Compositional Verification of Relaxed-Memory Program Transformations

Mike Dodds Mark Batty Alexey Gotsman

Abstract

This paper is about verifying program transformations on a relaxed memory model of the kind used in C/C++. 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, SC fences, and non-atomics. We also define a variant of our denotation that is finite for loop-free blocks of code, 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

This paper is about verifying program transformations in a concurrent relaxed-memory setting. We define a new reasoning principle that characterises whether two blocks of code are related, and we use this principle in a new tool that automatically checks whether program transformations are valid.

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 the program without introducing new observable behaviours – for example, a compiler might optimise the program or a programmer might refactor it. Such transformations are called *observational refinements*, and they ensure that properties of the original program will carry over to the transformed version, without needing to be re-checked.

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 programming 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. In a sequential language, a code-block can only interact with the context at its start and end-points; but in a concurrent language, a block can be observed in intermediate states.

Our work applies to a *relaxed memory* concurrent setting, where 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. In this paper we focus on *axiomatic* memory models of the type used in C/C++ and Java. In these models, program executions are represented by structures of events 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 [18, 23].

Approach. Suppose we have a proposed transformation from code block P to Q. For any code-block we define a *denotation* which summarises its behaviour in a restricted representative context by a set of *histories*. We then compare the denotations of P and Q to prove that the transformation is valid. This approach is compositional: it requires reasoning only about the code-blocks and representative contexts; the validity of the transformation in an arbitrary program will follow.

More technically, histories track context accesses to variables used in the code-block and temporal relationships between these accesses. They abstract from the syntactic structure of the rest of the program, and numerous other relationships between accesses tracked by the memory model. We prove that this approach is *fully abstract*, meaning that it can verify any valid transformation: considering only representative contexts does not lose generality.

We also define a variant of our denotation that is *finite* for loop-free blocks of code, 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, we show that, it is unnecessary to consider executions where two distinct operations in the context read the same write.

Using this finite denotation, we have implemented a prototype verification tool, Stellite. Our tool converts an input transformation into a model in the Alloy language [10], and then checks that the transformation is valid using the Alloy* solver [14]. 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.

Contribution. Our contribution is twofold: we define a compositional, fully-abstract reasoning principle for axiomatic relaxed models; and using this we show it is feasible to automatically verify relaxed-memory program transformations. Our model is derived from C/C++ 2011 standard [1], and we handle many of its key features, including release-acquire, non-atomic accesses, and SC fences. However, our aim is not to handle C/C++ per se: 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:

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

If $P_1 \preccurlyeq_{\mathsf{pr}} P_2$, then any property of observable behaviour established of P_2 will also hold of P_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*. Code-blocks are analogous to C/C++ blocks, while contexts are analogous to programs with a single block-shaped 'hole'. 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 $B_1 \preccurlyeq_{\mathsf{bl}} B_2$. We define this by lifting the definition of $\preccurlyeq_{\mathsf{pr}}$:

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

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 semantics of the programming language is highly significant in determining which transformations are valid. For example, in a sequential setting, where only initial and final values of variables are observable, the following code-blocks are observationally equivalent, i.e., either of them refines the other:

B_1: `write(x,2); write(x,5)` | B_2: `write(x,5)`

However in a standard concurrent setting where the environment can observe intermediate states, $\mathtt{x} = 2$ can be observed in B_2 but not B_1, meaning the code-blocks are no longer equivalent. A relaxed-memory setting removes the idea of a single state seen by all threads, and so further complicates the notion of observation.

Compositional verification. To establish $B_1 \preccurlyeq_{\mathsf{bl}} B_2$, it is undesirable to examine every possible syntactic context: there are an infinite number, with large amounts of irrelevant structure. Instead, the verification process should *also* be compositional. Our approach is to construct a *denotation* for each code-block – a simplified, ideally finite, summary of all possible interactions between the block and its context. We then define a *refinement relation* on denotations and use it to establish observational refinement without examining every context.

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*, i.e.:

$$B_1 \preccurlyeq_{\mathsf{bl}} B_2 \implies B_1 \sqsubseteq B_2$$

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 Memory Model

Our language's memory model is derived from the C/C++ 2011 standard (henceforth '*C11*'), as formalized by Batty et

al [1, 3]. However, we simplify in several ways — see end of section for details. In this section we describe a model without non-atomic operations: we restore them in §7.

Relaxed memory. In a sequentially consistent (SC) concurrent system, there is a total temporal order on reads and writes, and reads take the value of the most recent write; in particular, they cannot read values that have been overwritten, or that are written in the future. A *relaxed* (or *weak*) memory system weakens this total order, which allows behaviour forbidden under SC.

A standard relaxed behaviour is *store buffering*:

```
      write(x,0); write(y,0);
write(x,1);     || write(y,1);          (SB)
v1 := read(y);  || v2 := read(x);
```

Perhaps counter-intuitively, in most relaxed models $\texttt{v1} = \texttt{v2} = 0$ is a possible post-state. This cannot occur on an SC system: if $\texttt{v1} = 0$ then `write(y,1)` must be ordered after the read of `y`, which would order `write(x,1)` before the read of `x`, forcing it to assign $\texttt{v2} = 1$.

Another important example is *message passing*:

```
      write(f,0); write(x,0);
write(x,1);     || b := read(f);          (MP)
write(f,1);     || if (b==1)
                ||   r := read(x);
```

In many relaxed models, $\texttt{b} = 1 \wedge \texttt{r} = 0$ is a possible post-state. 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 $0 \in \mathsf{Val}$. We assume that each thread uses the same set of local variable names LVar (though the identity of local variables with the same name used in different threads is distinct).

The syntax of the programming language is as follows:

$$\begin{aligned} C \;::=\;& l := E \mid \texttt{write}(x, l) \mid l := \texttt{read}(x) \mid \\ & l := \texttt{RMW}(x, l_1, l_2) \mid l := \texttt{LL}(x) \mid l' := \texttt{SC}(x, l) \mid \\ & \texttt{fence} \mid C_1 \parallel C_2 \mid C_1; C_2 \mid \\ & \texttt{if}\,(l)\,\{C_1\}\,\texttt{else}\,\{C_2\} \mid \texttt{while}\,(l)\,\{C\} \mid \{-\} \\ E \;::=\;& l \mid l_1 = l_2 \mid l_1 \neq l_2 \mid \ldots \end{aligned}$$

Many of the constructs are standard. $\texttt{RMW}(x, l_1, l_2)$ is a *read-modify-write*. It reads the value of x and compares it to the value in l_1. If they are equal, then it assigns x to the value in l_2 and returns 1; otherwise it returns 0. $\texttt{LL}(x)$ and $\texttt{SC}(x, l)$ are load-link and store-conditional, which are used by many platforms to implement `RMW`. A load-link behaves as a standard read, but if it is followed by a store-conditional to the same location, the store fails if there are intervening writes to the same location. The `fence` command is an SC fence: interleaving such fences between all statements in a program would guarantee sequential consistent behaviour.

The construct $\{-\}$ represents a block-shaped hole in the program. The set Prog of *whole programs* consists of programs without holes, while the set Contx of *contexts* consists of programs with a single hole. To simplify our presentation, we assume that holes in a context only appear in loop-free positions. 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 single block-shaped hole syntactically replaced by B.

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:

$$\mathsf{Actions} \;\triangleq\; \mathsf{ActID} \times \mathsf{Kind} \times \mathsf{Option}(\mathsf{GVar}) \times \mathsf{Val}^*$$

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 read, write, ll, sc, and thieir failed variants ll_f and 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. The kind of the action dictates the number of variables and values: each read or write accesses one global variable and has one value.

Given an action v, we use $\mathsf{kind}(v)$, $\mathsf{gvar}(v)$ and $\mathsf{var}(v)$ as selectors for the different fields. We often write actions so as to elide action identifiers and the option type. For example, $\mathsf{read}(x, 3)$ stands for $\exists i.\, (i, \mathsf{read}, \mathsf{Just}(x), [3])$. We also sometimes elide values. Given a set of actions $\mathcal{A}$, we write e.g. $\mathsf{reads}(\mathcal{A})$ and $\mathsf{writes}(\mathcal{A})$ to identify actions of each kind in $\mathcal{A}$. We range over actions by u, v, read actions by r and write actions by w.

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[1] 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.

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. We show selected clauses in Figure 1. The full semantics is given in §A.

[1] We do not consider infinite computations in this paper, since their semantics in axiomatic memory models is not settled in the literature [5].

$$\langle \texttt{write}(x,l),\sigma\rangle \triangleq \{(\{\mathsf{write}(x,a)\},\emptyset,\sigma) \mid \sigma(l)=a\}$$
$$\langle l := \texttt{read}(x),\sigma\rangle \triangleq \{(\{\mathsf{read}(x,a)\},\emptyset,\sigma[l\mapsto a]) \mid a\in\mathsf{Val}\}$$
$$\langle C_1;C_2,\sigma\rangle \triangleq \{(\mathcal{A}_1 \uplus \mathcal{A}_2, \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup (\mathcal{A}_1\times\mathcal{A}_2),\sigma_2) \mid (\mathcal{A}_1,\mathsf{sb}_1,\sigma_1)\in\langle C_1,\sigma\rangle \wedge (\mathcal{A}_2,\mathsf{sb}_2,\sigma_2)\in\langle C_2,\sigma_1\rangle\}$$

Figure 1. Read, write, and sequential composition in the thread-local semantics. The full semantics is given in §A. 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.

To track the values of local variables in the thread-local semantics, we use maps $\sigma \in \mathsf{VMap} \triangleq \mathsf{LVar} \to \mathsf{Val}$. The thread-local semantics is defined using a function

$$\langle -,-\rangle \colon \mathsf{Prog}\times\mathsf{VMap} \to \mathcal{P}(\mathsf{PreExec}\times\mathsf{VMap})$$

Given a program P and an initial local variable map σ, $\langle P,\sigma\rangle$ yields the set of pre-executions of P paired with final variable maps and final open load-linked regions. Note that in Figure 1, the clause for `read`, a is unrestricted, and hence the read can take any value in Val.

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. The `RMW` command is encoded either as a plain read, or as a successful ll/sc pair. The `fence` command is encoded by a successful ll/sc pair to a distinguished variable *fen* $\in \mathsf{GVar}$ that is not otherwise read or written.[2]

Let σ_0 map every local variable to 0, then the thread-local semantics of a program $P \in \mathsf{Prog}$ is defined as follows:

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

Execution structure. The semantics of a program P is a set $[\![P]\!]$ of *executions* $X = (\mathcal{A},\mathsf{sb},\mathsf{at},\mathsf{mo},\mathsf{rf},\mathsf{hb}) \in \mathsf{Exec}$, where $(\mathcal{A},\mathsf{sb})$ is a pre-execution and the relations $\mathsf{at},\mathsf{mo},\mathsf{rf},\mathsf{hb} \subseteq \mathcal{A}\times\mathcal{A}$. 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.

- *Reads-from* (rf) is an injective map from write and sc actions to read, ll and ll_f actions. A write and reading 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 and sc actions to each distinct variable. This is a per-variable order in which *all* threads observe writes; two threads cannot observe writes to a variable in different orders.
- *Atomicity* ($\mathsf{at} \subseteq \mathsf{sb}$) is a bijection which associates each successful load-link action to a successful store-conditional on the same location.
- *Happens-before* (hb) is the closest the memory model has to global time – however, unlike the SC 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 $X \in \mathsf{Exec}$ that are compatible with the thread-local semantics $\langle P\rangle$ and that satisfy certain *validity axioms*, denoted $\mathsf{valid}(X)$:

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

The 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: $\neg\exists w_1,w_2.\ w_1 \underset{\mathsf{mo}}{\overset{\mathsf{hb}}{\rightleftarrows}} w_2$

  This axiom requires that the the order in which writes to a location become visible to threads cannot contradict the temporal order. However, note that writes may be ordered in mo but not hb.

- COHERENCE: $\neg\exists w_1,w_2,r.\ w_1 \xrightarrow{\mathsf{mo}} w_2 \xrightarrow{\mathsf{hb}} r$, with $w_1 \xrightarrow{\mathsf{rf}} r$

  This axiom generalises the SC prohibition on reading overwritten values. If two writes are ordered in mo, then intuitively the second overwrites the first. A read / RMW that follows some write in hb or mo cannot read from writes earlier in mo – these earlier writes have been overwritten. However, unlike in SC, hb is partial, so there may be multiple writes that an action can legally read.

- RFVAL:

$$\forall r.\ (\neg\exists w'.\ w' \xrightarrow{\mathsf{rf}} r) \implies (\mathsf{rval}(r)=0 \wedge (\neg\exists w.\ w \xrightarrow{\mathsf{hb}} r \wedge \mathsf{var}(w)=\mathsf{var}(r)))$$

  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

[2] This provides a stronger semantics than C/C++11; here we follow Lahav et al. [12], who argue this strengthening is sound under existing compilation strategies to common multiprocessors.

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.

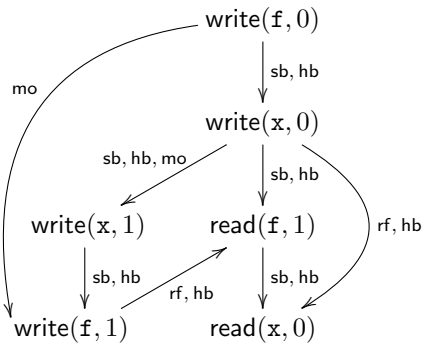
- ATOM: $\neg\exists w_1, l, w_2, s.\ w_1 \xrightarrow[\mathsf{rf}]{} l \overset{\mathsf{mo}}{\underset{\mathsf{at}}{\rightrightarrows}} w_2 \xrightarrow{\mathsf{mo}} s$

  For load-link store conditional-pairs, this axiom (adapted from Lahav et al. [12]) ensures that there is no mo-intervening write that would invalidate the store.

- NOFAIL:
  $\neg\exists l, f, s \in .\ l \overset{\mathsf{sb}}{\underset{\mathsf{at}}{\rightrightarrows}} f \xrightarrow{\mathsf{sb}} s\ \wedge\ \mathsf{gvar}(l) = \mathsf{gvar}(f) \wedge f \in (\mathsf{ll}_f(\mathcal{A}) \cup \mathsf{sc}_f(\mathcal{A}))$

  This final axiom ensures that ll/sc pairs cannot have intervening failed accesses to the same location.

Example executions. Our model forbids the problematic MP relaxed behaviour discussed above. We can try to construct an execution exhibiting the relaxed behaviour $\mathtt{b} = 1 \wedge \mathtt{r} = 0$ as follows. To avoid clutter, we omit hb edges obtained by transitivity.



(MP exec)

Although this execution is allowed by the thread-local semantics of the MP program, it is ruled out by the COHERENCE axiom. As hb is transitively closed, there is a derived hb edge $\mathsf{write}(\mathtt{x}, 1) \xrightarrow{\mathsf{hb}} \mathsf{read}(\mathtt{x}, 0)$, which forms a COHERENCE violation.

$$\mathsf{write}(\mathtt{x}, 0) \xrightarrow{\mathsf{mo}} \mathsf{write}(\mathtt{x}, 1) \xrightarrow{\mathsf{hb}} \mathsf{read}(\mathtt{x}, 0) \quad (\mathsf{rf})$$

Thus this is not an execution of the MP program. Indeed, any execution ending in $\mathsf{read}(\mathtt{x}, 0)$ is forbidden for the same reason, meaning that the MP relaxed behaviour cannot occur.

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:

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

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).

Differences from C11. Our model is derived from Batty et al's C11 formalization [3], with a number of simplifications.

We omit SC accesses to reduce complexity in the history. We also assume a static set of shared variables, and omit pointers. In both cases, we believe these limitations could be be lifted at the cost of more complexity and more challenging automation: see §10 for a discussion.

We omit RLX accesses to avoid well-known problems with thin-air values [5]. We know of at least four current proposals for fixing these problems, and it would be premature to select one.

We add LL-SC atomic instructions in addition to C11's RMW. This increases the context's observational power and gives us a full abstraction result. LL/SC is commonly available as a hardware instruction on platforms supporting C11.

4. Denotations of Code-Blocks

An obvious way to generate the denotation of a code-block would be to quantify over all its contexts. However, we show one needs only to consider a set of limited representative contexts. (In §5 we show that even these contexts can be cut down further, giving a finite denotation for code-blocks whose set of thread-local executions is finite.) We construct the denotation for a code-block in two steps:

1. Generate the *block-local* executions of the code-block. These are executions under a set of special cut-down contexts that are sufficient to represent interactions of all possible contexts with the block.
2. From these, extract an interaction summary called a *history*. The denotation is the set of all such histories.

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. We capture this with special action $\mathsf{call}(\sigma)$ at the start of the block, and $\mathsf{ret}(\sigma')$ at the

end, where $\sigma, \sigma' : \mathsf{LVar} \to \mathsf{Val}$ record the values of local variables at these points. Assume that variables in LVar are ordered: $l_1, l_2, \ldots, l_n$. Then call has the following encoding as an action, with fresh identifier i:

$$\mathsf{call}(\sigma) \triangleq (i, \mathsf{call}, \mathsf{None}, [\sigma(l_1), \ldots \sigma(l_n)])$$

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:

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

  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 consider only cases where LL/SC pairs do not cross block boundaries, so we need not consider boundary-crossing at edges.

When constructing block-local executions, we represent all possible interactions by quantifing over all possible choices of σ, σ', $\mathcal{A}_B$, R_B and S_B.

Together, call, ret, $\mathcal{A}_B$, R_B, and S_B represent a limited context, stripped of syntax, relations at, rf, sb, and mo, and actions on global variables other than VS_B. The set $[\![B, \mathcal{A}_B, R_B, S_B]\!]$ contains all executions of B under this special limited context. Formally, an execution $X = (\mathcal{A}, \mathsf{sb}, \mathsf{mo}, \mathsf{rf}, \mathsf{at}, \mathsf{hb})$ is in this set if:

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
   - $(\mathcal{A}_l, \mathsf{sb}_l, \sigma') \in \langle B, \sigma \rangle$
   - $\mathcal{A}_l = \mathcal{A} \setminus (\mathcal{A}_B \cup \{\mathsf{call}, \mathsf{ret}\})$
   - $\mathsf{sb} = \mathsf{sb}_l \cup \{(\mathsf{call}, u), (u, \mathsf{ret}) \mid u \in \mathcal{A}_l\}$

   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′, ATOM′, and NOFAIL′. We define HBDEF′ as:

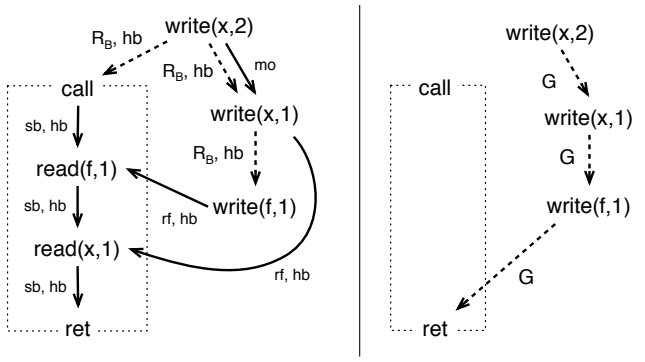


Figure 2. Left: block-local execution. Right: associated history.

$$\text{HBDEF}' \colon \ \mathsf{hb} = (\mathsf{sb} \cup \mathsf{rf} \cup R_B)^+ \textit{ and } \mathsf{hb} \text{ is acyclic.}$$

That is, the context relation R_B is added to hb. ATOM′ and NOFAIL′ are 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 choices that are consistent with the code-block. The *block-local executions* of B are all executions $X \in [\![B, \mathcal{A}_B, R_B, S_B]\!]$ for consistent choices of $\mathcal{A}_B$, R_B, S_B.

Example block-local execution. The left side of Figure 2 shows a block-local execution for the code-block

```
l1 := read(f); l2 := read(x)        (5)
```

Here the set VS_B of accessed global variables is $\{\mathtt{f}, \mathtt{x}\}$, the context action set $\mathcal{A}_B$ consists of the three writes, and the context relation 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 read of $\mathtt{f} = 1$:

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

Because hb includes sb, rf, and R_B, there is a transitive edge $\mathsf{wr}(\mathtt{x}, 1) \xrightarrow{\mathsf{hb}} \mathsf{rd}(\mathtt{x}, 1)$. The edge $\mathsf{wr}(\mathtt{x}, 2) \xrightarrow{\mathsf{mo}} \mathsf{wr}(\mathtt{x}, 1)$ is forced because the HBVSMO axiom prohibits mo from contradicting hb. Consequently, the COHERENCE axiom forces the program to read $\mathtt{x} = 1$. This is a product of the interaction between code, reads of context actions $\mathcal{A}_B$, and hb edges in R_B.

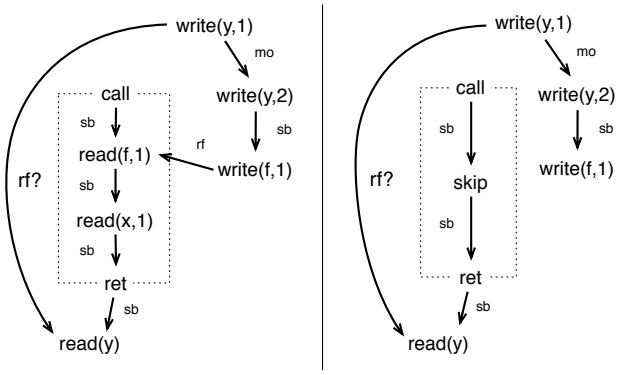
Histories. From any block-local execution X, its *history* summarises the interactions between the code-block and the context. Informally, the history records hb over context actions, call, and ret. More formally the history, written $\mathsf{hist}(X)$, is a pair (A, G) consisting of an action set A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

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

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 (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 2 shows the history corresponding to the block-local execution on the left.

To see why the history needs to record the guarantee at all, compare the code-block `read(f); read(x)` with a block consisting of just `skip`. Note that neither modifies any variables. Below we give executions of both blocks in a context that also reads and writes to a variable y.



On the left, the context read of y cannot take value 1 because there is a path $\mathsf{wr}(\mathtt{y},1) \xrightarrow{\mathsf{mo}} \mathsf{wr}(\mathtt{y},2) \xrightarrow{\mathsf{hb}} \mathsf{rd}(\mathtt{y})$ created by the read of f[3]. This means that reading $\mathtt{y} = 1$ contradicts the COHERENCE axiom. Conversely, on the right there is no such path and the context read of y can read 1. Thus these two code-blocks have different effects on the context through happens-before, even though they are identical in their writes to variables. The guarantee tracks these kinds of hb effects.

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.

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

To allow us to compare the denotations of two code-blocks, we first define a *refinement relation* on histories:

$$(A_1, G_1) \sqsubseteq_{\mathsf{h}} (A_2, G_2) \iff A_1 = A_2 \wedge G_2 \subseteq G_1$$

[3] The read of f could also take the initial value, but to illustrate the effects of hb we assume this does not happen.

Here the history (A_2, G_2) places fewer restrictions on the context than (A_1, G_1). This matches our definition of observational refinement in §3 – a weaker guarantee corresponds to more observable behaviours.

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

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

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 $\mathcal{A}, R, S$ to ensure that the behaviour can be replicated in the same context.

THEOREM 1 ($\sqsubseteq_{\mathsf{q}}$ adequacy). $B_1 \sqsubseteq_{\mathsf{q}} B_2 \implies B_1 \preccurlyeq_{\mathsf{bl}} B_2$

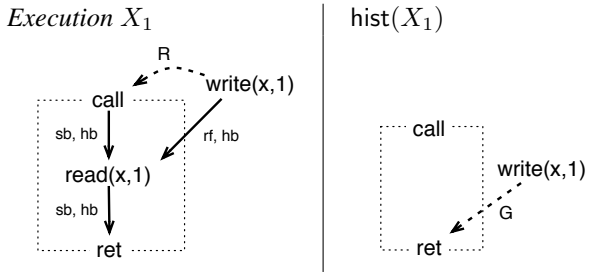
THEOREM 2 (Full abs. of $\sqsubseteq_{\mathsf{q}}$). $B_1 \preccurlyeq_{\mathsf{bl}} B_2 \implies B_1 \sqsubseteq_{\mathsf{q}} B_2$.

We prove Theorem 1 in §B, and discuss Theorem 2 in §8. As a corollary these two theorems, a program transformation $B_2 \rightsquigarrow B_1$ is valid if and only if $B_1 \sqsubseteq_{\mathsf{q}} B_2$ holds.

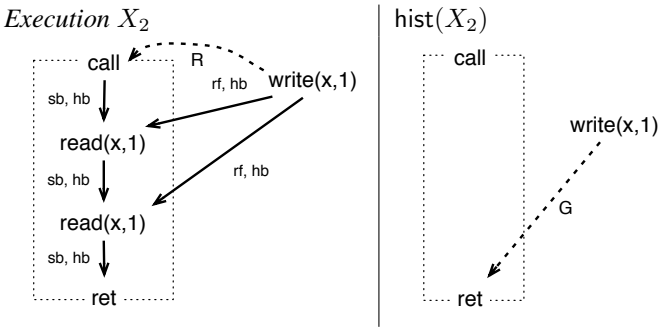
Validating a transformation. We now apply our approach to a simple program transformation. To verify this transformation, we require that $B_1 \sqsubseteq_{\mathsf{q}} B_2$.

B_2: `l:=read(x); l:=read(x)` $\rightsquigarrow$ B_1: `l:=read(x)`

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 write $\mathtt{x} = 1$, a context relation R relating the write to call, and an empty S relation. This choice of R forces the read to read from the context write:



We can exhibit an execution $X_2 \in [\![B_2, \mathcal{A}, R, S]\!]$ with a matching history by making both reads read from the same context write:



5. A Finite Denotation

The approach described above simplifies $\mathcal{A}/R/S$ contexts, but there are still infinitely many for any code-block, so we cannot simply enumerate in order to verify a transformation. To solve this, we modify our denotation. This gives us finiteness in some cases, meaning we can automatically check transformations (see our tool in §6). However, this comes at a cost: the new 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 that satisfy a predicate cut. Intuitively, the denotation must consider each *pattern* of context behaviour, but it need not consider every execution. For example, we need not have two context reads reading the same code-block write: one is sufficient.
- 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.

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

THEOREM 3 ($\sqsubseteq_{\mathsf{c}}$ adequacy). $B_1 \sqsubseteq_{\mathsf{c}} B_2 \implies B_1 \preccurlyeq_{\mathsf{bl}} B_2$.

The proof of adequacy is given 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, for example for loop-free blocks. Note that we also 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 the set* $\{X \mid \exists \mathcal{A}.\, X \in [\![B, \mathcal{A}, \emptyset, S]\!] \wedge \mathsf{cut}(X)\}$.

Cutting predicate. We first identify the actions in an 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 $\mathsf{code}(X)$:

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

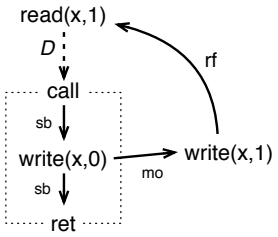
The predicate cut is the conjunction of cutR for reads, and cutW for writes. We consider a successful LL-SC pair as a single operation: if cut allows us to keep one half of the LL-SC, we can implicitly keep the other.

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

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.

The non-visible writes in cutW are required for adequacy (§D): they represent contexts where the block interleaves between a context write and read in mo.

Extended history ($\mathsf{hist}_{\mathsf{E}}$). 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 following block-local execution[4]:



It represents a set of larger execution contexts, but it cannot be embedded into a context that generates the dashed edge D as a hb. We represent such 'forbidden' edges D into a separate history component called a *deny*.

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

[4] We use this execution to illustrate the extended history, but in fact cut would not allow the context read.

below: $G, D \subseteq A \times A$.

$$\{(u,v) \mid \mathsf{HBvsMO\text{-}d}(u,v) \vee \mathsf{Cohere\text{-}d}(u,v) \vee \mathsf{RFval\text{-}d}(u,v)\} \cap \begin{pmatrix} (\mathsf{contx}(X) \times \mathsf{contx}(X)) \cup \\ (\mathsf{contx}(X) \times \{\mathsf{call}\}) \cup (\{\mathsf{ret}\} \times \mathsf{contx}(X)) \end{pmatrix}$$

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:

$\mathsf{HBvsMO\text{-}d}(u,v)$: $\exists w_1, w_2.\ w_1 \xrightarrow{\mathsf{hb}^*} u \overset{D}{\cdots\!\!>} v \xrightarrow{\mathsf{hb}^*} w_2$, with $w_2 \xrightarrow{\mathsf{mo}} w_1$

$\mathsf{Coherence\text{-}d}(u,v)$: $\exists w_1, w_2, r.\ w_1 \xrightarrow{\mathsf{mo}} w_2$, $w_2 \xrightarrow{\mathsf{hb}^*} u \overset{D}{\cdots\!\!>} v \xrightarrow{\mathsf{hb}^*} r$, $w_1 \xrightarrow{\mathsf{rf}} r$

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

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:

$$(A_2, G_2, D_2) \sqsubseteq_{\mathsf{E}} (A_2, G_2, D_2) \overset{\Delta}{\iff} A_1 = A_2 \wedge G_2 \subseteq G_1 \wedge D_2 \subseteq D_1$$

Counter-example to fully abstraction. Finiteness has a cost: the modified denotation is not fully abstract. To see this, consider blocks, B_1 : `skip` and B_2 : $\mathtt{read}(x)$. It is easy to see that $B_1 \sqsubseteq_{\mathsf{q}} B_2$ holds: the new read can read from either a hb-earlier write, 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 read 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. The problem is that the $\sqsubseteq_{\mathsf{c}}$ does not distinguish the position of W in hb, which removes precision from the denotation.

6. Verification Tool

In this section we describe Stellite[5], a tool that checks our $\sqsubseteq_{\mathsf{c}}$ relation using the Alloy system [10].

Input transformations are written in a simple C-like language. Stellite supports transformations with atomic reads and writes, and SC fences. It does not yet support NA accesses, ll/sc or branching control-flow, but these would not present fundamental difficulties. The Alloy encoding would be similar, albeit with a downclosure on histories, and the increased search space arising from this downclosure.

Stellite converts an input transformation $B_2 \rightsquigarrow B_1$ into an Alloy* model encoding the check $B_1 \sqsubseteq_{\mathsf{c}} B_2$. We use the higher-order Alloy* solver because standard Alloy cannot handle the existential quantification on histories in the definition of $\sqsubseteq_{\mathsf{c}}$.

As previously noted [24], there is a natural fit between Alloy models and C/C++-style axiomatic memory models. The actions of an execution, as well as locations, values, and all other basic notions are modelled as Alloy objects. Values are modelled abstractly, tracking only equality or inequality, which avoids having to enumerate them explicitly. The relations hb, rf, sb etc. are modelled by Alloy relations. Finally, validity axioms are modelled by Alloy predicates.

If a counter-example is discovered, the execution and history of B_1 can be viewed using the Alloy model visualiser, which has a similar appearance to the diagrams in this paper. Understanding why no corresponding history of B_2 exists is left to the user, but is usually obvious. As $\sqsubseteq_{\mathsf{c}}$ is not fully abstract, this counter-example could of course be spurious.

The Alloy* search is parameterised by the maximum size of the model it will examine. For this reason, Stellite is a bounded checking tool: it only examines histories up to a specified size, measured in total number of actions. However, our cutting predicate bounds the size of histories that must be considered, meaning that a sufficiently large bound guarantees that tool results are sound. Given a query $B_1 \sqsubseteq_{\mathsf{c}} B_2$, the bound needed varies depending on the number of internal actions on distinct locations in B_1. In our experiments we ran the tool with size bound of 10, sufficient to give validity all the optimisations we consider. Note that most transformations do not require such a large bound, and execution times improve considerably if it is reduced.

Experimental results. Figure 3 gives results for different transformations, many derived from [23] – we test all the examples from [23] that fit into our input language. Transformations of the sort that we check have led to bugs in GCC [15] and LLVM [8].

Two of the transformations cause the tool to timeout. This does not establish validity, but as with other bounded model-checking work, our experience is that counter-examples are found at shallow positions in the search space.

Note that some transformations are invalid because of their effect on local variables, e.g. $\mathtt{skip} \rightsquigarrow l := \mathtt{read}(x)$. The closely related transformation $\mathtt{skip} \rightsquigarrow \mathtt{read}(x)$ throws away the result of the read, and is consequently valid.

7. Transformations with Non-Atomics

We now extend our memory model and denotation to non-atomic (i.e. unsynchronised) accesses. In C11, these are

[5] *Stellite* is a cobalt-chromium alloy designed for wear-resistance.

Transformation	Valid?	Time (s)
Introduction:		
$\texttt{skip} \rightsquigarrow \texttt{fence}$	✓	76
$\texttt{skip} \rightsquigarrow \texttt{rd}(x)$	✓	429
$\texttt{skip} \rightsquigarrow l := \texttt{rd}(x)$	×	18
$l := \texttt{rd}(x) \rightsquigarrow l := \texttt{rd}(x); \texttt{wr}(x, l)$	×	72
$l := \texttt{rd}(x) \rightsquigarrow l := \texttt{rd}(y); l := \texttt{rd}(x)$	✓	∞
$l := \texttt{rd}(x) \rightsquigarrow l := \texttt{rd}(x); l := \texttt{rd}(x)$	✓	19851
$\texttt{wr}(x, l) \rightsquigarrow \texttt{wr}(x, l); \texttt{wr}(x, l)$	×	136
$\texttt{fence} \rightsquigarrow \texttt{fence}; \texttt{fence}$	✓	248
Elimination:		
$\texttt{fence} \rightsquigarrow \texttt{skip}$	×	15
$l := \texttt{rd}(x) \rightsquigarrow \texttt{skip}$	×	17
$l := \texttt{rd}(x); \texttt{wr}(x, l) \rightsquigarrow l := \texttt{rd}(x)$	×	64
$l := \texttt{rd}(x); l := \texttt{rd}(x) \rightsquigarrow l := \texttt{rd}(x)$	✓	2378
$\texttt{wr}(x, l); l := \texttt{rd}(x) \rightsquigarrow \texttt{wr}(x, l)$	✓	9118
$\texttt{wr}(x, m); \texttt{wr}(x, l) \rightsquigarrow \texttt{wr}(x, l)$	✓	24696
$\texttt{fence}; \texttt{fence} \rightsquigarrow \texttt{fence}$	✓	382
Exchange:		
$\texttt{fence}; l := \texttt{rd}(x) \rightsquigarrow l := \texttt{rd}(x); \texttt{fence}$	×	26
$\texttt{fence}; \texttt{wr}(x, l) \rightsquigarrow \texttt{wr}(x, l); \texttt{fence}$	×	50
$l := \texttt{rd}(x); \texttt{fence} \rightsquigarrow \texttt{fence}; l := \texttt{rd}(x)$	×	79
$\texttt{wr}(x, l); \texttt{fence} \rightsquigarrow \texttt{fence}; \texttt{wr}(x, l)$	×	145
$l := \texttt{rd}(x); \texttt{wr}(y, m) \rightsquigarrow \texttt{wr}(y, m); l := \texttt{rd}(x)$	×	28
$\texttt{rd}(y); \texttt{rd}(x) \rightsquigarrow \texttt{rd}(x); \texttt{rd}(y)$	×	59
$m := \texttt{rd}(y); l := \texttt{rd}(x) \rightsquigarrow l := \texttt{rd}(x); m := \texttt{rd}(y)$	×	118
$\texttt{wr}(y, m); l := \texttt{rd}(x) \rightsquigarrow l := \texttt{rd}(x); \texttt{wr}(y, m)$	✓	∞
$\texttt{wr}(y, m); \texttt{wr}(x, l) \rightsquigarrow \texttt{wr}(x, l); \texttt{wr}(y, m)$	×	641

Figure 3. Results for our checking tool for a range of examples. ✓ and × denote whether the transformation satisfies $\sqsubseteq_{\mathsf{c}}$. 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.

intended to enable sequential compiler optimisations that would be unsound in a concurrent context. Soundness is ensured by declaring any concurrent read-write or write-write pair to the same location a *data race*, which causes the whole program to have an undefined behaviour.

Memory model with non-atomics. Non-atomic reads and writes are added to the model by introducing new commands $\texttt{write}_{\mathsf{NA}}(x, l)$ and $l := \texttt{read}_{\mathsf{NA}}(x)$ and the corresponding kinds of actions: $\mathsf{write}_{\mathsf{NA}}, \mathsf{read}_{\mathsf{NA}} \in \mathsf{Kind}$. NA is set of all actions of these kinds. We partition global variable so they are either only accessed by non-atomics, or by atomics. 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: $\neg\exists w_1, w_2, r \in \mathsf{NA}.\; w_1 \xrightarrow{\mathsf{hb}} w_2 \xrightarrow{\mathsf{hb}} r$, $w_1 \xrightarrow{\mathsf{rf}} r$

Modification order (mo) does not cover non-atomic accesses, and we change the definition of happens-before (hb), so that non-atomic reads 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 example execution MP in §3. If all of the actions were made non-atomic, then no happens-before would be created, and by RFHBNA both reads would have to read from the initialisation.

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

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

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

$$[\![P]\!] \triangleq \begin{cases} [\![P]\!]^{\mathsf{NA}}_v, & \forall X \in [\![P]\!]^{\mathsf{NA}}_v.\, \mathsf{safe}(X) \\ \top, & \text{otherwise} \end{cases}$$

Thus, if the set of executions of P satisfying the validity axioms does not contain an unsafe execution, then semantics of P is this set; otherwise it is the distinguished value $\top$ representing undefined behaviour.

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

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

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 $\sqsubseteq^{\mathsf{NA}}_{\mathsf{q}}$, an extended refinement relation sensitive to non-atomics. Our relation satisfies the following theorems, proved in §B and §E.

THEOREM 5 ($\sqsubseteq^{\mathsf{NA}}_{\mathsf{q}}$ adequacy). $B_1 \sqsubseteq^{\mathsf{NA}}_{\mathsf{q}} B_2 \Rightarrow B_1 \preccurlyeq^{\mathsf{NA}}_{\mathsf{bl}} B_2$

THEOREM 6 ($\sqsubseteq^{\mathsf{NA}}_{\mathsf{q}}$ full abs). $B_1 \preccurlyeq^{\mathsf{NA}}_{\mathsf{bl}} B_2 \Rightarrow B_1 \sqsubseteq^{\mathsf{NA}}_{\mathsf{q}} B_2$

We first introduce the *downclosure* operator $X^{\downarrow}$, which constructs the set of $(\mathsf{hb} \cup \mathsf{rf})^+$ prefixes of an execution X:

$$X^{\downarrow} \triangleq \{X' \mid \exists A.\, X' = X|_A \wedge \forall (a, a') \in (\mathsf{hb}(X) \cup \mathsf{rf}(X))^+.\, a' \in A \Rightarrow a \in A\}$$

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

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

There are essentially two cases in this definition: either the X_2' we choose is safe or unsafe.

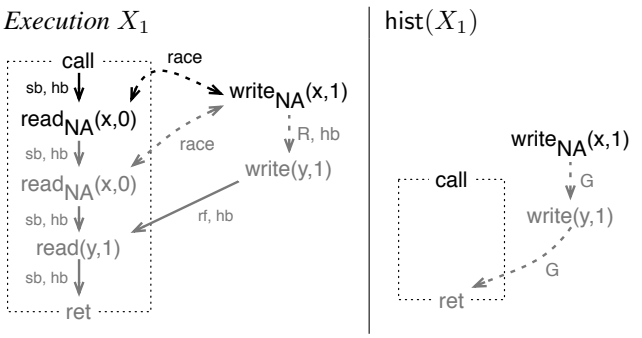
- X_2' **is safe.** In this case the situation resembles $\sqsubseteq_{\mathsf{q}}$, i.e. X_2' is a complete, not prefixed, execution, and it has a related history to X_1. If the code-block B_2 is guaranteed to be safe, for example because it has no non-atomic accesses, the above definition is equivalent to $\sqsubseteq_{\mathsf{q}}$.
- X_2' **is unsafe.** In this case, any program $C(B_2)$ which has X_2' as a sub-execution has semantics $\top$. Therefore, we need only match histories *up to the point that* X_2' *becomes unsafe*. This ensures the unsafety will actually occur in the (unknown) whole program $C(B_2)$. In $\sqsubseteq_{\mathsf{q}}^{\mathsf{NA}}$, if X_2' unsafe, then it can be a prefix, not a complete execution. We then must witness a prefix X_1' of X_1 with a corresponding history. After this point, X_1 and X_2 can behave entirely differently.

  This prefixing in the definition of $\sqsubseteq_{\mathsf{q}}^{\mathsf{NA}}$ is required for full abstraction—it would be adequate to always require complete executions with related histories.

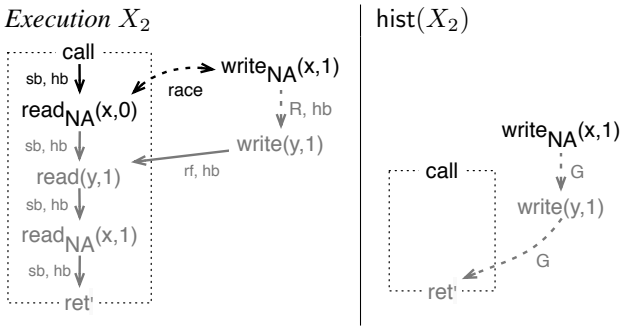
Validating a transformation. Consider the following *anti-roach-motel* transformation:

$$
\begin{array}{lcl}
B_2 \colon \mathtt{l1} := \mathtt{read}_{\mathsf{NA}}(\mathtt{x}); & \rightsquigarrow & B_1 \colon \mathtt{l1} := \mathtt{read}_{\mathsf{NA}}(\mathtt{x}); \\
\quad \mathtt{l2} := \mathtt{read}(\mathtt{y}); & & \quad \mathtt{l3} := \mathtt{read}_{\mathsf{NA}}(\mathtt{x}); \\
\quad \mathtt{l3} := \mathtt{read}_{\mathsf{NA}}(\mathtt{x}); & & \quad \mathtt{l2} := \mathtt{read}(\mathtt{y});
\end{array}
$$

To verify the transformation, we must establish that $B_1 \sqsubseteq_{\mathsf{q}}^{\mathsf{NA}} B_2$. 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 write of $\mathtt{x} = 1$ and an atomic write of $\mathtt{y} = 1$, and a context relation R relating the write of x to the write of y:



Note the data races between reads and a write over x. We can exhibit an 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 read of y takes the value of the context write, so CoherNA forces the second read of x to read from the context write 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 a prefix X_1' of X_1. We have that $\mathsf{hist}(X_1') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(X_2')$, even though the histories $\mathsf{hist}(X_1)$ and $\mathsf{hist}(X_2)$ do not correspond. Reasoning in this way shows that the transformation is valid.

Finite denotation with NA. We have also defined a finite variant of $\sqsubseteq_{\mathsf{q}}^{\mathsf{NA}}$, using the cutting strategy described in §5. We defer the details to §C.

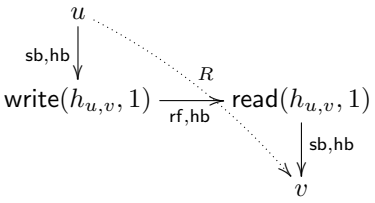
8. Full Abstraction

We give a proof of full abstraction (Theorems 2 and 6) in §E. Our proof depends on constructing a special syntactic context that is sensitive to one particular history. Given an execution X produced from a block B, we write C_X for the resulting special context. The construction of 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, this this 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.

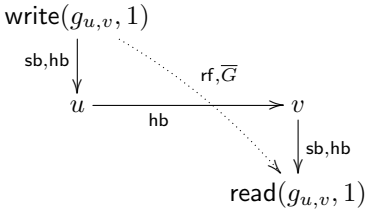
Context construction. The definition of C_X is given in §E – here we sketch its behaviour. C_X executes the context operations from X in parallel with the block. It wraps these operations in auxiliary wrapper code to enforce R and check the history. If wrapper code fails, it writes to an error variable, which thereby alters the observable behaviour.

The context must generate edges in R. This is enforced by wrappers $\mathtt{Rrel}_{u,v}$ and $\mathtt{Racq}_{u,v}$, one for each edge $(u, v) \in R$. Each wrapper uses a watchdog variable to create a hb-edge: if $\mathtt{Racq}_{u,v}$ does not read the value written by $\mathtt{Rrel}_{u,v}$, then the error variable is written. A successful read gives the

following shape:



The context must also prohibit history edges beyond those in the original guarantee G. This is checked by wrappers $\texttt{Nrel}_{u,v}$ and $\texttt{Nacq}_{u,v}$, one for each (u, v) *not* in G. Once again happens-before edges are detected using watchdog variables. If $\texttt{Nacq}_{v,u}$ *does* read the value written by $\texttt{Nrel}_{v,u}$, then there is an errant history edge, and the error location is written. An erroneous execution has the following shape (omitting the write to the error location):



Our context construction requires LL/SC, not just C11's standard RMW. This is because it is necessary to inject wrapper code between LL and SC in order to detect and enforce hb-edges. This is not possible with RMW alone.

9. Related Work

Our approach builds on Batty et al. [4], who generalise linearizability [9] to the C11 memory model. They represent interactions between a library and its clients by sets of histories consisting of a guarantee and a deny; we do the same for code and context. However, we cannot assume *information hiding*: Batty et al. assume that the variables used by the library cannot be directly accessed by clients. Also, we establish both adequacy and full abstraction, propose a finite denotation, and build an automated verification tool.

Our approach is broadly similar to Brookes's seminal concurrency semantics [6]. In both cases, a code block is represented by a denotation capturing possible interactions with an abstracted context. In [6], 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.

In order to achieve full abstraction, Brookes assumes a powerful `await()` instruction which blocks until the global state satisfies a boolean expression. Our full abstraction result does not require this: 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 [7]; TSO [11]; and SC-DRF [17]. Also, [7, 17] define tools for verifying program transformations. All three approaches are based on traces, and are therefore not directly portable to C11-style axiomatic models. All three also target substantially stronger (i.e. more restrictive) relaxed models than C11.

Methods for verifying code transformations, either manually or using proof assistants, have been proposed for several relaxed models: TSO [19, 20, 22], Java [18] and C/C++ [23]. 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. There has also been various work on automatically verifying compiler optimisations under sequential consistency [13, 16, 25].

Several tools exist for testing (not verifying) program transformations on axiomatic models: for example Morisset et al. [12] for GCC, and Chakraborty et al. [8] for LLVM. Similarly, Alloy is used in the MemSAT tool for simulation of the Java memory model [21]. Our Alloy encoding of the memory model is also similar to the input files for the Herd/Cat memory model simulator [2].

10. Conclusions and Extensions

We have proposed the first compositional method for verifying program transformation on a C/C++-style memory model. Our method only requires reasoning about the code-block being transformed in minimal contexts, which simplifies proofs and allows push-button verification of transformations. We have defined two variants on our approach with different motivations: one that achieves maximal precision by guaranteeing full abstraction and one that supports automatic verification by guaranteeing finiteness.

Extensions. We omit SC and RLX accesses, but both are handled in [4]. We believe very similar techniques could be applied. For SC this would require an extra relation in the history. For RLX, this requires a non-compositional safety check. Note that [4] gives RLX an unrealistically strong semantics as a stopgap solution the thin-air problem.

Our language is based on shared variables, not addressable memory, so for example we cannot write `y:=*x; z:=*y`. To support addressable memory, the block-local execution construction would need to quantify over actions on all memory locations, not just a static variable set VS. The rest of our theory would remain the same however, because C11-style models grant no special status to pointer values. Cutting down to a finite denotation would then require some extra abstraction over memory.

References

[1] *Programming Languages — C++*. 2011. ISO/IEC JTC1 SC22 WG21.

[2] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *PLDI*, page 7, 2014.

[3] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.

[4] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, pages 235–248, 2013.

[5] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *ESOP*, pages 283–307, 2015.

[6] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

[7] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *CC*, 2010.

[8] S. Chakraborty and V. Vafeiadis. Validating optimizations of concurrent C/C++ programs. In *CGO*, pages 216–226, 2016.

[9] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[10] D. Jackson. *Software Abstractions – Logic, Language, and Analysis*. MIT Press, revised edition, 2012.

[11] R. Jagadeesan, G. Petri, and J. Riely. Brookes is relaxed, almost! In *FOSSACS*, pages 180–194, 2012.

[12] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In *POPL*, pages 649–662, 2016.

[13] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *PLDI*, pages 22–32, 2015.

[14] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *ICSE*, pages 609–619, 2015.

[15] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, pages 187–196, 2013.

[16] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified peephole optimizations for CompCert. In *PLDI*, pages 448–461, 2016.

[17] D. Poetzl and D. Kroening. Formalizing and checking thread refinement for data-race-free execution models. In *TACAS*, 2016.

[18] J. Sevcík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, pages 27–51, 2008.

[19] J. Sevcík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *POPL*, pages 43–54, 2011.

[20] J. Sevcík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, 2013.

[21] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: checking axiomatic specifications of memory models. In *PLDI*, pages 341–350, 2010.

[22] V. Vafeiadis and F. Zappa Nardelli. Verifying fence elimination optimisations. In *SAS*, pages 146–162, 2011.

[23] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220, 2015.

[24] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. Automatically comparing memory consistency models. In *POPL*, 2017.

[25] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *PLDI*, pages 175–186, 2013.

A. Collected Definitions

Execution observational refinement

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

Program observational refinement

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

Program observational refinement with NA

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

Block observational refinement

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

History abstraction

$$(A_1, G_1) \sqsubseteq_{\mathsf{h}} (A_2, G_2) \ \stackrel{\Delta}{\iff} \ A_1 = A_2 \wedge G_2 \subseteq G_1$$

Quantified abstraction

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

Extended history abstraction

$$(A_2, G_2, D_2) \sqsubseteq_{\mathsf{E}} (A_2, G_2, D_2) \ \stackrel{\Delta}{\iff} \ A_1 = A_2 \wedge G_2 \subseteq G_1 \wedge D_2 \subseteq D_1$$

Cut abstraction

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

Cut predicates

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

Execution downclosure

$$X^{\downarrow} \ \triangleq \ \{X' \mid \exists A.\, X' = X|_A \wedge \forall (a, a') \in (\mathsf{hb}(X) \cup \mathsf{rf}(X))^{+}.\, a' \in A \Rightarrow a \in A\}$$

Quantified abstraction with NA

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

Thread-local semantics. 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.

$$
\begin{aligned}
\langle \mathtt{write}(x,l),\sigma\rangle &\triangleq \{(\{\mathsf{write}(x,a)\},\emptyset,\sigma) \mid \sigma(l)=a\}\\
\langle l := \mathtt{read}(x),\sigma\rangle &\triangleq \{(\{\mathsf{read}(x,a)\},\emptyset,\sigma[l\mapsto a]) \mid a\in\mathsf{Val}\}\\
\langle l := \mathtt{RMW}(x,l_1,l_2),\sigma\rangle &\triangleq \{(\{r,w\},(r,w),\sigma[l\mapsto 1]) \mid \sigma(l_1)=a\wedge\sigma(l_2)=b\wedge r=\mathsf{ll}(x,a)\wedge w=\mathsf{sc}(x,b)\}\cup\\
&\qquad \{(\{\mathsf{read}(x,a)\},\emptyset,\sigma[l\mapsto 0]) \mid \sigma(l_1)\neq a\}\\
\langle l := \mathtt{LL}(x),\sigma\rangle &\triangleq \{(\{\mathsf{ll}(x,a)\},\emptyset,\sigma[l\mapsto a]) \mid a\in\mathsf{Val}\}\ \cup\ \{(\{\mathsf{ll}_f(x,a)\},\emptyset,\sigma[l\mapsto a]) \mid a\in\mathsf{Val}\}\\
\langle l' := \mathtt{SC}(x,l),\sigma\rangle &\triangleq \{(\{\mathsf{sc}(x,a)\},\emptyset,\sigma[l'\mapsto 1]) \mid \sigma(l)=a\}\ \cup\ \{(\{\mathsf{sc}_f(x)\},\emptyset,\sigma[l'\mapsto 0])\}\\
\langle \mathtt{fence},\sigma\rangle &\triangleq \{(\{r,w\},(r,w),\sigma) \mid r=\mathsf{ll}(\mathit{fen},0)\wedge w=\mathsf{sc}(\mathit{fen},0)\}\\
\langle C_1 \parallel C_2,\sigma\rangle &\triangleq \{(\mathcal{A}_1\uplus\mathcal{A}_2,\mathsf{sb}_1\cup\mathsf{sb}_2,\sigma) \mid (\mathcal{A}_1,\mathsf{sb}_1,\sigma_1)\in\langle C_1,\sigma\rangle\wedge(\mathcal{A}_2,\mathsf{sb}_2,\sigma_2)\in\langle C_2,\sigma\rangle\}\\
\langle C_1;C_2,\sigma\rangle &\triangleq \{(\mathcal{A}_1\uplus\mathcal{A}_2,\mathsf{sb}_1\cup\mathsf{sb}_2\cup(\mathcal{A}_1\times\mathcal{A}_2),\sigma_2) \mid\\
&\qquad (\mathcal{A}_1,\mathsf{sb}_1,\sigma_1)\in\langle C_1,\sigma\rangle\wedge(\mathcal{A}_2,\mathsf{sb}_2,\sigma_2)\in\langle C_2,\sigma_1\rangle\}\\
\langle \mathsf{if}(l)\,\{C_1\}\,\mathsf{else}\,\{C_2\},\sigma\rangle &\triangleq \begin{cases}\langle C_2,\sigma\rangle, & \text{if } \sigma(l)=0\\ \langle C_1,\sigma\rangle, & \text{otherwise}\end{cases}
\end{aligned}
$$

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:
  $$\langle\{-\},\sigma\rangle \quad\triangleq\quad \{(\{c,r\},\{c\to r\},\sigma') \mid c=\mathsf{call}(\sigma)\wedge r=\mathsf{ret}(\sigma')\}$$
  - X satisfies HBDEF', ATOM′, NOFAIL′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', ATOM with ATOM′ and NOFAIL with NOFAIL′.

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:

$$(a,b) \in \mathsf{hb}(l) \quad \Longrightarrow \quad (a,b) \in (\mathsf{sb}(l) \cup \mathsf{rf}_{\mathsf{AT}}(l) \cup R)^+ \tag{8}$$

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

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′, NOFAIL′, 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. Read-modify-writes 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, ATOM and NOFAIL. (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 and NOFAIL, 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:

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

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:

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

- 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

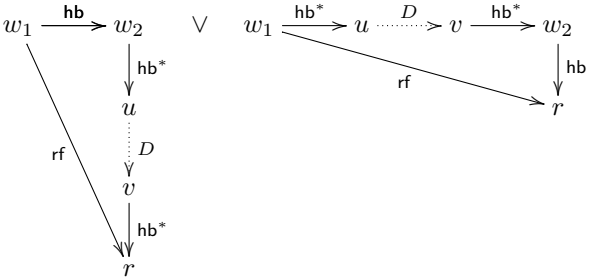
We now define $\sqsubseteq_{\mathsf{c}}^{\mathsf{NA}}$, a refinement between denotations which includes both cutting and non-atomics.

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

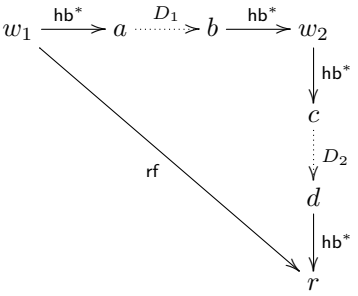
$$u, a, c \in \mathsf{ret} \cup \mathsf{interf}(X) \qquad\qquad v, b, d \in \mathsf{call} \cup \mathsf{interf}(X)$$

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.

$\mathsf{CoNA\text{-}d}(u, v)$: $\exists w_1, w_2, r \in \mathsf{NA}.$



$\mathsf{CoNA\text{-}d2}(a,b,c,d)$: $\exists w_1, w_2, r \in \mathsf{NA}.$



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:

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

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

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

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

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

We then define *cut abstraction* as follows:

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

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, NOFAIL 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 $Y \in [\![B_2, \mathcal{A}, R]\!]_{vr}^{\downarrow}$ 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) $Y'' \in [\![B_2, \mathcal{A}, R, S]\!]_{vr}^{\downarrow}$; (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:

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

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

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

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 $X \in [\![B_1, \mathcal{A}, S]\!]_d^{\downarrow}$. 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 $X' \in [\![B_1, \mathcal{A}, S]\!]_d^{\downarrow}$.
  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 $\mathsf{hist}_{\mathsf{E}}(X') \sqsubseteq_{\mathsf{E}} \mathsf{hist}_{\mathsf{E}}(Y')$.
  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 $\mathsf{hist}_{\mathsf{E}}(X) \sqsubseteq_{\mathsf{E}} \mathsf{hist}_{\mathsf{E}}(Y)$.
- 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 $\mathsf{hist}_{\mathsf{E}}(X) \sqsubseteq_{\mathsf{E}} \mathsf{hist}_{\mathsf{E}}(Y)$, 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, NOFAIL 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). $B_1 \sqsubseteq_{\mathsf{c}}^3 B_2 \implies B_1 \sqsubseteq_{\mathsf{c}}^2 B_2$

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). $B_1 \sqsubseteq_{\mathsf{c}} B_2 \implies B_1 \sqsubseteq_{\mathsf{c}}^3 B_2$

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). $B_1 \sqsubseteq_{\mathsf{c}} B_2 \implies B_1 \sqsubseteq_{\mathsf{d}}^{\mathsf{NA}} B_2$

Proof. Prove this as a sequence of implications:

$$B_1 \sqsubseteq_{\mathsf{c}} B_2 \implies B_1 \sqsubseteq_{\mathsf{c}}^3 B_2 \implies B_1 \sqsubseteq_{\mathsf{c}}^2 B_2 \implies B_1 \sqsubseteq_{\mathsf{c}}^1 B_2 \implies B_1 \sqsubseteq_{\mathsf{d}}^{\mathsf{NA}} B_2$$

Each implication step is proved in a lemma above. □

E. Proof of Theorem 6

E.1 Proof structure.

Assume $B_1 \preccurlyeq_{\mathsf{bl}} B_2$; we have to prove $B_1 \sqsubseteq_{\mathsf{q}} B_2$. At the high level, we prove this with the following steps:

1. Following (7), consider arbitrary $\mathcal{A}$, R, S and $X_1 \in [\![B_1, \mathcal{A}, R, S]\!]$.
2. We use X_1 to construct a special context C. 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 detects a mismatch with X_1, it writes to a special observable error variable $e \in \mathsf{OVar}$. The context C is constructed in such a way that for any code-block B' and any execution $Y \in [\![C(B')]\!]$ in which e is not written, the following 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) $\mathsf{hb}(Y)$ does not include any of the edges in the complement of the guarantee in $\mathsf{hist}(X_1)$.
3. We show that there is an execution $Z_1 \in [\![C(B_1)]\!]$ 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 $B_1 \preccurlyeq_{\mathsf{bl}} B_2$, applying (2) to the context C, we get an execution $Z_2 \in [\![C(B_2)]\!]$ where e is never written.
5. By the construction of C, we have (a) and (b). Using this fact, we construct an execution $X_2 \in [\![B_2, \mathcal{A}, R, S]\!]$ where the actions generated by B_2 match those in Z_2 and the call and ret actions match those in X_1. Let $\mathsf{hist}(X_1) = (A_1, G_1)$ and $X_2 = (A_2, G_2)$. Using (a), we show $A_1 = A_2$ 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 (7), the required $B_1 \sqsubseteq_{\mathsf{q}} B_2$.

E.2 Full abstraction with non-atomics

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

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

$$\begin{array}{l} \exists Y \in [\![B_2, \mathcal{A}, R, S]\!]_v^{\downarrow}. \\ \quad \exists X' \in X^{\downarrow}. \mathsf{hist}(X') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Y) \wedge \\ \qquad \mathsf{safe}(Y) \implies (\mathsf{safe}(X) \wedge (X' = X) \wedge Y \in [\![B_2, \mathcal{A}, R, S]\!]_v) \end{array} \tag{10}$$

• Apply the construction lemma to B_1, R, S and X to find a context C_X, and an execution Z:

$$\begin{array}{l} Z \in [\![C_X(B_1)]\!]_v \wedge \\ \mathsf{code}(Z) = X \wedge \\ \mathsf{hbC}(Z) = R \wedge \mathsf{atC}(Z) = S \wedge \\ \forall B'. \forall Z' \in [\![C_X(B')]\!]_v. \\ \quad ((A(\mathsf{contx}(Z)) = A(\mathsf{contx}(Z'))) \implies (\mathsf{hist}(Z) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Z')) \wedge \mathsf{at}(\mathsf{contx}(Z)) = \mathsf{at}(\mathsf{contx}(Z'))) \wedge \\ \quad (\neg\mathsf{safe}(Z') \implies \exists X' \in X^{\downarrow}. \exists W \in [\![B', \mathcal{A}, R, S]\!]_v^{\downarrow}. \neg\mathsf{safe}(W) \wedge \mathsf{hist}(X') \sqsubseteq_{\mathsf{h}} \mathsf{hist}(W)) \end{array} \tag{11}$$

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

$$\begin{array}{l} (\quad \neg\mathsf{safe}(C_X(B_2)) \vee \\ \quad (\mathsf{safe}(C_X(B_1)) \wedge \\ \qquad \forall X \in [\![C_X(B_1)]\!]_v. \exists Y \in [\![C_X(B_2)]\!]_v. \\ \qquad\quad \mathcal{A}(X|_{V_{C_X}}) = \mathcal{A}(Y|_{V_{C_X}})) \wedge \\ \qquad\quad \mathsf{hb}(X|_{V_{C_X}}) \subseteq \mathsf{hb}(Y|_{V_{C_X}}))) \end{array} \tag{12}$$

and then case split on $\mathsf{safe}(C_X(B_2))$.

• **Case 1:** $\mathsf{safe}(C_X(B_2))$.

- By 12, 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 11, we have:

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

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

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

- Now simplify using the definition of hbC and atC.

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

- Choose Y as the witness for our goal 10, and X as the witness for the inner quantification. 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 11 with B_2 and Z' to get:

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

  This case is completed by noting that W satisfies 10.

□

F. Proof of Theorems 2 and 6

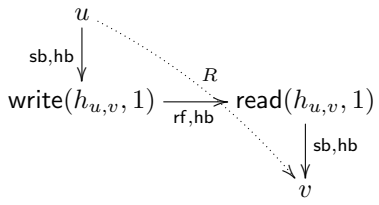
The proof of full abstraction relies on a special context construction that is sensitive enough to pick apart any observable difference in one execution and another. The construction is built from one execution, replicating context hb and at, or writing a context error variable. The context ensures that when composed with any block, all executions with matching context hb and at produce histories that are abstractions of the original execution. For unsafe executions, this history abstraction need only hold over a hb-prefix that includes a safety violation.

The context construction. Figure 4 describes the construction of context C from an execution $X \in [\![B, \mathcal{A}, R, S]\!]$ (for brevity, we use syntactic sugar that elides manipulations of local variables). The context is a parallel composition of threads: one for the parameter code-block $\{-\}$, one for each ll/sc pair in at, ensuring replication of S, and one each for all other actions in $\mathcal{A}$ — these are collectively ranged over by m in Fig. 4.

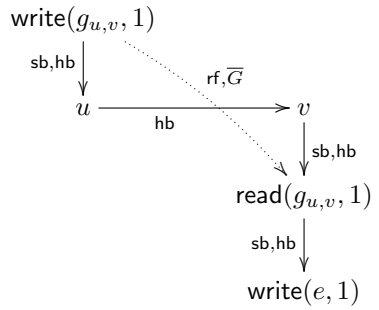
We introduce functions `call` and `ret` on the indices m, mapping $\{-\}$ to the call and ret actions in X, respectively, and acting as the identity otherwise. The construction then consists of several wrapper functions:

1. Innermost is `check`(m): for a read or a write action $u \in \mathcal{A}$, `check`(u) executes the corresponding operation and, in the case of a read or a sc, compares the value recorded with the one specified by u. The command `check`$(\{-\})$ initialises local variables to the values specified by the call action in X, runs the parameter 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 and halts the thread.

2. The wrappers $\texttt{Rrel}_m$ and $\texttt{Racq}_m$ replicate R. 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 $u = \texttt{ret}(m)$; $\texttt{Racq}_m$ is constructed symmetrically. These wrappers use watchdog variables $h_{u,v}$ to create happens-before by writing to and reading as in the MP test of §3. If $\texttt{Racq}_{u,v}$ does *not* read the value written by $\texttt{Rrel}_{u,v}$, then it writes to the error variable. Otherwise, $\texttt{Rrel}_{u,v}$ is sequenced after u and $\texttt{Racq}_{u,v}$ before v, reproducing R with an execution of the following shape:



3. The wrappers $\texttt{Nrel}_m$ and $\texttt{Nacq}_m$ ensure that the history of an error-free execution of $C(B')$ abstracts the history of the original execution, i.e. there are no new guarantee edges beyond those of the original G. Each $\texttt{Nrel}_m$ is built up of a sequence of invocations of $\texttt{Nrel}_{u,v}$, one for each edge (u, v) that might be in the history, but is not covered by R or G. $\texttt{Racq}_m$ is constructed symmetrically. Errant happens-before edges are detected using watchdog variables $g_{u,v}$, again relying on the mechanics of MP (§3): $\texttt{Nrel}_{u,v}$ and $\texttt{Nacq}_{u,v}$ respectively write to and read from $g_{u,v}$. If $\texttt{Nacq}_{v,u}$ *does* read the value written by $\texttt{Nrel}_{v,u}$, then there is a new guarantee edge, e is written. $\texttt{Rrel}_{u,v}$ is sequenced *before* u and $\texttt{Racq}_{u,v}$ *after* v, so an erroneous execution has the following shape:



Here, the RFVAL axiom (§3) forces $\texttt{Nacq}_{u,v}$ to read from $\texttt{Nrel}_{u,v}$ and write to e. Consequently, a non-erroneous execution does not contain errant happens-before edges. As required by the theorem.

To illustrate the construction, we use the execution X in Figure 2, for B defined by (5). The set $\mathcal{A}$ consists of the three writes outside the dashed rectangle, and R is given by the dashed edges. The edge $(\mathsf{write}(\mathtt{x}, 2), \mathsf{call}) \in R$ is reproduced by the invocation of $\texttt{Racq}_{\mathsf{write}(\mathtt{x},2),\mathsf{call}}$ on the first thread and $\texttt{Rrel}_{\mathsf{write}(\mathtt{x},2),\mathsf{call}}$ on the second. The edges from G that we need to consider are $(\mathsf{call}, \mathsf{write}(\mathtt{f}, 1))$ and $(\mathsf{call}, \mathsf{write}(x, 1))$, and the edge $(\mathsf{call}, \mathsf{write}(\mathtt{f}, 1)) \in \overline{G}$ is covered by the invocation of $\texttt{Nrel}_{\mathsf{call},\mathsf{write}(\mathtt{f},1)}$ on the first thread and $\texttt{Nacq}_{\mathsf{call},\mathsf{write}(\mathtt{f},1)}$ on the fourth.

F.1 Construction Lemma with non-atomics, with read-modify-writes

LEMMA 18 (Construction Lemma).

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

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 4 with one minor change: have `check` halt the thread if the error variable is written.

It remains to show that there exists $Z \in [\![C_X(B)]\!]_v$ such that:

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

Figure 4. Context construction.

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

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

Figure 5. Example of the `check` function and the context construction for the execution in Figure 2.

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

- 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.
  - Consruct at as follows: for code actions choose these edges to match X, and for the context part, note that there is no choice: 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 $Z' \in [\![C_X(B')]\!]_v$.

▪ First, show that $(A(\mathsf{contx}(Z)) = A(\mathsf{contx}(Z'))) \implies \mathsf{hist}(Z) \sqsubseteq_{\mathsf{h}} \mathsf{hist}(Z') \wedge \mathsf{at}(\mathsf{contx}(Z)) = \mathsf{at}(\mathsf{contx}(Z'))$

- 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:

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

- 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 $[\![B', \mathcal{A}, R, S]\!]_v$, and observing that W is in $[\![B', \mathcal{A}, R, S]\!]_v^{\downarrow}$. 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'.

□