note that our proof of full abstraction for the language with C11 non-atomics requires the language to also include LL-SC, not just C11's standard CAS: the former operation increases the observational power of the context. However, for the version of our approach *without* non-atomics (§4) CAS would be sufficient to prove full abstraction.

$$
\begin{aligned}
C_X &= \|_m\ (\mathtt{Racq}_m(\mathtt{Nrel}_m;\mathtt{check}(m)(\mathtt{Nacq}_m(\mathtt{Rrel}_m))))\\
\mathtt{Rrel}_m &= \mathtt{Rrel}_{\mathtt{ret}(m),v_1};\ \ldots;\mathtt{Rrel}_{\mathtt{ret}(m),v_n},\\
&\quad \text{where } \{v_1,\ldots,v_n\} = \{v \mid (\mathtt{ret}(m),v) \in R\}\\
\mathtt{Rrel}_{u,v} &= \mathtt{store}(h_{u,v},1)\\
\mathtt{Racq}_m(N) &= \mathtt{Racq}_{u_1,\mathtt{call}(m)}(\ldots\mathtt{Racq}_{u_n,\mathtt{call}(m)}(N)\ldots),\\
&\quad \text{where } \{u_1,\ldots,u_n\} = \{u \mid (u,\mathtt{call}(m)) \in R\}\\
\mathtt{Racq}_{u,v}(N) &= \mathtt{if}\ (\mathtt{load}(h_{u,v}))\ N\ \mathtt{else}\ \mathtt{store}(e,1)\\
\mathtt{Nrel}_m &= \mathtt{Nrel}_{\mathtt{call}(m),v_1};\ \ldots;\mathtt{Nrel}_{\mathtt{call}(m),v_n},\\
&\quad \text{where } \{v_1,\ldots,v_n\} = \{v \mid (\mathtt{call}(m),v) \in H\}\\
\mathtt{Nrel}_{u,v} &= \mathtt{store}(g_{u,v})\\
\mathtt{Nacq}_m(N) &= \mathtt{Nacq}_{u_1,\mathtt{ret}(m)}(\ldots\mathtt{Nacq}_{u_n,\mathtt{ret}(m)}(N)\ldots),\\
&\quad \text{where } \{u_1,\ldots,u_n\} = \{u \mid (u,\mathtt{ret}(m)) \in H\}\\
\mathtt{Nacq}_{u,v}(N) &= \mathtt{if}\ (\neg\mathtt{load}(g_{u,v}))\ N\ \mathtt{else}\ \mathtt{store}(e,1)
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{check}(\{-\}) = {}& \mathtt{l1} := 0; \mathtt{l2} := 0;\\
& \{-\};\\
& \mathtt{if}\ (\mathtt{l1} \neq 1)\ \{\mathtt{store}(\mathtt{e},1)\};\\
& \mathtt{if}\ (\mathtt{l2} \neq 1)\ \{\mathtt{store}(\mathtt{e},1)\}
\end{aligned}
$$

$$
\begin{aligned}
C_X = {}& \mathtt{Racq}_{\mathsf{store(x,2),call}}(\mathtt{Nrel}_{\mathsf{call,store(x,1)}};\\
&\quad \mathtt{Nrel}_{\mathsf{call,store(f,1)}};\\
&\quad \mathtt{check}(\{-\}))\\
&\| \ \mathtt{store}(\mathtt{x},2);\ \mathtt{Rrel}_{\mathsf{store(x,2),ret}};\\
&\quad \mathtt{Rrel}_{\mathsf{store(x,2),store(x,2)}};\\
&\| \ \mathtt{Racq}_{\mathsf{store(x,2),store(x,1)}}(\mathtt{store}(\mathtt{x},1);\\
&\quad \mathtt{Nacq}_{\mathsf{ret,store(x,1)}}(\mathtt{Rrel}_{\mathsf{store(x,1),store(f,1)}}))\\
&\| \ \mathtt{Racq}_{\mathsf{store(x,1),store(f,1)}}(\mathtt{store}(\mathtt{f},1);\\
&\quad \mathtt{Nacq}_{\mathsf{ret,store(f,1)}}(\mathtt{skip}))
\end{aligned}
$$

Fig. 9. *Left:* Definition of the construction of C_X for $X \in [\![B, \mathcal{A}, R]\!]$. We define H and check() in the text. The symbol m ranges over context actions $\mathcal{A}$ and a hole $\{-\}$. *Right:* Example of check() and the construction for the execution in Figure 4.

Proof structure. We now sketch the proof structure for full abstraction, for simplicity eliding the treatment of non-atomics and LL-SC. The full proof is given in §E. Assume $B_1 \preccurlyeq_{\mathsf{bl}} B_2$; we have to prove $B_1 \sqsubseteq_{\mathsf{q}} B_2$.

(1) Following the definition of $\sqsubseteq_{\mathsf{q}}$ (def. (7) in §4), consider arbitrary $\mathcal{A}$, R, and $X_1 \in [\![B_1, \mathcal{A}, R, \emptyset]\!]$ ($\emptyset$ is due to the fact that we ignore LL-SC).

(2) We use X_1 to construct the special context C_{X_1} (defined below). The context performs the actions specified by $\mathcal{A}$ and monitors executions to ensure that they do not significantly diverge from X_1, e.g., by checking that the values returned by context reads match those in $\mathcal{A}$. If C_{X_1} detects a mismatch with X_1, it writes to a special observable error variable $e \in \mathsf{OVar}$. The context C_{X_1} is constructed in such a way that for any code-block B' and any execution $Y \in [\![C_{X_1}(B')]\!]$ in which e is not written, the following three facts hold:
   (a) the actions of $\mathcal{A}$ appear in Y, and the actions by B' in Y transform local variables in a way consistent with the call and ret actions in X_1;
   (b) $\mathsf{hb}(Y)$ includes the edges in R;
   (c) $\mathsf{hb}(Y)$ is included in the guarantee of $\mathsf{hist}(X_1)$.

(3) We show that there is an execution $Z_1 \in [\![C_{X_1}(B_1)]\!]$ where the actions generated by B_1 match those in X_1, and where e is not written; the latter implies that the above properties (a), (b) and (c) hold of Z_1.

(4) Since $B_1 \preccurlyeq_{\mathsf{bl}} B_2$, by applying the definition of $\preccurlyeq_{\mathsf{bl}}$ (def. (2) in §4) to the special context C_{X_1}, we get an execution $Z_2 \in [\![C_{X_1}(B_2)]\!]$ where e is never written.

(5) By the construction of C_{X_1}, we know facts (a) and (b). Using this, we construct an execution $X_2 \in [\![B_2, \mathcal{A}, R, \emptyset]\!]$ where the actions generated by B_2 match those in Z_2 and the call and ret actions match those in X_1. Let $\mathsf{hist}(X_1) = (\mathcal{A}_1, G_1)$ and $X_2 = (\mathcal{A}_2, G_2)$. Using (a), we show $\mathcal{A}_1 = \mathcal{A}_2$ and using (c) we show $G_2 \subseteq G_1$. This establishes $\mathsf{hist}(X_1) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X_2)$, and by def. (7), gives $B_1 \sqsubseteq_{\mathsf{q}} B_2$.

Context construction. We next describe the construction of the context C_X for an execution $X \in [\![B, \mathcal{A}, R, \emptyset]\!]$ and argue that it satisfies the above properties (a)-(c). To illustrate the construction, we use the execution X in Figure 4, for the block B defined by def (5). The context C_X is defined on the left in Figure 9 and an application to the example is given on the right (for brevity, we use syntactic sugar that elides manipulations of local variables).

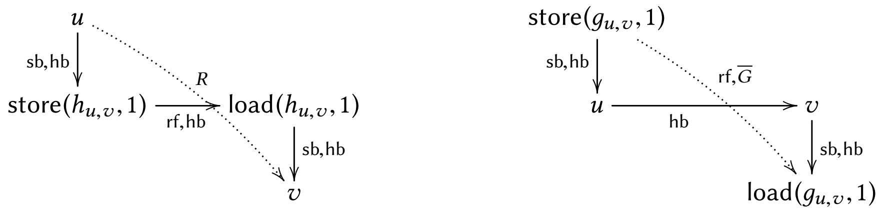


Fig. 10. Context construction execution shapes. *Left:* Shape enforcing edges in R. *Right:* shape prohibiting edges not in G.

The context C_X is a parallel composition of threads: one for the parameter code-block $\{-\}$, and one each action in $\mathcal{A}$—these are collectively ranged over by m in Figure 9. We introduce functions `call` and `ret` on the indices m, mapping $\{-\}$ to the call and ret actions in X, respectively, and acting as the identity otherwise. Recall that for our example execution X, the set $\mathcal{A}$ consists of the three writes outside the dashed rectangle. Our construction consists of several wrapper functions, introduced below.

(1) Innermost is `check`(m), which for brevity, we only describe informally. For a read or a write action $u \in \mathcal{A}$, `check`(u) executes the corresponding operation and, in the case of a read, compares the value read with the one specified by u. The command `check`$(\{-\})$ initialises local variables to the values specified by the call action in X, runs the code-block, $\{-\}$, and then compares the local variables with the values specified by the ret action in X. If there is a mismatch in the above cases, `check` writes to the error variable e. In this way, it ensures that property (a) holds in error-free executions. We give an example of `check` on the right of Figure 9.
(2) The wrappers $\texttt{Rrel}_m$ and $\texttt{Racq}_m$ ensure property (b). Recall the type (4) of R; in our running example from Figure 4, R is given by the dashed edges. Each $\texttt{Rrel}_m$ is built up of a sequence of invocations of $\texttt{Rrel}_{u,v}$, one for each edge $(u, v) \in R$ outgoing from $u = \texttt{ret}(m)$; the wrapper $\texttt{Racq}_m$ is constructed symmetrically. These wrappers use watchdog variables $h_{u,v}$ to create happens-before edges as in the MP test of Figure 1. Namely, $\texttt{Rrel}_{u,v}$ and $\texttt{Racq}_{u,v}$ respectively write to and read from the variable $h_{u,v}$. If $\texttt{Racq}_{u,v}$ does *not* read the value written by $\texttt{Rrel}_{u,v}$, then it writes to the error variable e. The invocation of $\texttt{Rrel}_{u,v}$ is sequenced after u and that of $\texttt{Racq}_{u,v}$ before v. Hence, any non-erroneous execution contains the shape on the left of Figure 10. This reproduces the required R edge (u, v) in the happens-before. In our running example, the edge (store(x, 2), call) $\in R$ is reproduced by the invocation of $\texttt{Racq}_{\text{store(x,2),call}}$ on the first thread and $\texttt{Rrel}_{\text{store(x,2),call}}$ on the second.
(3) The wrappers $\texttt{Nrel}_m$ and $\texttt{Nacq}_m$ ensure property (c), prohibiting new happens-before edges beyond those in the original guarantee G of hist(X). We identify pairs that must be monitored with the relation H: the edges of $\overline{G}$ matching the type (6) of a guarantee that are not already covered by the reverse of R. In our running example from Figure 4, the edges from $\overline{G}$ that we need to consider are (call, write(f, 1)) and (call, write$(x, 1)$). Each $\texttt{Nrel}_m$ is built up of a sequence of invocations of $\texttt{Nrel}_{u,v}$, one for each edge $(u, v) \in H$ outgoing from $u = \texttt{call}(m)$; the wrapper $\texttt{Racq}_m$ is constructed symmetrically. The above wrappers detect errant happens-before edges using watchdog variables $g_{u,v}$, again relying on the mechanics of the MP test of Figure 1. Namely, $\texttt{Nrel}_{u,v}$ and $\texttt{Nacq}_{u,v}$ respectively write to and read from a watchdog variable $g_{u,v}$. If $\texttt{Nacq}_{v,u}$ *does* read the value written by $\texttt{Nrel}_{v,u}$, then it writes to the error variable e. The invocation of $\texttt{Rrel}_{u,v}$ is sequenced *before* u and that of $\texttt{Racq}_{u,v}$ *after* v. Hence, if an execution includes a happens-before edge (u, v), then it contains the shape shown on the right of Figure 10 (omitting the write to the error location). Here the happens-before edge (u, v) and the RFval axiom (§3) force the read in $\texttt{Nacq}_{u,v}$ to read from the write in $\texttt{Nrel}_{u,v}$, leading to a write to e. Hence, a non-erroneous

execution does not contain errant happens-before edges. In our example the edge $(\mathsf{call}, \mathsf{store}(\mathtt{f}, 1)) \in \overline{G}$ is covered by the invocation of $\mathtt{Nrel}_{\mathsf{call},\mathsf{store}(\mathtt{f},1)}$ on the first thread and $\mathtt{Nacq}_{\mathsf{call},\mathsf{store}(\mathtt{f},1)}$ on the fourth.

9 RELATED WORK

Our approach builds on Batty et al. (2013), which generalises linearizability (Herlihy and Wing 1990) to the C11 memory model. Batty et al. represented interactions between a library and its clients by sets of histories consisting of a guarantee and a deny; we do the same for code-block and context. However, Batty et al. assumed *information hiding*, i.e., that the variables used by the library cannot be directly accessed by clients; we lift this assumption here. Also, we establish both adequacy and full abstraction, propose a finite denotation, and build an automated verification tool.

Our approach is broadly similar to the seminal concurrency semantics of Brookes (1996). In both cases, a code block is represented by a denotation capturing possible interactions with an abstracted context. In Brookes, denotations are sets of traces, consisting of sequences of global program states; context actions are represented by changes in these states. To handle the more complex axiomatic memory model, our denotation consists of sets of context actions and relations on them, with context actions explicitly represented as such. Also, in order to achieve full abstraction, Brookes assumes a powerful atomic `await()` instruction which blocks until the global state satisfies a predicate. Our full abstraction result does not require this: all our instructions operate on single locations, and our strongest instruction is LL-SC, which is commonly available on hardware platforms.

Brookes-like approaches have been applied to several relaxed models: operational hardware models (Burckhardt et al. 2010), TSO (Jagadeesan et al. 2012), and SC-DRF (Poetzl and Kroening 2016). Also, (Burckhardt et al. 2010; Poetzl and Kroening 2016) define tools for verifying program transformations. All three approaches are based on traces rather than partial orders, and are therefore not directly portable to C11-style axiomatic memory models. All three also target substantially stronger (i.e. more restrictive) relaxed models than ours.

Methods for verifying code transformations, either manually or using proof assistants, have been proposed for several relaxed models: TSO (Vafeiadis and Zappa Nardelli 2011; Ševčík et al. 2011, 2013), Java (Ševčík and Aspinall 2008) and C/C++ (Vafeiadis et al. 2015). These methods are non-compositional in the sense that verifying a transformation requires considering the trace set of the entire program — there is no abstraction of the context. We abstract both the sequential and concurrent context and thereby support automated verification. The above methods also model transformations as rewrites on program executions, whereas we treat them directly as modifications of program syntax; the latter corresponds more closely to actual compilers. Finally, these methods all require considerable proof effort; we build a tool that can verify transformations automatically.

There has also been various work on automatically verifying compiler optimisations under sequential consistency. For example, Alive (Lopes et al. 2015) and Peek (Mullen et al. 2016) are tools for verifying sequential peephole optimisations on LLVM and CompCert respectively. Vellvm is a formalization of the LLVM intermediate representation that has been used to formally verify sequential SSA-based optimisations (Zhao et al. 2013).

Our tool is a sound verification tool – that is, transformations are verified for all context and all executions of unbounded size. Several tools exist for testing (not verifying) program transformations on axiomatic memory models by searching for counter-examples to correctness, e.g., Lahav et al. (2016) for GCC and Chakraborty and Vafeiadis (2016) for LLVM. Alloy was used by Wickerson et al. (2017) in a testing tool for comparing memory models – this includes comparing language-level constructs with their compiled forms. Alloy has also been used in the MemSAT tool for simulation of the Java memory model (Torlak et al. 2010). Finally, our Alloy encoding of the memory model is similar to the input files for the Herd/Cat memory model simulator (Alglave et al. 2014).

10 CONCLUSIONS

We have proposed the first fully abstract denotational semantics for an axiomatic relaxed memory model, and using this, we have built the first tool capable of automatically verifying program transformation on such a model.

The key technical challenge of our work is that axiomatic models are defined in a global non-compositional style. We have shown that it is possible to recover a powerful form of compositionality that can be applied to prove useful properties of relaxed code.

Our theory lays the groundwork for further research into the properties of axiomatic models. In particular, our definition of the denotation as a set of histories and our context reduction techniques should be portable to other axiomatic models based on happens-before, such as those for hardware (Alglave et al. 2014) and distributed systems (Burckhardt et al. 2014). Using our techniques, we are confident that further sound verification tools can be developed built based on bounded model-checking techniques. We are also hopeful that our work will feed into memory-model design, which is often motivated by support for key compiler transformations.

ACKNOWLEDGMENTS

Thanks to Jeremy Jacob, and John Wickerson, for comments and suggestions. Dodds is supported by a Royal Society Industrial Fellowship. Batty is supported by a Lloyds Register Foundation and Royal Academy of Engineering Research Fellowship.

REFERENCES

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), Article 7, 74 pages. DOI: http://dx.doi.org/10.1145/2627752

James H. Anderson and Mark Moir. 1995. Universal Constructions for Multi-object Operations. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, NY, USA, 184–193. DOI: http://dx.doi.org/10.1145/224964.224985

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 235–248. DOI: http://dx.doi.org/10.1145/2429069.2429099

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Proceedings*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–307. DOI: http://dx.doi.org/10.1007/978-3-662-46669-8_12

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 55–66. DOI: http://dx.doi.org/10.1145/1926385.1926394

Stephen Brookes. 1996. Full Abstraction for a Shared-Variable Parallel Language. *Information and Computation* 127, 2 (1996), 145 – 163. DOI: http://dx.doi.org/10.1006/inco.1996.0056

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 271–284. DOI: http://dx.doi.org/10.1145/2535838.2535848

Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. 2010. Verifying Local Transformations on Relaxed Memory Models. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10/ETAPS'10)*. Springer-Verlag, Berlin, Heidelberg, 104–123. DOI: http://dx.doi.org/10.1007/978-3-642-11970-5_7

Soham Chakraborty and Viktor Vafeiadis. 2016. Validating Optimizations of Concurrent C/C++ Programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 216–226. DOI: http://dx.doi.org/10.1145/2854038.2854051

Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2006. *A Local Shape Analysis Based on Separation Logic*. Springer Berlin Heidelberg, Berlin, Heidelberg, 287–302. DOI: http://dx.doi.org/10.1007/11691372_19

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. DOI: http://dx.doi.org/10.1145/78969.78972

Daniel Jackson. 2012. *Software Abstractions – Logic, Language, and Analysis* (revised ed.). MIT Press.

Radha Jagadeesan, Gustavo Petri, and James Riely. 2012. Brookes is Relaxed, Almost!. In *Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures (FOSSACS'12)*. Springer-Verlag, Berlin, Heidelberg, 180–194. DOI: http://dx.doi.org/10.1007/978-3-642-28729-9_12

Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. ACM, New York, NY, USA, 759–767. DOI: http://dx.doi.org/10.1145/2933575.2934536

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 175–189. DOI: http://dx.doi.org/10.1145/3009837.3009850

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-acquire Consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 649–662. DOI: http://dx.doi.org/10.1145/2837614.2837643

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI 2017*.

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 22–32. DOI: http://dx.doi.org/10.1145/2737924.2737965

Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-purpose Higher-order Relational Constraint Solver. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 609–619. http://dl.acm.org/citation.cfm?id=2818754.2818829

Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 187–196. DOI: http://dx.doi.org/10.1145/2491956.2491967

Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified Peephole Optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 448–461. DOI: http://dx.doi.org/10.1145/2908080.2908109

Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 622–633. DOI: http://dx.doi.org/10.1145/2837614.2837616

Daniel Poetzl and Daniel Kroening. 2016. Formalizing and Checking Thread Refinement for Data-Race-Free Execution Models. In *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. Springer-Verlag New York, Inc., New York, NY, USA, 515–530. DOI: http://dx.doi.org/10.1007/978-3-662-49674-9_30

The C++ Standards Committee. 2011. *Programming Languages — C++*. ISO/IEC JTC1 SC22 WG21.

Emina Torlak, Mandana Vaziri, and Julian Dolby. 2010. MemSAT: Checking Axiomatic Specifications of Memory Models. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 341–350. DOI: http://dx.doi.org/10.1145/1806596.1806635

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 209–220. DOI: http://dx.doi.org/10.1145/2676726.2676995

Viktor Vafeiadis and Francesco Zappa Nardelli. 2011. Verifying Fence Elimination Optimisations. In *Proceedings of the 18th International Conference on Static Analysis (SAS'11)*. Springer-Verlag, Berlin, Heidelberg, 146–162. http://dl.acm.org/citation.cfm?id=2041552.2041566

Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming (ECOOP '08)*. Springer-Verlag, Berlin, Heidelberg, 27–51. DOI: http://dx.doi.org/10.1007/978-3-540-70592-5_3

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-memory Concurrency and Verified Compilation. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 43–54. DOI: http://dx.doi.org/10.1145/1926385.1926393

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (June 2013), 50 pages. DOI: http://dx.doi.org/10.1145/2487241.2487248

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 190–204. DOI: http://dx.doi.org/10.1145/3009837.3009838

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-based Optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 175–186. DOI: http://dx.doi.org/10.1145/2491956.2462164

A COLLECTED DEFINITIONS

Execution observational refinement.

$$X \preccurlyeq_{\mathsf{ex}} Y \quad \overset{\Delta}{\iff} \quad \mathcal{A}(X|_{\mathsf{OVar}}) = \mathcal{A}(Y|_{\mathsf{OVar}}) \ \land \ \mathsf{hb}(Y|_{\mathsf{OVar}}) \subseteq \mathsf{hb}(X|_{\mathsf{OVar}})$$

Program observational refinement.

$$P_1 \preccurlyeq_{\mathsf{pr}} P_2 \overset{\Delta}{\iff} \forall X_1 \in [\![P_1]\!].\, \exists X_2 \in [\![P_2]\!].\, X_1 \preccurlyeq_{\mathsf{ex}} X_2$$

Program observational refinement with NA.

$$P_1 \preccurlyeq^{\mathsf{NA}}_{\mathsf{pr}} P_2 \overset{\Delta}{\iff} (\mathsf{safe}(P_2) \implies \mathsf{safe}(P_1) \land P_1 \preccurlyeq_{\mathsf{pr}} P_2)$$

Block observational refinement.

$$B_1 \preccurlyeq_{\mathsf{bl}} B_2 \quad \overset{\Delta}{\iff} \quad \forall C.\ C(B_1) \preccurlyeq_{\mathsf{pr}} C(B_2)$$

History abstraction.

$$(\mathcal{A}_1, G_1) \sqsubseteq_{\mathsf{h}} (\mathcal{A}_2, G_2) \quad \overset{\Delta}{\iff} \quad \mathcal{A}_1 = \mathcal{A}_2 \land G_2 \subseteq G_1$$

Quantified abstraction.

$$B_1 \sqsubseteq_{\mathsf{q}} B_2 \quad \overset{\Delta}{\iff} \quad \forall \mathcal{A}, R, S.\, \forall X_1 \in [\![B_1, \mathcal{A}, R, S]\!].\, \exists X_2 \in [\![B_2, \mathcal{A}, R, S]\!].\, \mathsf{hist}(X_1) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X_2)$$

Extended history abstraction.

$$(\mathcal{A}_2, G_2, D_2) \sqsubseteq_{\mathsf{E}} (\mathcal{A}_2, G_2, D_2) \overset{\Delta}{\iff} \mathcal{A}_1 = \mathcal{A}_2 \land G_2 \subseteq G_1 \land D_2 \subseteq D_1$$

Cut abstraction.

$$B_1 \sqsubseteq_{\mathsf{c}} B_2 \quad \overset{\Delta}{\iff} \quad \forall \mathcal{A}, S.\, \forall X_1 \in [\![B_1, \mathcal{A}, \emptyset, S]\!].\, \mathsf{cut}(X_1) \implies \exists X_2 \in [\![B_2, \mathcal{A}, \emptyset, S]\!].\, \mathsf{hist}_{\mathsf{E}}(X_1) \sqsubseteq_{\mathsf{E}} \mathsf{hist}_{\mathsf{E}}(X_2)$$

Cut predicates.

$$\begin{aligned}
\mathsf{vis}(X) &\overset{\Delta}{\iff} \mathsf{code}(X) \cup \{u \mid \exists v \in \mathsf{code}(X).\, u \xrightarrow{\mathsf{rf}} v \lor v \xrightarrow{\mathsf{rf}} u\} \\
\mathsf{cut}'(X) &\overset{\Delta}{\iff} \mathsf{cutR}(X) \land \mathsf{cutW}(X) \\
\mathsf{cutR}(X) &\overset{\Delta}{\iff} \mathsf{reads}(X) \subseteq \mathsf{vis}(X) \land \forall r_1, r_2 \in \mathsf{contx}(X).\, r_1 \neq r_2 \implies \neg\exists w.\, w \xrightarrow{\mathsf{rf}} r_1 \land w \xrightarrow{\mathsf{rf}} r_2 \\
\mathsf{cutW}(X) &\overset{\Delta}{\iff} \forall w_1, w_2 \in (\mathsf{contx}(X) \setminus \mathsf{vis}(X)).\, w_1 \xrightarrow{\mathsf{mo}} w_2 \implies \exists w_3 \in \mathsf{vis}(X).\, w_1 \xrightarrow{\mathsf{mo}} w_3 \xrightarrow{\mathsf{mo}} w_2
\end{aligned}$$

Execution downclosure.

$$X^{\downarrow} \overset{\Delta}{=} \{X' \mid \exists \mathcal{A}.\, X' = X|_{\mathcal{A}} \land \forall (a, a') \in (\mathsf{hb}(X) \cup \mathsf{rf}(X))^{+}.\, a' \in \mathcal{A} \Rightarrow a \in \mathcal{A}\}$$

Quantified abstraction with NA.

$$\begin{aligned}
B_1 \sqsubseteq^{\mathsf{NA}}_{\mathsf{q}} B_2 \overset{\Delta}{\iff}\ & \forall \mathcal{A}, R, S.\, \forall X_1 \in [\![B_1, \mathcal{A}, R, S]\!]^{\mathsf{NA}}_{v}.\, \exists X_2 \in [\![B_2, \mathcal{A}, R, S]\!]^{\mathsf{NA}}_{v}. \\
& (\mathsf{safe}(X_2) \implies \mathsf{safe}(X_1) \land \mathsf{hist}(X_1) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X_2)) \ \land \\
& (\neg\mathsf{safe}(X_2) \implies \exists X_2' \in (X_2)^{\downarrow}.\, \exists X_1' \in (X_1)^{\downarrow}.\, \neg\mathsf{safe}(X_2') \land \mathsf{hist}(X_1') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X_2'))
\end{aligned}$$

B PROOF OF THEOREM 1

We now prove adequacy of $\sqsubseteq_{\mathsf{q}}^{\mathsf{NA}}$. As $\sqsubseteq_{\mathsf{q}}^{\mathsf{NA}} \implies \sqsubseteq_{\mathsf{q}}$, this suffices to prove adequacy of $\sqsubseteq_{\mathsf{q}}$. Our proof need several auxiliary notions:

- $\mathsf{codeE}(X)$ is the projection of an execution X to actions in $(\mathsf{codeE}(X) \cup \mathsf{interf}(X) \cup \{\mathsf{call}, \mathsf{ret}\})$.
- The *interface actions* are actions on variables in VS_B that occur in the context. These are context actions that can affect the behaviour of the code-block. We write $\mathsf{interf}(X)$ for this set.
- $\mathsf{contxE}(X)$ is the projection of an execution X to the context. This is a more complex projection than $\mathsf{codeE}(X)$ because it removes mo and rf over actions in $\mathsf{interf}(X)$. Let $\mathcal{I} = \mathsf{contx}(X) \cup \{\mathsf{call}, \mathsf{ret}\}$ and $\mathcal{C} = \mathsf{contx}(X) \setminus \mathsf{interf}(X)$. Then

$$\mathsf{contxE}(X) = (A(X)|_{\mathcal{I}}, \mathsf{hb}(X)|_{\mathcal{I}}, \mathsf{sb}(X)|_{\mathcal{I}}, \mathsf{mo}(X)|_{\mathcal{C}}, \mathsf{rf}(X)|_{\mathcal{C}})$$

- $\mathsf{hbC}(X)$ is the context-side projection of hb to interface actions. In other words, the projection of $\mathsf{hb}(X)$ to pairs in:

$$(\mathsf{interf}(X) \times \mathsf{interf}(X)) \cup (\mathsf{interf}(X) \times \{\mathsf{call}\}) \cup (\{\mathsf{ret}\} \times \mathsf{interf}(X))$$

- $\mathsf{atC}(X)$ is the context-side projection of at to context actions: i.e. the projection of $\mathsf{at}(X)$ to pairs in $(\mathsf{interf}(X) \times \mathsf{interf}(X))$.
- $[\![C, R, S]\!]_v$ is the *context-local* execution of a single-hole context C – this is an analogous notion to the block-local execution, except that rf and mo are not generated for the interface. Here R is a relation representing dependencies in hb arising from the code and S represents code at edges. An execution X is in this set iff:
  - R is a code-side relation on interface actions $\mathsf{interf}(X)$:

$$R \subseteq (\mathsf{interf}(X) \times \mathsf{interf}(X)) \cup (\mathsf{interf}(X) \times \{\mathsf{ret}\}) \cup (\{\mathsf{call}\} \times \mathsf{interf}(X))$$

  - S is a code-side relation on interface actions $\mathsf{interf}(X)$:

$$S \subseteq (\mathsf{interf}(X) \times \mathsf{interf}(X))$$

  - The execution satisfies the thread-local semantics:

$$(A(X), \mathsf{sb}(X)) \in \langle C \rangle$$

    We assume that a singleton hole has the following thread-local semantics:

$$\langle \{-\}, \sigma \rangle \quad \triangleq \quad \{(\{c, r\}, \{c \to r\}, \sigma') \mid c = \mathsf{call}(\sigma) \land r = \mathsf{ret}(\sigma')\}$$

  - X satisfies HBDEF', ATOM′, ACYCLICITY, RFWF, HBVSMO, COHERENCE, RFHBNA, COHERNA.
  - The projection $X|_{\mathsf{contxE}(X)}$ satisfies RFVAL, MOWF. mo and rf are disjoint from actions in $\mathsf{interf}(X)$.

  We sometimes write $[\![C]\!]_v$ to stand for $[\![C, \emptyset, \emptyset]\!]$, i.e. the set of context-local executions with empty code-side relations.

LEMMA 7 (DECOMPOSITION). *Assume* $X \in [\![C(B)]\!]_v$*, and no there are no* at *edges in* C *spanning* B*, nor any between the actions of* B *and* C*. Then* $\mathsf{codeE}(X) \in [\![B, \mathsf{interf}(X), \mathsf{hbC}(X), \mathsf{atC}(X)]\!]_v$ *and* $\mathsf{contxE}(X) \in [\![C, \mathsf{hbL}(X), \mathsf{atL}(X)]\!]_v$.

PROOF (CODE). We have several proof obligations.

- $\mathsf{hbC}(X)$ and $\mathsf{atC}(X)$ are context-side relations on interface actions (trivial by definition).
- $(\mathsf{codeE}(\mathsf{codeE}(X)), \mathsf{sb}(\mathsf{codeE}(X))) \in \langle B \rangle$, i.e. the execution satisfies the thread-local semantics.
- The actions in $\mathsf{codeE}(\mathsf{codeE}(X))$ are in between a call / ret pair in sb. We assume we can introduce call / ret freely to satisfy this requirement.
- $\mathsf{codeE}(X)$ satisfies the validity axioms for a block-local execution – note that this replaces HBDEF with HBDEF', and ATOM with ATOM′.

For the first obligation, we argue inductively over the structure of C. First assume that $C = \{-\}$, i.e. C consists only of a hole. In this case the result holds immediately from the thread-local semantics. For the inductive case, assume C is a composite one-hole context, e.g. $C_1; C_2(-)$ / $C_1(-); C_2$ / $C_1 \| C_2(-)$ / etc.

For the fourth obligation, we prove $\mathsf{codeE}(X)$ satisfies the validity axioms by arguing in turn about each. Assume the following shorthand:

$$\mathsf{codeE}(X) \quad = \quad (A(l), \mathsf{hb}(l), \mathsf{at}(l)\mathsf{sb}(l), \mathsf{mo}(l), \mathsf{rf}(l))$$

HBdef': Let $R = \mathsf{hbC}(X)$. Now prove in both directions:

$$(a,b) \in \mathsf{hb}(l) \quad \Longrightarrow \quad (a,b) \in (\mathsf{sb}(l) \cup \mathsf{rf}_{\mathsf{AT}}(l) \cup R)^+ \tag{9}$$

$$(a,b) \in (\mathsf{sb}(l) \cup \mathsf{rf}_{\mathsf{AT}}(l) \cup R)^+ \quad \Longrightarrow \quad (a,b) \in \mathsf{hb}(l) \tag{10}$$

For the first case, any (a,b) in $\mathsf{hb}(l)$ must have code or interface actions at both ends, and must have originated from a path $(a,b) \in (\mathsf{sb}(X) \cup \mathsf{rf}_{\mathsf{AT}}(X))^+$. By construction, there are no rf-edges between $\mathsf{codeE}(X)$ and $\mathsf{contxE}(X)$. Therefore, portions of the path which stray into the context must enter and leave through call, ret, or actions in $\mathsf{interf}(X)$. These portions of the path will be summarised by $\mathsf{hbC}(X)$. As a result, for any such path, there must be an equivalent path $(a,b) \in (\mathsf{sb}(l) \cup \mathsf{rf}_{\mathsf{AT}}(l) \cup \mathsf{hbC}(X))^+$.

For the second case, we make a similar argument. For any pair $(c,d) \in \mathsf{hbC}(X)$, there must be a path $(c,d) \in (\mathsf{sb}(X) \cup \mathsf{rf}_{\mathsf{AT}}(X))$. As a consequence, for any (a,b) in $(\mathsf{sb}(l) \cup \mathsf{rf}(l) \cup \mathsf{hbC}(X))^+$, there must be a path $(a,b) \in (\mathsf{sb}(X) \cup \mathsf{rf}_{\mathsf{NA}}(X))$. Thus $(a,b) \in \mathsf{hb}(X)$. As $\mathsf{hb}(l)$ is a projection of $\mathsf{hb}(X)$, this completes the proof.

Atom′, AcyclicityRFwf, MOwf, Coherence, RFHBNA, CoherNA: all hold immediately by the fact that $\mathsf{codeE}(X)$ is a projection of X.

RFval: holds because $\mathsf{code}(X)$ contains exactly the actions in X that are on locations $a \in \mathsf{gv}_B$. Therefore, the projection cannot remove the origin write for a read.

□

Proof (context). Similar argument to the code. □

Lemma 8 (Completion lemma). *Let X be an execution. If $\mathsf{valid}(X)$ and $(A(X), \mathsf{sb}(X)) \in \langle Q \rangle^{\downarrow}$, then $X \in [\![Q]\!]_v^{\downarrow}$.*

Proof. We require the existence of a $Y \in [\![Q]\!]_v$ such that $X \in Y^{\downarrow}$. To prove this, we iteratively extend X by adding sb-final actions, and show that the new execution can in each case be made valid. As all executions are finite, this proves the result.

Assume the current execution is X_i. We choose an $A(X_{i+1})$ and $\mathsf{sb}(X_{i+1})$ such that the new execution is extended by a single sb-final action, and that $(A(X_{i+1}), \mathsf{sb}(X_{i+1}) \in \langle Q \rangle^{\downarrow}$. We now need to show that we can construct a valid X_{i+1}.

Case-split on the type of the new action. Non-atomics read from their immediate hb predecessor, or the init value if none exists. Atomic reads read from the end of mo, and writes can be added to the end of mo. Compare-and-swaps read from the end of mo. All of these cases preserve the validity axioms.

Note that if the new action is a read, we may need to fix its value depending on an earlier write. This depends on the property of *receptiveness* – given a prefix $(A, \mathsf{sb}) \in \langle Q' \rangle$ and a read r that is sb-maximal, any value can be given to the read. This property follows from the thread-local semantics: the only tricky cases are conditionals and LL-SC, where receptiveness is guaranteed by the fact that any possible value is represented in the set of possible reads.

□

Lemma 9 (Safety completion). *Let X, Y be valid executions. $\neg\mathsf{safe}(X)$ and $X \in Y^{\downarrow}$ implies $\neg\mathsf{safe}(Y)$.*

PROOF. Prove the contrapositive: $\mathsf{safe}(Y) \implies \mathsf{safe}(X)$. This holds immediately from the fact that in a safe execution, potentially racy actions must be related in hb. □

LEMMA 10 (COMPOSITION). *Let X and Y be executions such that $X \in [\![B, \mathcal{A}, \mathsf{hbC}(Y), \mathsf{atC}(Y)]\!]_v^{\downarrow}$ and $Y \in [\![C, \mathcal{A}, R', S']\!]_v^{\downarrow}$ with no LL/SC pairs crossing the block boundary in each case, with $\mathsf{hist}(Y) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X)$ and with $\mathsf{atL}(X) = S'$. Then there exists an execution Z such that $Z \in [\![C(B)]\!]_v^{\downarrow}$. Furthermore:*

- *If $\neg\mathsf{safe}(X)$ or $\neg\mathsf{safe}(Y)$, then $\neg\mathsf{safe}(Z)$.*
- *If $\mathsf{safe}(X)$, $\mathsf{safe}(Y)$, and $\mathsf{safe}(Z)$, and $X \in [\![B, \mathcal{A}, \mathsf{hbC}(Y), \mathsf{atC}(Y)]\!]_v$ and $Y \in [\![C, \mathcal{A}, R', S']\!]_v$, then $Z \in [\![C(B)]\!]_v$ and $\mathsf{contxE}(Y) \preccurlyeq_{\mathsf{ex}} \mathsf{contxE}(Z)$.*

PROOF. We begin by defining Z. Taking each term of the execution in turn:

- The action set $A(Z)$ is the union of the two action sets $A(X)$ and $A(Y)$, merging call, return and interface actions.
- $\mathsf{sb}(Z) = (\mathsf{sb}(X) \cup \mathsf{sb}(Y))^+$.
- $\mathsf{mo}_Z = (\mathsf{mo}(X) \cup \mathsf{mo}(Y))$ – as the two mo relations are disjoint, no transitive closure is needed.
- $\mathsf{rf}_Z = (\mathsf{rf}(X) \cup \mathsf{rf}(Y))$ – likewise.
- $\mathsf{hb}_Z = (\mathsf{sb}(Z) \cup \mathsf{rf}_{\mathsf{AT}}(Z))^+$, ie, according to HBDEF.
- $\mathsf{at}_Z = \mathsf{at}(X) \cup \mathsf{at}(Y)$.

We first need to show that $Z \in [\![C(L)]\!]_v^{\downarrow}$. To do this we use the completion lemma: thus our proof obligations are $(A(Z), \mathsf{sb}(Z)) \in \langle C(B_2)\rangle^{\downarrow}$ and $\mathsf{valid}(Z)$.

We observe that that $(A(Z), \mathsf{sb}(Z)) \in \langle C(B_2)\rangle^{\downarrow}$ is obvious from the thread-local semantics.

Next prove that $\mathsf{valid}(Z)$. HBDEF holds by construction. RFWF, RFVAL, MOWF, RBDEF are true trivially as for each variable, validity is checked solely in either the code or context. This leaves ACYCLICITY, HBVSMO, COHERENCE, COHERNA and ATOM. (RFHBNA needs to be treated specially – see below).

- For ACYCLICITY, a violation would correspond to a path in $(\mathsf{sb}(Z) \cup \mathsf{rf}_{\mathsf{AT}}(Z) \cup \mathsf{rf}_{\mathsf{NA}})^+$. As this path cannot appear in either X or Y, it must cross between the two: each point where it does so must be an interface action or call / return. As a result, a corresponding violation can be constructed in X.

  Call-to-return paths are in $(\mathsf{sb}(X) \cup \mathsf{rf}_{\mathsf{AT}}(X))^+ \cup \mathsf{rf}_{\mathsf{NA}}(X))^+$. Conversely, return-to-call paths are in $(\mathsf{sb}(Y) \cup \mathsf{rf}_{\mathsf{AT}}(Y) \cup \mathsf{rf}_{\mathsf{NA}}(Y))^+$. As Y satisfies RFHBNA, $\mathsf{rf}_{\mathsf{NA}}(Y) \in \mathsf{hb}(Y)$. Thus the return-to-call portions of the path are in $\mathsf{hbC}(Y)$. This contradicts the assumption that X satisfies ACYCLICITY.
- For HBVSMO, a violation consists of a write pair w_1, w_2 such that $(w_1, w_2) \in \mathsf{hb}(Z)$ and $(w_2, w_1) \in \mathsf{mo}(Z)$. As mo is partitioned between code and context, either both writes are in X or both in Y. By assumption, the violation is not solely in X or Y, so the path from w_1 to w_2 in $(\mathsf{sb} \cup \mathsf{rf}_{\mathsf{AT}})^+$ must contain a sequence of interface actions or call / return.
  1. If the writes are in X, then mo is replicated immediately. The block-local portions of the path are in $(\mathsf{sb}(X) \cup \mathsf{rf}_{\mathsf{AT}}(X))^+$, while the context-local portions are in $\mathsf{hbC}(X)$. Thus we can replicate the violation.
  2. If the writes are in Y, we can use a similar argument. However, we also appeal to the fact that $\mathsf{hist}(Y) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X)$, which means that $\mathsf{hbL}(X) \subseteq \mathsf{hbL}(Y)$. This means that any code-side hb edge in X can be replicated in Y to recreate the violation.
- For COHERENCE and COHERNA, we note that rf and mo are partitioned between X and Y. Therefore we can apply the same argument as for the previous axioms to show the hb edges for a violation must exist in either X or Y.
- Similarly, for ATOM we note that at is partitioned between X and Y so any violation must exist in either X or Y.

Finally, we consider RFHBNA. As $\mathsf{hist}Y \sqsubseteq_{\mathsf{h}} \mathsf{hist}X$, composing the two may weaken hb and generate violations on the context side. To solve this, we convert the RFHBNA violation to a safety violation. Take a $Z' \in Z^{\downarrow}$ such that there is a single $(\mathsf{hb} \cup \mathsf{rf})$-final RFHBNA violation. We redirect the origin of this read to its immediate hb-predecessor, or the initialisation value if this does not exist. This gives an execution Z'' which satisfies RFHBNA, but violated DRF. All the other validity axioms are preserved under prefixing, so by the completion lemma, $Z'' \in [\![C(B)]\!]_v^{\downarrow}$. We use Z'' as our constructed execution.

We now need to show that $\neg\mathsf{safe}(X)$ or $\neg\mathsf{safe}(Y)$ implies $\neg\mathsf{safe}(Z)$. If we had to fix an RFHBNA violation, the new execution Z'' is unsafe by construction. Otherwise, composition can only weaken hb, meaning any violation is trivially replicated.

Conversely, we need to show that if $\mathsf{safe}(X)$, $\mathsf{safe}(Y)$, and $\mathsf{safe}(Z)$, and $X \in [\![B, \mathcal{A}, \mathsf{hbC}(Y), \mathsf{atC}(Y)]\!]_v$ and $Y \in [\![C, \mathcal{A}, R', S']\!]_v$, then $Z \in [\![C(B)]\!]_v$ and $\mathsf{contxE}(Y) \preccurlyeq_{\mathsf{ex}} \mathsf{contxE}(Z)$. As Z is safe, we know we did not have to fix a RFHBNA violation. For the rest of the proof, the same arguments as above give us a valid execution $Z \in [\![C(B)]\!]_v$.

It remains to show that $\mathsf{contxE}(Y) \preccurlyeq_{\mathsf{ex}} \mathsf{contxE}(Z)$. Inclusion of context actions follows from the construction of Z. Inclusion on hb follows from the fact that $\mathsf{hist}(Y) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X)$. Thus the composition can only weaken hb over context actions. □

Theorem 11 (Adequacy). *$B_1 \sqsubseteq_{\mathsf{q}}^{\mathsf{NA}} B_2 \implies B_1 \preccurlyeq_{\mathsf{bl}} B_2$ for blocks that include only matched* LL/SC *pairs.*

Proof. Our objective from the definition of $\preccurlyeq_{\mathsf{bl}}$ is the following property:

$$\begin{aligned} \forall C, V.\ \ & \neg\mathsf{safe}(C(B_2)) \vee \\ & (\mathsf{safe}(C(B_1)) \wedge \forall X \in [\![C(B_1)]\!]_v.\, \exists Y \in [\![C(B_2)]\!]_v.\, X|_V \preccurlyeq_{\mathsf{ex}} Y|_V) \end{aligned}$$

Begin the proof by picking an arbitrary C, V. The proof then proceeds by the normal steps: decomposition, abstraction, then composition.

- Case-split on whether $C(B_2)$ is safe or unsafe. If unsafe, we are done immediately. Therefore we can assume $\mathsf{safe}(C(B_2))$.
- Pick an arbitrary execution $X \in [\![C(B_1)]\!]_v$.
- Apply the decomposition lemma to show that that $\mathsf{contxE}(X) \in [\![C, \mathsf{hbL}(X), \mathsf{atL}(X)]\!]_v$ and $\mathsf{codeE}(X) \in [\![B_1, \mathsf{hbC}(X), \mathsf{atC}(X)]\!]_v$.
- Expand the definition of $\sqsubseteq_{\mathsf{q}}^{\mathsf{NA}}$, and pick $R = \mathsf{hbC}(X)$ and $S = \mathsf{atC}(X)$. This gives us executions $Y \in [\![B_2, \mathcal{A}, \mathsf{hbC}(X), \mathsf{atC}(X)]\!]_v^{\downarrow}$ and $X' \in \mathsf{codeE}(X)^{\downarrow}$ such that:

$$\begin{aligned} & \mathsf{hist}(X') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Y) \wedge \\ & \quad \mathsf{safe}(Y) \implies (\mathsf{safe}(X') \wedge (X' = \mathsf{codeE}(X)) \wedge Y \in [\![B_2, \mathcal{A}, \mathsf{hbC}(X)]\!]_v) \end{aligned}$$

- Case-split on whether $\mathsf{safe}(Y) \wedge \mathsf{safe}(\mathsf{contxE}(X))$ holds. If not, then apply the composition lemma to build an execution $Z \in [\![C(B_2)]\!]_v^{\downarrow}$ such that $\neg\mathsf{safe}(Z)$. By lemma 9, there must exist a $Z' \in [\![C(B_2)]\!]_v$ such that $\neg\mathsf{safe}(Z')$, which contradicts our assumption that $C(B_2)$ is safe.

  Conversely, suppose $\mathsf{safe}(Y) \wedge \mathsf{safe}(\mathsf{contxE}(X))$ holds. In this case, we apply the context lemma to build a $Z \in [\![C(B_2)]\!]_v$ such that $\mathsf{contxE}(X) \preccurlyeq_{\mathsf{ex}} \mathsf{contxE}(Z)$. All actions on observable variables in V must be be in the context, which means that $X|_V \preccurlyeq_{\mathsf{ex}} Z|_V$ must also hold.

  It remains to prove that $\mathsf{safe}(X)$ holds. First we observe that $\mathsf{safe}(\mathsf{codeE}(X))$ holds by the abstraction theorem. As $\mathsf{safe}(\mathsf{contxE}(X))$ also holds, the result follows immediately.

□

C NON-ATOMICS AND DENIES

We now define $\sqsubseteq_{\mathsf{c}}^{\mathsf{NA}}$, a refinement between denotations which includes both cutting and non-atomics.

To do this we first need extra deny shapes. In the following, the variables obey the following constraint:

$$u, a, c \in \mathsf{ret} \cup \mathsf{interf}(X) \qquad\qquad v, b, d \in \mathsf{call} \cup \mathsf{interf}(X)$$

All the actions a, b, c, d, u, v are pairwise distinct. Note that some of the hb-edges are transitively closed, meaning that syntactically distinct actions might be the same – e.g. w_1 and u in HBvsMO-d.

$$\mathsf{CoNA\text{-}d}(u,v): \quad \exists w_1, w_2, r \in \mathsf{NA}. \quad \begin{tikzcd} w_1 \arrow[r, "\mathsf{hb}"] \arrow[dddr, "\mathsf{rf}"'] & w_2 \arrow[d, "\mathsf{hb}^*"] \\ & u \arrow[d, dotted, "D"] \\ & v \arrow[d, "\mathsf{hb}^*"] \\ & r \end{tikzcd} \quad \vee \quad \begin{tikzcd} w_1 \arrow[r, "\mathsf{hb}^*"] \arrow[drrr, "\mathsf{rf}"'] & u \arrow[r, dotted, "D"] & v \arrow[r, "\mathsf{hb}^*"] & w_2 \arrow[d, "\mathsf{hb}"] \\ & & & r \end{tikzcd}$$

$$\mathsf{CoNA\text{-}d2}(a,b,c,d): \quad \exists w_1, w_2, r \in \mathsf{NA}. \quad \begin{tikzcd} w_1 \arrow[r, "\mathsf{hb}^*"] \arrow[ddddrrr, "\mathsf{rf}"'] & a \arrow[r, dotted, "D_1"] & b \arrow[r, "\mathsf{hb}^*"] & w_2 \arrow[d, "\mathsf{hb}^*"] \\ & & & c \arrow[d, dotted, "D_2"] \\ & & & d \arrow[d, "\mathsf{hb}^*"] \\ & & & r \end{tikzcd}$$

As before, we need a few notions to define the deny theorem.

- $\mathsf{denyL}(X)$ contains all the binary denies:

$$\mathsf{denyL}(X) \;\triangleq\; \mathsf{HBvsMO\text{-}d} \cup \mathsf{CoWR\text{-}d} \cup \mathsf{Init\text{-}d} \cup \mathsf{CoNA\text{-}d}$$

- $\mathsf{denyNA}(X)$ contains the quaternary denies: $\mathsf{denyNA}(X) \;\triangleq\; \mathsf{CoNA\text{-}d2}$
- $\mathsf{guarNA}(X)$ is the projection of $(\mathsf{rf}_{\mathsf{NA}} \cup \mathsf{hb})^+$ to pairs in

$$(\mathsf{interf}(X) \times \mathsf{interf}(X)) \cup (\mathsf{interf}(X) \times \{\mathsf{ret}\}) \cup (\{\mathsf{call}\} \times \mathsf{interf}(X))$$

- Let $\mathcal{I}$ be the set of actions $\mathsf{interf}(X) \cup \{\mathsf{call}, \mathsf{ret}\}$. The *augmented history* of X, written $\mathsf{hist}_{\mathsf{E}}(X)$, is defined as

$$\mathsf{hist}_{\mathsf{E}}(X) \;\triangleq\; (A(X)|_{\mathcal{I}}, \mathsf{hbL}(X), \mathsf{denyL}(X), \mathsf{guarNA}(X), \mathsf{denyNA}(X))$$

- Two augmented histories, $H = (\mathcal{A}, G, D, M, N)$, $H' = (\mathcal{A}', G', D', M', N')$ are related $H \sqsubseteq_{\mathsf{h}} H'$ iff

$$\mathcal{A} = \mathcal{A}' \wedge G' \subseteq G \wedge D' \subseteq D \wedge M' \subseteq M \wedge N' \subseteq N$$

- $\mathsf{FinalNA}(X, a)$ holds if the action a is (1) an NA action, and (2) is the hb-final action in the code block in X.
- $\mathsf{hbA}(X, a)$, for an execution X and action a is the projection of $\mathsf{hb}(X)$ to pairs in $(\{a\} \times \mathsf{interf}(X)) \cup (\mathsf{interf}(X) \times \{a\})$

- The comparison $a \leq_{\mathsf{na}}^{X,Y} b$ ensures that b participates in a race if a does. Formally, the comparison holds if: (1) a and b are actions on the same location; (2) b is a write if a is a write; and (3) $\mathsf{hbA}(Y, b) \subseteq \mathsf{hbA}(X, a)$. The final condition is needed to ensure that history edges cannot spuriously prevent a race in Y.
- The set of *almost-valid* executions $[\![B, \mathcal{A}, R, S]\!]_{vr}$ is defined identically to the standard semantics, except that it permits RFHBNA not to hold. We write $[\![B, \mathcal{A}, S]\!]_{vr}$ stands for $[\![B, \mathcal{A}, \emptyset, S]\!]_{vr}$

We then define *deny abstraction* as follows:

$$\begin{aligned} B_1 \sqsubseteq_{\mathsf{d}}^{\mathsf{NA}} B_2 \quad \stackrel{\Delta}{=} \quad & \forall S.\, \forall X \in [\![B_1, \mathcal{A}, S]\!]_{vr}^{\downarrow}.\, \exists Y \in [\![B_2, \mathcal{A}, S]\!]_{vr}^{\downarrow}.\, \mathsf{hist}_{\mathsf{E}}(X) \sqsubseteq_{\mathsf{E}} \mathsf{hist}_{\mathsf{E}}(Y) \wedge \\ & (\forall a.\, \mathsf{FinalNA}(X, a) \implies \exists b \in A(Y).\, a \leq_{\mathsf{na}}^{X,Y} b) \wedge \\ & (X \in [\![B_1, \mathcal{A}, S]\!]_{vr} \implies Y \in [\![B_2, \mathcal{A}, S]\!]_{vr}) \end{aligned}$$

In addition to the cutting predicated defined in the body of the paper, we need the following to cover NA cuts.

$$\begin{aligned} \mathsf{NAcutR}(X) \quad \stackrel{\Delta}{=} \quad & \forall r_1, r_2 \in (\mathsf{interf}(X) \cap \mathsf{Read} \cap \mathsf{NA}). \\ & \quad (\mathsf{val}(r_1) = \mathsf{val}(r_2) = \mathsf{init} \vee \exists w.\, w \xrightarrow{\mathsf{rf}} r_1 \wedge w \xrightarrow{\mathsf{rf}} r_2) \\ & \quad\quad \implies (r_1 = r_2) \\ \mathsf{NAcutW}(X) \quad \stackrel{\Delta}{=} \quad & \forall w_1, w_2 \in (\mathsf{interf}(X) \cap \mathsf{Write} \cap \mathsf{NA}). \\ & \quad (\mathsf{loc}(w_1) = \mathsf{loc}(w_2)) \implies \\ & \quad\quad (w_1 = w_2) \vee (\exists r \in \mathsf{code}(X).\, w_1 \xrightarrow{\mathsf{rf}} r \vee w_2 \xrightarrow{\mathsf{rf}} r) \end{aligned}$$

The context cutting predicate is defined as the conjunction of these predicates:

$$\mathsf{cut}^{\mathsf{NA}}(X) \stackrel{\Delta}{=} \mathsf{cutR}(X) \wedge \mathsf{cutW}(X) \wedge \mathsf{NAcutR}(X) \wedge \mathsf{NAcutW}(X)$$

We then define *cut abstraction* as follows:

$$\begin{aligned} B_1 \sqsubseteq_{\mathsf{c}}^{\mathsf{NA}} B_2 \quad \stackrel{\Delta}{=} \quad & \forall X \in [\![B_1]\!]_{vr}^{\downarrow}.\, \mathsf{cut}^{\mathsf{NA}}(X) \implies \\ & \quad \exists Y \in [\![B_2]\!]_{vr}^{\downarrow}.\, \mathsf{hist}_{\mathsf{E}}(X) \sqsubseteq_{\mathsf{E}} \mathsf{hist}_{\mathsf{E}}(Y) \wedge \\ & \quad (\forall a.\, \mathsf{FinalNA}(X, a) \implies \exists b \in A(Y).\, a \leq_{\mathsf{na}}^{X,Y} b) \wedge \\ & \quad (X \in [\![B_1]\!]_{vr} \implies Y \in [\![B_2]\!]_{vr}) \end{aligned}$$

Theorem 12. $B_1 \sqsubseteq_{\mathsf{d}}^{\mathsf{NA}} B_2 \implies B_1 \sqsubseteq_{\mathsf{q}}^{\mathsf{NA}} B_2$

Proof. Pick a context-side $\mathcal{A}$, R and an execution $X \in [\![B_1, \mathcal{A}, R, S]\!]_v$. Case-split on $\mathsf{safe}(X)$ – suppose first that it does not hold.

- Pick a prefix $X' \in X$ and action $a \in A(X')$ such that (1) X' contains precisely one safety violation, which includes a; and (2) $\mathsf{FinalNA}(X', a)$ holds.
- Generate a new execution X'' by building hb as $(\mathsf{sw} \cup \mathsf{sb})^+$ (i.e. kick out R). As all axioms but RFHBNA are preserved under reduction of hb, $X' \in [\![B_1, \mathcal{A}, S]\!]_{vr}^{\downarrow}$.
- Apply the assumption to give an execution $Y' \in [\![B_2, \mathcal{A}, S]\!]_{vr}^{\downarrow}$, such that $\mathsf{hist}_{\mathsf{E}}(X'') \sqsubseteq_{\mathsf{E}} \mathsf{hist}_{\mathsf{E}}(Y')$. By the theorem, there must exist an action b to the same location such that $a \leq_{\mathsf{na}} b$.
- Build Y from Y' by defining $\mathsf{hb}(Y)$ as $\mathsf{sb}(Y') \cup \mathsf{rf}_{\mathsf{NA}}(Y') \cup R$, and keeping other relations the same. We now need to establish that (1) $\mathsf{hist}(X') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Y)$; (2) $\neg\mathsf{safe}(Y)$; and (3) $\mathsf{valid}(Y)$.
- $\mathsf{hist}(X') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Y)$ holds from the fact that $\mathsf{hist}_{\mathsf{NA}}(X'') \sqsubseteq_{\mathsf{E}} \mathsf{hist}_{\mathsf{E}}(Y')$, and both X' and Y are derived by adding the same relation R.
- To show $\neg\mathsf{safe}(Y)$ we observe that action a in X' participates in a race. As actions in a code-block are sb-sequenced, the other action c forming the race must be in $\mathsf{interf}(X')$. If (b, c) does not form a race in Y, then (b, c) or (c, b) must be in $\mathsf{hb}(Y)$. Any such path must be in $R \cup \mathsf{hbL}(Y) \cup \mathsf{hbA}(Y', b)$. The

corresponding path must exist in $R \cup \mathsf{hbL}(X) \cup \mathsf{hbA}(X'', b)$, which rules out the race in X and contradicts the assumption.

- Finally, we need to prove that $\mathsf{valid}(Y)$. HBDEF' holds by construction. RFWF, RFVAL, MOWF, ATOM are invariant under adding hb-edges, and so follow immediately from $\mathsf{valid}(Y')$. This leaves ACYCLICITY, COHERENCE, HBVSMO, COHERNA, and RFHBNA.

  All but RFHBNA are covered by a deny (RFHBNA requires special treatment). A new violation of an axiom caused by edges from R would induce a corresponding deny shape in $\mathsf{hist}_\mathsf{E}(Y')$. As $\mathsf{hist}_\mathsf{E}(X') \sqsubseteq_\mathsf{E} \mathsf{hist}_\mathsf{E}(Y)$ this deny shape must also be in X'. However, this means that the corresponding violation can be replicated in X', which contradicts the assumption that $\mathsf{valid}(X')$ holds.
- Thus, we have an almost-valid execution $Y \in [\![B_2, \mathcal{A}, R]\!]^{\downarrow}_{vr}$ such that $\mathsf{hist}(X') \sqsubseteq_\mathsf{h} \mathsf{hist}(Y)$; (2) $\neg\mathsf{safe}(Y)$.

  To complete the proof, we need to fix violations of RFHBNA. We use the same approach as in the proof of Theorem 10: (1) build a shorter prefix in $Y^{\downarrow}$ which contains precisely one violation of RFHBNA; (2) redirect the read to a valid origin, using the receptiveness of the thread-local semantics. This redirection does alter the history, because non-atomic reads do not appear in the quantified history. This gives an execution Y'' such that (1) $Y'' \in [\![B_2, \mathcal{A}, R, S]\!]^{\downarrow}_{vr}$; (2) $\neg\mathsf{safe}(Y'')$; and (3) $\mathsf{hist}(Y'') \in \mathsf{hist}(Y^{\downarrow})$.

  We finally need to show that there exists an $X''' \in X^{\downarrow}$ such that $\mathsf{hist}(X''') \sqsubseteq_\mathsf{h} \mathsf{hist}(Y'')$. This necessarily exists by application of the history prefixing lemma. Note that X''' may not necessarily be unsafe, but Y'' is guaranteed to be unsafe by construction.

Now suppose that $\mathsf{safe}(X)$ holds. We use essentially the same proof structure as above: the constructed Y may be safe or unsafe, depending whether we need to fix violations of RFHBNA. □

D PROOF OF THEOREMS 3 AND 5

We now prove that $\sqsubseteq_\mathsf{c}^{\mathsf{NA}}$ is adequate. Note that because $\sqsubseteq_\mathsf{c}^{\mathsf{NA}} \implies \sqsubseteq_\mathsf{c}$, we implicitly prove $\sqsubseteq_\mathsf{c}$ adequate. We define several versions of the abstractions with different levels of context cutting:

$$
\begin{array}{rcl}
B_1 \sqsubseteq_\mathsf{c}^{i} B_2 & \triangleq & \forall X \in [\![B_1, \mathcal{A}]\!]^{\downarrow}_{vr}.\, \mathsf{cut}^i(X) \implies \\
& & \quad \exists Y \in [\![B_2, \mathcal{A}]\!]^{\downarrow}_{vr}.\, \mathsf{hist}_\mathsf{E}(X) \sqsubseteq_\mathsf{E} \mathsf{hist}_\mathsf{E}(Y) \wedge \\
& & \quad (\forall a.\, \mathsf{FinalNA}(X, a) \implies \exists b \in A(Y).\, a \leq_{\mathsf{na}}^{X,Y} b) \wedge \\
& & \quad (X \in [\![B_1, \mathcal{A}]\!]_{vr} \implies Y \in [\![B_2, \mathcal{A}]\!]_{vr})
\end{array}
$$

We define several versions of the cutting predicate, incrementally cutting more of the context.

$$
\begin{array}{rcl}
\mathsf{cut}^1(X) & \triangleq & \mathsf{cutR}(X) \\
\mathsf{cut}^2(X) & \triangleq & \mathsf{cutR}(X) \wedge \mathsf{cutW}(X) \\
\mathsf{cut}^3(X) & \triangleq & \mathsf{cutR}(X) \wedge \mathsf{cutW}(X) \wedge \mathsf{NAcutR}(X)
\end{array}
$$

LEMMA 13 (ATOMIC READ CUTTING). $B_1 \sqsubseteq_\mathsf{c}^{1} B_2 \implies B_1 \sqsubseteq_\mathsf{d}^{\mathsf{NA}} B_2$

PROOF.
- Pick an execution $X \in [\![B_1, \mathcal{A}, S]\!]^{\downarrow}_{d}$. We now want to build a corresponding execution such that cutR holds.
- Identify a subset $\mathcal{A}' \subseteq A(X)$ such that $\mathsf{cutR}(X|_{\mathcal{A}'})$ holds, and no larger subset exists. We call this maximal projected execution X'. We use $\mathcal{A}_R$ to refer to the removed actions $\mathcal{A} \setminus \mathcal{A}'$.
- It's straightforward to see that $\mathcal{A}_R \subseteq \mathsf{Read} \cap \mathsf{interf}(X)$. Context actions aren't required by the thread-local semantics, and removing context reads preserves validity, so $X' \in [\![B_1, \mathcal{A}, S]\!]^{\downarrow}_{d}$.

  It's also straightforward from the definition of cutR to see that any read r in $\mathcal{A}_R$ is removed for one of two reasons:
  - *context-read.* The associated write for r is in the context.

- *duplicate-read*. The associated write is read by another context read r' which is not removed. We call this r' the *representative* for r.

- We have an execution $X' \in [\![L_1, \mathcal{A}, S]\!]_d^{\downarrow}$ such that $\mathsf{cutR}(X')$ holds. Now apply the assumption to produce an execution $\exists Y' \in [\![L_2, \mathcal{A}, S]\!]_d^{\downarrow}$ such that $\mathsf{hist_E}(X') \sqsubseteq_\mathsf{E} \mathsf{hist_E}(Y')$.

  Build a new execution Y by re-injecting the actions from $\mathcal{A}_R$. As all of these actions are context reads, the only relation that must change is rf.
  - If the action is a context-read, direct rf to the context write it pointed to in X. This must still exist by history inclusion.
  - If the action is a duplicate-read, direct rf to the write read by its representative. The origin for the representative write must exist by validity of Y'.

  It now remains to show that that $Y \in [\![L_2, \mathcal{A}, S]\!]_d^{\downarrow}$ and $\mathsf{hist_E}(X) \sqsubseteq_\mathsf{E} \mathsf{hist_E}(Y)$.
- To show that $Y \in [\![B_2, \mathcal{A}, S]\!]_d^{\downarrow}$, we only need to show that Y is valid. Adding new atomic context reads to a valid execution is guaranteed to preserve validity, as long as they are equipped with valid origin writes in rf.
- To show that $\mathsf{hist_E}(X) \sqsubseteq_\mathsf{E} \mathsf{hist_E}(Y)$, we have two obligations: $\mathsf{hbL}(Y) \subseteq \mathsf{hbL}(X)$, and $\mathsf{denyL}(Y) \subseteq \mathsf{denyL}(X)$. The former is a trivial consequence of the way we construct Y.

  For the latter, we reason by contradiction for each of the deny shapes:
  - HBvsMO-d and Acyc-d: As context reads are terminal in hb, the only case we need to consider is the one where $u \in \mathcal{A}_R$ and the remainder of the shape is not removed. Otherwise the deny is entirely replicated in Y', and thus in X. If u is a duplicate-read, the deny is replicated in Y using its representative. If u is a context-read, a deny edge exists $w_1 \xrightarrow{\mathsf{d}} v$. In either case, it is easy to see that the deny $u \xrightarrow{\mathsf{d}} v$ must be replicable in X, contradicting the assumption.
  - Cohere-d and Init-d: Similarly, the cases we need to consider are (1) $u \in \mathcal{A}_R$, (2) $v = r$ and $r \in \mathcal{A}_R$, and (3) both. In the first case, the same argument applies as with HBvsMO. In the second, we can replace r with its representative. In both cases, it's straightforward to replicate the deny $u \xrightarrow{\mathsf{d}} v$ is replicated in X. The third case just combines the arguments from the other two.
  - CoNA-d and CoNA-d2: Ruled out as actions in $\mathcal{A}_R$ must be hb-terminal. This precludes any such action participating in one of these non-atomic shapes.
- Finally, we need to show that any final NA action in X is replicated in Y, and that Y is complete if X is complete. Both properties are inherited trivially from Y'.

□

Lemma 14 (Atomic write cutting). $B_1 \sqsubseteq_\mathsf{c}^2 B_2 \implies B_1 \sqsubseteq_\mathsf{c}^1 B_2$

Proof.
- Pick an $X \in [\![B_1, \mathcal{A}, S]\!]_d^{\downarrow}$ such that $\mathsf{cutR}(X)$ holds.
- Now we build an X' such that $X' \in [\![B_1, \mathcal{A}, S]\!]_d$ and $\mathsf{cutR}(X) \wedge \mathsf{cutW}'(X)$ holds. First identify the set of non-visible write actions for each location z:

$$\mathcal{A}^z = \{a \in \mathcal{A} \mid \mathsf{loc}(a) = z \wedge a \in (\mathsf{Write} \cap \mathsf{Atomic}) \wedge \neg\mathsf{visible}(a)\}$$

  Partition this set into maximal disjoint nonempty subsets $\mathcal{B}_1^z, \mathcal{B}_2^z \ldots$ such that:

$$\mathcal{B}_i^z \subseteq \mathcal{A}^z \wedge (\forall a_1, a_2 \in \mathcal{B}_n^z.\, \neg\exists w \notin \mathcal{B}_n^z.\, a_1 \xrightarrow{\mathsf{mo}} w \xrightarrow{\mathsf{mo}} a_2)$$

  In other words, each set $\mathcal{B}$ is a maximal set of non-visible writes so that there is no intervening write in mo. Thus, either a set $\mathcal{B}$ is mo-minimal / maximal, or it has a visible action which is its immediate mo-predecessor / successor. We call these actions $w_p^{\mathcal{B}}$ and $w_s^{\mathcal{B}}$ respectively. (The cases where $\mathcal{B}$ is minimal / maximal are ignored as they are simpler versions of the case where $w_p^{\mathcal{B}}$ and $w_s^{\mathcal{B}}$ exist.

Note that due to Coherence, if there is a LL-SC in $\mathcal{B}$, it must either read from a write in $\mathcal{B}$, or from $w_p^{\mathcal{B}}$. Similarly, if $w_s^{\mathcal{B}}$ is a LL-SC, it must read from a write in $\mathcal{B}$.

To build X', replace each $\mathcal{B}$ with a single LL-SC pair $w_n^{\mathcal{B}}$ (as above, call this a *representative*). Take as the value that is read the value of $w_p^{\mathcal{B}}$, and take as the written value the mo-final value written in $\mathcal{B}$. We modify the rest of the execution as follows:

- As each set $\mathcal{B}$ is mo-contiguous in X, we don't need to modify mo other than to insert the new LL-SC pair.
- As the execution satisfies cutR, we have already kicked out all the context reads. We direct rf so that $w_p^{\mathcal{B}} \xrightarrow{\mathsf{rf}} w_n^{\mathcal{B}}$. If $w_s^{\mathcal{B}}$ is a LL-SC, we direct rf so that $w_n^{\mathcal{B}} \xrightarrow{\mathsf{rf}} w_s^{\mathcal{B}}$.
- Introducing $w_n^{\mathcal{B}}$ may generate new hb edges, so we regenerate hb according to HBdef'.

- We now need to show that X' is valid. This is simple for most of the axioms because the writes that are removed can only be read by their immediate mo-successor. However, if $w_s^{\mathcal{B}}$ is a LL-SC, then we might generate an hb-edge $w_p^{\mathcal{B}} \xrightarrow{\mathsf{hb}} w_s^{\mathcal{B}}$ which did not previously exist. We therefore need to show that Acyclicity, HBvsMO, Coherence, CoherNA still hold in X'.
  - HBvsMO, Acyclicity: The two writes w_1, w_2 responsible must be on a different location from $w_p^{\mathcal{B}}$ and $w_s^{\mathcal{B}}$: otherwise the violation would be an HBvsMO violation in X. Any hb-path between two actions on different locations must pass through the code. If the $w_p^{\mathcal{B}}$ and $w_s^{\mathcal{B}}$ are not themselves in the code, we can replicate the violation immediately using the hb-adjacent internal actions a_p/a_s.
  - Coherence, CoherNA: Again, the responsible writes / reads must be to a different location from $w_p^{\mathcal{B}}$ and $w_s^{\mathcal{B}}$. Otherwise we can generate a violation using the LL-SC $w_s^{\mathcal{B}}$, and the fact that mo follows hb. Apply the same reasoning as the previous point to replicate the violation in X.

  We also need to show that $\mathsf{cutR}(X') \wedge \mathsf{cutW}(X')$ holds. It's obvious that $\mathsf{cutR}(X')$ still holds – we have introduced no extra reads. $\mathsf{cutW}(X')$ holds because each new write $w_n^{\mathcal{B}}$ is separated in mo by a visible action.
- Apply the assumption to give an execution $Y' \in [\![L_2, \mathcal{A}, S]\!]$ such that $X' \sqsubseteq_{\mathsf{E}} Y'$.

  Now build the execution Y. To do this, replace each representative LL-SC $w_n^{\mathcal{B}}$ in Y' with the corresponding actions in $\mathcal{B}$. In other words, for any pair of actions in a single set $\mathcal{B}$, take mo the same as in $\mathsf{mo}(X)$. For an action in $\mathcal{B}$ and some other action, relate it in mo as in $\mathsf{mo}(Y')$ for the set representative.

  We need to show that (1) Y is valid, (2) $\mathsf{hist}_{\mathsf{E}}(X) \sqsubseteq_{\mathsf{E}} \mathsf{hist}_{\mathsf{E}}(Y)$.
- *Validity.* Modifying Y' to Y alters rf, mo, and hb.

  RFwf, RFval, MOwf, Atom are obvious by construction.

  Acyclicity holds because hb edges are only removed between existing writes, and introduced between actions represented in $\mathcal{B}$, which are by definition unrelated to context actions aside from at $w_p^{\mathcal{B}}$ and $w_s^{\mathcal{B}}$. Therefore any cycle would exist inside $\mathcal{B}$, and thus in X.

  HBvsMO holds because actions in $\mathcal{B}$ are introduced at a single point in mo represented by $w_n^{\mathcal{B}}$. Any hb-edges inside $\mathcal{B}$ must be consistent with mo, or a similar violation could be replicated in X.

  Coherence, CoherNA holds because any violation for non-$\mathcal{B}$ reads/write could be replicated in Y' using the representative LL-SC $w_n^{\mathcal{B}}$. A violation inside $\mathcal{B}$ could immediately be replicated in X.

  RFHBNA holds because any hb-path in Y' that is broken in Y must pass through the code. Therefore, the path must be replicable through sb, which contradicts the violation.

- $\mathsf{hbL}(Y) \subseteq \mathsf{hbL}(X)$. Actions in $\mathcal{B}$ in X are only related to each other and $w_p^{\mathcal{B}}$ / $w_s^{\mathcal{B}}$ in hb. For paths in hb outside some $\mathcal{B}$, it must be that $\mathsf{hbL}(X) = \mathsf{hbL}(X') \subseteq \mathsf{hbL}(Y') = \mathsf{hbL}(Y)$. As paths inside $\mathcal{B}$ are identical between X and Y, any hb-path can be replicated.
- $\mathsf{guarNA}(Y) \subseteq \mathsf{guarNA}(X)$. Trivial by the previous argument, and the fact $\mathcal{B}$-sets only cover atomic actions.
- $\mathsf{denyL}(Y) \subseteq \mathsf{denyL}(X)$. Prove by contradiction: assume a deny shape in Y that is not in X.
  - HBvsMO-d / Acyc-d: Suppose a deny shape involving writes w_1/w_2.
    * w_1/w_2 not in any $\mathcal{B}$. As actions in $\mathcal{B}$ are not read/written in the code, any hb path which includes actions in $\mathcal{B}$ and which passes through the code, must enter and exit $\mathcal{B}$ through other context actions, a, b. There is a deny $a \xrightarrow{\mathsf{d}} b$ in Y' by construction, and thus in X. hb-paths inside $\mathcal{B}$ are identical in X and Y. Combining this gives us a deny in X.
    * w_1/w_2 entirely inside $\mathcal{B}$: reproducible trivially as mo/hb are identical between Y and X.
    * w_1 outside $\mathcal{B}$, w_2 inside. In this case, there must be a deny in Y' and X' with the representative: $w_p^{\mathcal{B}} \xrightarrow{\mathsf{d}} b$ (using the same argument as above). Substituting $\mathcal{B}$ for the representative in X builds the violation.
    * w_2 outside $\mathcal{B}$, w_1 inside. Symmetrical to previous case.
  - Cohere-d / Init-d: the deny shape involves writes $w_1/w_2/r$.
    * w_1, w_2, r all in $\mathcal{B}$: replicated trivially.
    * w_1, w_2, r all outside $\mathcal{B}$: replicated trivially.
    * w_1, w_2 in $\mathcal{B}$, r outside: r can only be $w_s^{\mathcal{B}}$, shape ruled out by construction.
    * w_1 in $\mathcal{B}$, w_2, r outside: deny replicated in Y' / X' using representative. Rebuild the violation when reintroducing $\mathcal{B}$ in X.
    * All three outside $\mathcal{B}$: trivial.
    * w_2, r in $\mathcal{B}$, w_1 outside: w_1 can only be $w_p^{\mathcal{B}}$, shape ruled out by construction.
    * w_2 in $\mathcal{B}$, w_1, r outside: deny replicated in Y' using representative, rebuild in X when adding $\mathcal{B}$.
    * r in $\mathcal{B}$, w_1, w_2 outside: w_1 can only be $w_p^{\mathcal{B}}$, shape ruled out by construction.
  - CoNA-d2: similar argument to Cohere-d, exept that some cases are ruled out by the fact that elements in $\mathcal{B}$ are necessarily atomic.
- $\mathsf{denyNA}(X) \subseteq \mathsf{denyNA}(Y)$. Similar argument to CoNA-d2 above.
- The Final NA and completeness properties are satisfied for the same reason as in the previous proof.

□

Lemma 15 (NA read cutting). $B_1 \sqsubseteq^3_{\mathsf{c}} B_2 \implies B_1 \sqsubseteq^2_{\mathsf{c}} B_2$

Proof.

- Pick an $X \in [\![B_1, \mathcal{A}, S]\!]^{\downarrow}$ such that $\mathsf{cut}^2(X)$ holds. Build X' using the same approach as in atomic read cutting: X' is a maximal sub-execution such that $\mathsf{NAcutR}(X)$ holds.

  From the structure of NAcutR, the actions $\mathcal{A}_R$ removed from X must all be non-atomic reads. Just as before, removed reads have a *representative* that remains in X' and that reads from the same write. Unlike in the atomic case, reads from context writes also have representatives. This is necessary to detect new writes that might violated CoNA-d2 (which in turn is needed because NA writes are not ordered in mo).

  X' is valid because the axioms are invariant under read removal.
- We then apply the assumption to build an execution $Y' \in [\![B_2, \mathcal{A}, S]\!]^{\downarrow}_{vr}$. Finally we build Y by restoring the cut actions, with $\mathsf{rf}(Y')$ built in the same way as for the atomic cutting case.

  Almost-validity is preserved trivially because the inserted reads are not part of hb. Deny inclusion is ensured by the fact that the inserted reads are placed at the same position as their representatives: any violation would immediately be replicated by the representative. The FinalNA and completion property are unaffected from Y'.

□

Lemma 16 (NA write cutting). $B_1 \sqsubseteq_c B_2 \implies B_1 \sqsubseteq_c^3 B_2$

Proof.
- Pick an $X \in [\![B_1, \mathcal{A}, S]\!]^{\downarrow}$ such that $\mathsf{cut}^3(X)$ holds.
- As NAcutWdoesn't discriminate on the basis of mo, we can replace the set of all context writes to a location with a single representative write. We build X' as a maximal sub-execution such that NAcutWholds, and 'orphan' context reads are removed. As NAcutRholds, each NA write has at most one context read. Note that as the execution is maximal, if at least one write to a location had an associated read, then the representative will have an associated read.
- Validity for X' is trivial as the removed writes cannot participate in hb, or be read in the code. We then build Y' by applying the theorem to give an almost-valid execution of B_2. Finally, we build Y by re-inserting the removed reads and writes. The only relation that needs to be updated is rf, which associates removed reads with their origin write.

  Preservation of almost-validity follows from the fact that the inserted writes are disjoint from all other actions in the execution relations. Deny inclusion holds because any deny shape in Y that involves a removed write / read can be easily replicated using the representative.

□

Theorem 17 (Cut adequacy). $B_1 \sqsubseteq_c B_2 \implies B_1 \sqsubseteq_d^{\mathsf{NA}} B_2$

Proof. Prove this as a sequence of implications:

$$B_1 \sqsubseteq_c B_2 \implies B_1 \sqsubseteq_c^3 B_2 \implies B_1 \sqsubseteq_c^2 B_2 \implies B_1 \sqsubseteq_c^1 B_2 \implies B_1 \sqsubseteq_d^{\mathsf{NA}} B_2$$

Each implication step is proved in a lemma above. □

E PROOFS OF THEOREMS 2 AND 6 (FULL ABSTRACTION)

Context construction. The full context construction including LL/SC is included in Figure 11. The key difference from Figure 9 is that successful context LL/SC pairs in X are arranged on a single thread, allowing the store conditional to suceed in C_X.

Proof of Theorem 6. We now prove Theorem 6: full abstraction of $\sqsubseteq_q^{\mathsf{NA}}$ for programs and contexts that do not use read-modify-write accesses, $B_1 \preccurlyeq_{\mathsf{bl}}^{\mathsf{NA}} B_2 \implies B_1 \sqsubseteq_q^{\mathsf{NA}} B_2$. Note that this implies Theorem 2.

Proof.
- Start by choosing an arbitrary R, S and $X \in [\![B_1, \mathcal{A}, R, S]\!]_v^{\mathsf{NA}}$. It remains to show that:

$$\begin{aligned}&\exists Y \in [\![B_2, \mathcal{A}, R, S]\!]_v^{\mathsf{NA}}.\\&(\mathsf{safe}(Y) \implies \mathsf{safe}(X) \wedge \mathsf{hist}(X) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Y)) \;\wedge\\&(\neg\mathsf{safe}(Y) \implies \exists X_1' \in (X)^{\downarrow}. \exists X_2' \in (Y)^{\downarrow}. \neg\mathsf{safe}(X_2') \wedge \mathsf{hist}(X_1') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X_2'))\end{aligned} \tag{11}$$

$$
\begin{aligned}
C_X &= (;_{(m_1,m_2)\in \mathsf{at}}\, a(m_1); a(m_2)) \parallel (\|_{m \setminus |\mathsf{at}|}\, a(m)) \\
a(m) &= \mathtt{Racq}_m(\mathtt{Nrel}_m; \mathtt{check}(m)(\mathtt{Nacq}_m(\mathtt{Rrel}_m))) \\
\mathtt{Rrel}_m &= \mathtt{Rrel}_{\mathtt{ret}(m),v_1}; \ldots ; \mathtt{Rrel}_{\mathtt{ret}(m),v_n}, \\
&\quad \text{where } \{v_1, \ldots, v_n\} = \{v \mid (\mathtt{ret}(m), v) \in R\} \\
\mathtt{Rrel}_{u,v} &= \mathtt{store}(h_{u,v}, 1) \\
\mathtt{Racq}_m(N) &= \mathtt{Racq}_{u_1,\mathtt{call}(m)}(\ldots \mathtt{Racq}_{u_n,\mathtt{call}(m)}(N) \ldots), \\
&\quad \text{where } \{u_1, \ldots, u_n\} = \{u \mid (u, \mathtt{call}(m)) \in R\} \\
\mathtt{Racq}_{u,v}(N) &= \mathtt{if}\ (\mathtt{load}(h_{u,v}))\ N\ \mathtt{else}\ \mathtt{store}(e, 1) \\
\mathtt{Nrel}_m &= \mathtt{Nrel}_{\mathtt{call}(m),v_1}; \ldots ; \mathtt{Nrel}_{\mathtt{call}(m),v_n}, \\
&\quad \text{where } \{v_1, \ldots, v_n\} = \{v \mid (\mathtt{call}(m), v) \in H\} \\
\mathtt{Nrel}_{u,v} &= \mathtt{store}(g_{u,v}) \\
\mathtt{Nacq}_m(N) &= \mathtt{Nacq}_{u_1,\mathtt{ret}(m)}(\ldots \mathtt{Nacq}_{u_n,\mathtt{ret}(m)}(N) \ldots), \\
&\quad \text{where } \{u_1, \ldots, u_n\} = \{u \mid (u, \mathtt{ret}(m)) \in H\} \\
\mathtt{Nacq}_{u,v}(N) &= \mathtt{if}\ (\neg\mathtt{load}(g_{u,v}))\ N\ \mathtt{else}\ \mathtt{store}(e, 1)
\end{aligned}
$$

Fig. 11. Context construction: m ranges over context actions $\mathcal{A}$ and a code-block B'.

- Apply the construction lemma (Lemma 18, below) to B_1, R, S and X to find a context C_X, and an execution Z:

$$
\begin{aligned}
& Z \in [\![C_X(B_1)]\!]_v^{\mathsf{NA}} \wedge \\
& \mathsf{code}(Z) = X \wedge \\
& \mathsf{hbC}(Z) = R \wedge \mathsf{atC}(Z) = S \wedge \\
& \forall B'. \forall Z' \in [\![C_X(B')]\!]_v^{\mathsf{NA}}. \\
& \quad ((A(\mathsf{contx}(Z)) = A(\mathsf{contx}(Z'))) \implies (\mathsf{hist}(Z) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Z')) \wedge \mathsf{at}(\mathsf{contx}(Z)) = \mathsf{at}(\mathsf{contx}(Z'))) \wedge \\
& \quad (\neg\mathsf{safe}(Z') \implies \exists X' \in X^{\downarrow}. \exists W \in ([\![B', \mathcal{A}, R, S]\!]_v^{\mathsf{NA}})^{\downarrow}. \neg\mathsf{safe}(W) \wedge \mathsf{hist}(X') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(W))
\end{aligned}
\tag{12}
$$

- Specialise observation with the context C_X and the set of all variables used in C_X, V_{C_X}, to get:

$$
\begin{aligned}
(\quad & \neg\mathsf{safe}(C_X(B_2)) \vee \\
& (\mathsf{safe}(C_X(B_1)) \wedge \\
& \quad \forall X \in [\![C_X(B_1)]\!]_v^{\mathsf{NA}}. \exists Y \in [\![C_X(B_2)]\!]_v^{\mathsf{NA}}. \\
& \qquad \mathcal{A}(X|_{V_{C_X}}) = \mathcal{A}(Y|_{V_{C_X}})) \wedge \\
& \qquad \mathsf{hb}(X|_{V_{C_X}}) \subseteq \mathsf{hb}(Y|_{V_{C_X}})))
\end{aligned}
\tag{13}
$$

  and then case split on $\mathsf{safe}(C_X(B_2))$.
- **Case 1:** $\mathsf{safe}(C_X(B_2))$.
  - By 13, there is an execution, Z' of $C_X(B_2)$ with:

$$
\mathsf{hb}(Z|_{V_{C_X}}) = \mathsf{hb}(Z'|_{V_{C_X}}) \wedge \mathcal{A}(Z|_{V_{C_X}}) = \mathcal{A}(Z'|_{V_{C_X}})
$$

  - By construction of C_X, the variables V_{C_X} cover all context variables, so we have:

$$\mathsf{hb}(\mathsf{contx}(Z)) = \mathsf{hb}(\mathsf{contx}(Z')) \ \wedge A(\mathsf{contx}(Z)) = A(\mathsf{contx}(Z'))$$

- Appealing to 12, we have:

$$\mathsf{hist}(Z) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Z')$$

- Now apply the decomposition lemma to Z' to get the execution Y:

$$Y \in [\![B_2, \mathcal{A}, \mathsf{hbC}(Z'), \mathsf{atC}(Z')]\!]_v^{\mathsf{NA}} \ \wedge \mathsf{code}(Z') = Y$$

- Now simplify using the definition of hbC and atC.

$$Y \in [\![B_2, \mathcal{A}, R, S]\!]_v^{\mathsf{NA}} \ \wedge \mathsf{hbC}(Z') = R = \mathsf{hbC}(Z) \wedge \mathsf{atC}(Z') = S = \mathsf{atC}(Z)$$

- Choose Y as the witness for our goal 11. Note that the presence of a safety violation in X or Y would contradict the safety of $C_X(B_2)$ and $C_X(B_1)$. It is left to show that:

$$\mathsf{hist}(X) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Y)$$

- Unfolding the definition of $\sqsubseteq_{\mathsf{h}}$, we have:

$$A(Z) = A(Z') \ \wedge \mathsf{hbL}(Z') \subseteq \mathsf{hbL}(Z)$$

  and it is left to show that,

$$A(X) = A(Y) \ \wedge \mathsf{hbL}(Y) \subseteq \mathsf{hbL}(X)$$

- Note that X and Y are the code-block projections of Z and Z' respectively, and we are done.

- **Case 2:** $\neg\mathsf{safe}(C_X(B_2))$.
  - Identify an unsafe valid execution of $C_X(B_2)$, Z', and specise the final conjunct of 12 with B_2 and Z' to get:

$$X' \in X^{\downarrow} \wedge W \in ([\![B_2, \mathcal{A}, R, S]\!]_v^{\mathsf{NA}})^{\downarrow} \ \wedge \neg\mathsf{safe}(W) \ \wedge \mathsf{hist}(X')^{\downarrow} \sqsubseteq_{\mathsf{h}} \mathsf{hist}(W)$$

  Then by the definition of $^{\downarrow}$, there exists a $W' \in [\![B_2, \mathcal{A}, R, S]\!]_v^{\mathsf{NA}}$ such that $W \in W'^{\downarrow}$, and this case is completed by noting that W' and W satisfy 11.

□

E.1 Context construction

Lemma 18 (Construction Lemma).

$$\begin{array}{l}
\forall B\, \mathcal{A}, R, S\, .\, \forall X \in [\![B, \mathcal{A}, R, S]\!]_v^{\mathsf{NA}}. \\
\quad \exists C_X.\, \exists Z \in [\![C_X(B)]\!]_v^{\mathsf{NA}}. \\
\quad\quad (\mathsf{code}(Z) = X) \wedge \\
\quad\quad (\mathsf{hbC}(Z) = R) \wedge (\mathsf{atC}(Z) = S) \wedge \\
\quad\quad \forall B'.\, \forall Z' \in [\![C_X(B')]\!]_v^{\mathsf{NA}}. \\
\quad\quad\quad ((A(\mathsf{contx}(Z)) = A(\mathsf{contx}(Z'))) \implies \mathsf{hist}(Z) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Z') \ \wedge \mathsf{at}(\mathsf{contx}(Z)) = \mathsf{at}(\mathsf{contx}(Z'))) \wedge \\
\quad\quad\quad (\neg\mathsf{safe}(Z') \implies \exists X' \in X^{\downarrow}.\, \exists W \in ([\![B', \mathcal{A}, R, S]\!]_v^{\mathsf{NA}})^{\downarrow}.\, \neg\mathsf{safe}(W) \wedge \mathsf{hist}(X') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(W))
\end{array}$$

Proof.

- Start by choosing an arbitrary B, $\mathcal{A}$, R, S and $X \in [\![B, \mathcal{A}, R, S]\!]$.

- Construct the client C_X as specified in Figure 9 with one minor change: have check halt the thread if the error variable is written.

  It remains to show that there exists $Z \in [\![C_X(B)]\!]_v^{\mathsf{NA}}$ such that:
  (1) $(\mathsf{code}(Z) = X) \wedge (\mathsf{hbC}(Z) = R) \wedge (\mathsf{atC}(Z) = S)$
  (2) $\forall B'.\, \forall Z' \in [\![C_X(B')]\!]_v^{\mathsf{NA}}$.
  $((\mathcal{A}(\mathsf{contx}(Z)) = \mathcal{A}(\mathsf{contx}(Z'))) \implies \mathsf{hist}(Z) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Z') \wedge \mathsf{at}(\mathsf{contx}(Z)) = \mathsf{at}(\mathsf{contx}(Z'))) \wedge$
  $(\neg\mathsf{safe}(Z') \implies \exists X' \in X^{\downarrow}.\, \exists W \in ([\![B', \mathcal{A}, R, S]\!]_v^{\mathsf{NA}})^{\downarrow}.\, \neg\mathsf{safe}(W) \wedge \mathsf{hist}(X') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(W))$
- We first establish 1.
  - Appealing to the thread local semantics and the structure of $C_X(B)$, choose Z_p, a pre-execution of $C_X(B)$ that does not write the error variable, and whose code projecion matches X.
  - Note that at is generated from the thread-local semantics matching X, and for the context part, each LL/SC pair is in its own thread, so there is only one way to link them.
  - Consruct *mo* as follows: for code actions choose these edges to match X, and for the context part, note that there is no choice: at each location there is only one write after the initialisation.
  - Construct rf as follows: for code actions choose these edges to match X, and for that context actions set rf to be coincident with an R edge in the case of Racq or from the initialisation write in the case of Nacq. Note that the context projection of happens-before matches R by construction.
  - Let Z be the combination of Z_p, mo and rf. Show that Z is valid.
    * The only axioms that could fail are *Acyclicity* over some Racq or *Coherence* over some Nacq.
    * In the first case, any cycle would be made up of code hb and R edges, and would also be a cycle in X, a contradiction.
    * A *Coherence* violation over some Nacq implies the existence of a hb edge from the associated Nrel to the Nacq. This violates the rules used to construct C_X, and is a contradiction.
- Now establish 2.
  - Start by choosing arbitrary B' and $Z' \in [\![C_X(B')]\!]_v^{\mathsf{NA}}$.
  - First, show that $(\mathcal{A}(\mathsf{contx}(Z)) = \mathcal{A}(\mathsf{contx}(Z'))) \implies \mathsf{hist}(Z) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Z') \wedge \mathsf{at}(\mathsf{contx}(Z)) = \mathsf{at}(\mathsf{contx}(Z'))$
    * Z does not write e, so neither does Z' (they have an equal context projection).
    * By construction of C_X, the histories of Z abstract the histories of Z' and the at relations match.
  - Now suppose Z' is unsafe. It remains to show:

$$\exists X' \in X^{\downarrow}.\, \exists W \in ([\![B', \mathcal{A}, R, S]\!]_v^{\mathsf{NA}})^{\downarrow}.\, \neg\mathsf{safe}(W) \wedge \mathsf{hist}(X') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(W)$$

    * C_X uses only atomic and local variables that cannot exhibit safety violations: each violation must be amongst the actions of $\mathcal{A}$ and the actions generated by B'.
    * Identify a safety violation in Z' and consider the prefix Z'_p containing only $\mathsf{hb} \cup \mathsf{rf}$ predecessors of the actions of the violation.
    * There are no writes to e in Z'_p: after any such write, the thread is stopped, so it cannot appear in the prefix Z'_p.
    * Below, we establish that for every thread of Z'_p except those that contain the safety violation from which it is constructed, the error variable is not written in the corresponding thread of Z'.
      · Consider the $\mathsf{hb} \cup \mathsf{rf}$ edges that draw actions into the prefix Z'_p from some thread t, there are two cases: the edge arises from a `Rrel`/`Racq` pair, or it is created by a read, from write w, in the code block or a context action.
      · In the first case e is not written in check or Nacq on t, because that would halt the thread before the call to `Rrel`.

  - In the second case, no call to `Nacq` writes e, and calls of `commit` only write to e in the case of a failing store conditional, contradiciting the existence of write w in Z', so w is only performed in threads that never write to e.
- From Z'_p, we construct W by applying the decomposition lemma Z'_p to get an execution W, completing this to an execution in $[\![B', \mathcal{A}, R, S]\!]_v^{\mathsf{NA}}$, and observing that W is in $([\![B', \mathcal{A}, R, S]\!]_v^{\mathsf{NA}})^{\downarrow}$. W is unsafe by construction.
- Take A to be the set of all context actions in W together with the call and ret actions present in W. Let X' be the projection of X to the $\mathsf{hb} \cup \mathsf{rf}$ predecessors of A, so that $X' \in X^{\downarrow}$.
- It remains to show that there is no edge in $\mathsf{hb}(W)$ between the actions of A that is not present in the guarantee of X'_p, G'. By construction of W, any extraneous $\mathsf{hb}(W)$ edge must end at one of the threads hosting the violation. There is only one code block, so at least one of the threads has to be executing a context action. There is no single action that both creates an incomming hb edge and causes a safety violation, so any additional $\mathsf{hb}(W)$ edge must end at the code block. In that case, W does not include the ret action following the racy action in hb, and neither does X', and there can be no new edge in G'.

$\square$