Maximality of Atomic Causality

Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu

Department of Computer Science, University of Illinois at Urbana-Champaign
201 N. Goodwin, Urbana, IL 61801
Email: {tserban2,fengchen,grosu}@cs.uiuc.edu

Abstract.

1 Introduction

2 Traces, linearizations, and causality

Events represent atomic steps observed in the execution of a program. In this paper, we focus on multi-threaded programs and consider the following types of events (other types can be easily added): write/read of variables, and acquiring/releasing of locks. A statement in the program may produce multiple events. Events need to store enough information about the program state to allow the observer to analyze the trace.

To allow for a general treatment of events, we here consider events to be abstract entities from an infinite “collection” of events. Events can be “observed” through (partial) attribute mappings from events to a concrete domain. We will describe events by a tuple of pairs attribute:value listing the value of each defined attribute mapping for the particular event. The only attribute mappings considered in this paper will be: thread – the thread generating the event, type – the type of the event, target – the memory location accessed by the event, and state – the value read/written by the current event. For example, the description of an event could be 
\( e_1 : (\text{thread} = t_1, \, \text{type} = \text{write}, \, \text{target} = x, \, \text{state} = 1) \), which says that event \( e_1 \) is a write on location \( x \) with value 1, produced by thread \( t_1 \).

Note that the identity of an event is not reduced to the identity of the attributes defined for that event, so we could have two events with identical attributes, yet distinct.

**Definition 1.** A trace \( \tau = e_1 e_2 \cdots e_n \) is a finite ordered sequence of distinct events. Let \( \mathcal{E}_\tau = \{e_1, e_2, \ldots, e_n\} \) be the alphabet of \( \tau \) and let \( <_\tau \) be the total order induced by \( \tau \) on \( \mathcal{E}_\tau \).

The thread ordering \( <^t_\tau \) on \( \tau \) is given by \( e <^t_\tau e' \) if \( e <_\tau e' \) and \( \text{thread}(e) = \text{thread}(e') \). The restriction of \( \tau \) to thread \( i \), written \( \pi_i(\tau) \), is the trace obtained from \( \tau \) by erasing all events \( e \) such that \( \text{thread}(e) \neq i \).

A linearization of a set of events \( E \) is a trace \( \tau \) such that \( \mathcal{E}_\tau = E \). An interleaving of a trace \( \tau \) is a linearization \( \tau' \) of \( \mathcal{E}_\tau \) such that \( <_{\tau'} \) includes the thread ordering of \( \tau \). Let interleavings(\( \tau \)) be the set of all interleavings of \( \tau \).
The following could be considered as an alternative definition of interleavings. The proof follows trivially from the fact that $<_\tau^\tau$ can be partitioned in a set of total orders, one for each thread.

**Proposition 1.** $\tau'$ is an interleaving of $\tau$ iff

$$\mathcal{E}_{\tau'} = \mathcal{E}_\tau \text{ and } <_{\tau'} = <_\tau^\tau.$$ 

Given a trace $\tau$ observed during the execution of a system, there might be multiple linearizations of its alphabet which could be observed on the system under different interleavings of the threads. Classical happens-before dependence, originally introduced in the context of distributed systems [2], was used to detect concurrency bugs for concurrent systems [3–5], based on the fact that all interleavings of an observed trace which preserve the happens-before dependence could be potential executions of the system.

![Fig. 1. WR dependence](image)

### 2.1 Write-Read dependence

The most obvious kind of happens-before dependence is the write-read dependence [6]. Given two events $e_1, e_2 \in \mathcal{E}_\tau$, $e_2$ write-read depends on $e_1$ in $\tau$ iff $e_1$ is a write of a location $x$ and $e_2$ is a read of $x$ such that $e_1$ is the latest write on $x$ that happens-before $e_2$. Formally,

**Definition 2.** $e_2$ write-read depends on $e_1$ in $\tau$, written $e_1 \sqsubseteq_{wr} e_2$, if $\tau = \tau_1 e_1 \tau_2 e_2 \tau_3$, $\text{target}(e_1) = \text{target}(e_2)$, $\text{type}(e_1) = \text{write}$, $\text{type}(e_2) = \text{read}$, and for all $e \in \mathcal{E}_{\tau_2}$, either $\text{target}(e) \neq \text{target}(e_1)$, or $\text{type}(e) \neq \text{write}$.

An alternative intuition for the write-read dependence is that $e_1 \sqsubseteq_{wr} e_2$ iff the value read by event $e_2$ is the value written by $e_1$.

Happens-before approaches usually enforce a read-write dependence to guarantee that no write event on the same location could occur between a write event and the read event which depends on it. This requirement was relaxed in [6] by noticing that one only needs to guarantee that each set containing one
A linearization $\tau'$ of $\mathcal{E}_\tau$ preserves the write-read dependence of $\tau$ if for any two events $e_1, e_2 \in \mathcal{E}_\tau$, $e_1 \sqsubset_{wr} e_2$ iff $e_1 \sqsubset_{wr} e_2$.

The above definition basically says that any linearization $\tau'$ of $\tau$ preserving the write-read dependence, will also preserve all atomic write-read sets.

2.2 On lock atomicity

In [6] it is argued that lock atomicity can be handled through usual write-read dependence, by regarding lock acquire as a write event and lock release as a read event, both on the locking object. Although this might be true for the case when all acquired locks are released before the end of the trace, this approach does not work in general. For example, consider the case when dealing with partial traces as when doing on-line analysis of a system. In that case, we would like to ensure that an acquire event with no subsequent release event (at the moment the trace is considered) would be the last acquire event on that lock in any sound linearization of the original trace.

To have a more faithful treatment of lock acquire/release operations we choose to handle them as proper types of events. The set of possible event types is therefore enriched with types acquire and release. This enables the definition of an important concurrency concept, namely that of mutual exclusion:

**Definition 4.** A trace $\tau$ satisfies mutual exclusion iff for any $e_1, e_2$ such that $\tau = \tau_1 e_1 \tau_2 e_2 \tau_3$, $\text{type}(e_1) = \text{type}(e_2) = \text{acquire}$, and $\text{target}(e_1) = \text{target}(e_2)$, there exists $e \in \mathcal{E}_{\tau_2}$ such that $\text{target}(e) = \text{type}(e_1)$, $\text{type}(e) = \text{release}$.

For simplicity we have assumed in the above definition that there is only one resource that can be acquired. However, we believe the definition could be easily extended to encompass counter-based synchronization objects such as semaphores. One can notice some similarities between the write-read dependence and the mutual exclusion property. This is due to the fact that, in order to ensure the preservation of the write-read dependence, one need to execute a write event atomically (w.r.t. other write events on the same location) with its depending reads. This situation is similar to that encountered in the context of structured locks where a block guarded by a lock is executed atomically w.r.t other blocks guarded by the same lock.

One could see that the above definition works better for incomplete traces, by requiring that an acquired lock be released before it is re-acquired. However, in the context of structured locks and complete traces (in which each acquired lock is released), mutual exclusion could be subsumed by write-read dependence as suggested in [6].
2.3 The atomic causal model

The atomic causal model is an abstraction of a trace consisting of two partial orders: the thread ordering giving the sequential execution of each thread, and the write-read dependence giving the happens-before relation between events. Let us now introduce the central concept of this paper, namely the causal feasibility, i.e., the feasibility with the atomic causal model.

Definition 5. A interleaving of $\tau$ preserving the write-read dependence and satisfying the mutual exclusion property is termed a causally feasible linearization of $\tau$.

In [6] it is implicitly assumed that for any causally feasible linearization of a trace there is an execution of the multi-threaded system generating it. However, as in all other happens-before based techniques we know of, no actual proof of feasibility is presented, this making the improvements of one or another technique seem ad-hoc and giving no theoretical information about the coverage of the technique. This paper presents a first such result, regarding the feasibility of the atomic causal model. We will prove – for a specific language, but with generic conditions – not only that causally atomic feasible linearizations can be obtained as executions of the system, but also that, if one only considers the semantics of the language, the set of obtained feasible linearizations is maximal for the observed execution.

3 CIMP: A Concurrent Imperative Language

Let us consider a very simple concurrent imperative language. The imperative part consists from arithmetic expressions with integers and variables, comparison, statements such as assignment conditional, or loop. The concurrency is enabled by the fact that multiple statements can be put in parallel, by means of the $\parallel$ construct. Synchronized blocks are supported by the sync construct and are guarded by locks. Locks and variables partition the set of names in two distinct classes. Variables are uses only for storage properties, while locks are only used for synchronization. To avoid extra-counters on each lock, we sill statically reject programs with nested synchronization blocks protected by the same lock. For simplicity, there is no language construct supporting dynamic creation of threads.

The rational for choosing this toy language is that it is the simplest language exhibiting concurrent behaviours, yet it is complex enough to allow the extrapolation of our results to real programming languages.

Before discussing the semantics, let us define two operations: one to select a thread from a program by its index, and another to replace the thread statement at a given index by a given statement.

Definition 6. Given a program $p$ and an integer $i$, let $\pi_i(p)$ denote the projection of $p$ to thread $i$, partially defined by $\pi_1(Stmt \parallel Pgm) = Stmt$, and $\pi_{i+1}(Stmt \parallel Pgm) = \pi_i(Pgm)$, if $i > 0$. 


SYNTAX

Int := all integer numbers
Var := all variable identifiers
Lock := all lock identifiers
Name := Var | Lock
AExp := Int | Var | AExp + AExp
BExp := AExp ≤ AExp
Stmt := skip | Stmt; Stmt | Var := AExp
| if BExp then Stmt else Stmt | while BExp do Stmt
Pgm := nil | Stmt || Pgm

SEMANTICS

\[\langle A_1, \sigma, \tau, \text{td} \rangle \rightarrow \langle A'_1, \sigma, \tau' \rangle\]
\[\langle A_2, \sigma, \tau, \text{td} \rangle \rightarrow \langle A'_2, \sigma, \tau' \rangle\]
\[\langle I_1 + I_2, \sigma, \tau, \text{td} \rangle \rightarrow \langle I_1 + \text{int}_I I_2, \sigma, \tau, \text{td} \rangle\]
\[\langle I_1 \leq \text{int}_I I_2, \sigma, \tau, \text{td} \rangle \rightarrow \langle I_1 \leq \text{int}_I I_2, \sigma, \tau, \text{td} \rangle\]
\[\langle X := A, \sigma, \tau, \text{td} \rangle \rightarrow \langle X := A', \sigma, \tau' \rangle\]
\[\langle \text{if} B \text{ then } \text{St}_1 \text{ else } \text{St}_2, \sigma, \tau, \text{td} \rangle \rightarrow \langle \text{if } B' \text{ then } \text{St}_1 \text{ else } \text{St}_2, \sigma, \tau' \rangle\]
\[\langle \text{while } B \text{ St}, \sigma, \tau, \text{td} \rangle \rightarrow \langle \text{while } B \text{ St} \text{ else skip}, \sigma, \tau \rangle\]
\[\langle \text{if } \text{true then } \text{St}_1 \text{ else } \text{St}_2, \sigma, \tau, \text{td} \rangle \rightarrow \langle \text{St}_1, \sigma, \tau \rangle\]

\[\langle \text{if } \text{false then } \text{St}_1 \text{ else } \text{St}_2, \sigma, \tau, \text{td} \rangle \rightarrow \langle \text{St}_2, \sigma, \tau \rangle\]

\[\langle \text{while } B \text{ St}, \sigma, \tau, \text{td} \rangle \rightarrow \langle \text{while } B \text{ St} \text{ else skip}, \sigma, \tau \rangle\]
\[\langle P, \sigma, \tau, \text{td} \rangle \rightarrow \langle P', \sigma', \tau' \rangle\]
\[\langle X := I, \sigma, \tau, \text{td} \rangle \rightarrow \langle \text{skip}, \sigma([I,V]/X), \tau' \rangle\]
\[\langle \text{thread} = \text{td}, \text{type} = \text{read}, \text{target} = X, \text{state} = \sigma(X) \rangle\]
\[\langle \text{thread} = \text{td}, \text{type} = \text{write}, \text{target} = X, \text{state} = (I,V) \rangle\]

Table 1. ACIMP: Dynamic and Abstract Trace Semantics of CIMP.
Given a program \( p \) and a statement \( s \), let \( p[s/i] \) denote the replacement of the \( i \)th thread by \( s \). Formally, \( Stmt \parallel Pgm[Stmt/1] = Stmt \parallel Pgm \) and \( Stmt \parallel Pgm[Stmt/i + 1] = Stmt \parallel (Pgm[Stmt/i]) \), for \( i > 0 \).

Table 1 presents ACIMP, the SOS-style dynamic and abstract trace semantics of the CIMP language. The following types of configuration are used: \( \langle \text{Syn, State, Trace} \rangle \) and \( \langle \text{Syn, State, Trace, Int} \rangle \), where \( \text{Syn} \) ranges over \( \text{AExp}, \text{BExp}, \text{Stmt}, \) and \( \text{Pgm} \). The final parameter of the second type of configuration is used to maintain the id of the current thread being executed. We use \( X, L, A, B, St, P, \sigma, e, \) and \( \tau \) as meta-variables representing the \( \text{Var, Lock, AExp, BExp, Stmt, Pgm, State, Event, and Trace} \), respectively. All names (variables and locks) are shared among all threads.

To emphasize the language-based characteristics of an execution, which are invariable to the concrete semantics of a program, we use abstract states in the definition, that is, we enrich each store location to contain an additional abstract value together with its concrete integer value given by the semantics of the program. An abstract state of a program is mapping variables appearing in the program to integer-abstract value pairs and locks to integer values (representing the index of the thread holding them). We assume an arbitrarily large, yet numerable, set of abstract values, and a way to generate fresh values by request. These abstract values are included for theoretic purposes only; we will show that executions with abstract states are in a direct correspondence with executions with a normal state. Given a state \( \sigma \) and a variable \( X \), one can inspect the current integer value held by \( \sigma \) for \( X \) by using \( \sigma_X \); to retrieve the entire pair, one would simply use \( \sigma(X) \). An initial state for a program is a special state of the program, named \( \sigma_0 \), (eventually indexed by a number) and is totally defined on the set of names appearing in the program, assuming for each variable \( X \) the initial value \((0, V_X)\), where \( V_X \) is a fresh abstract value, and for each lock name \( X \), the initial value 0. The substitution of the value of \( X \) by an integer-abstract value pair \((I, V)\) is denoted as usual by \( \sigma[(I, V)/X] \). Traces are sequences of events as described by Definition 1. For this particular language, events will only contain attributes \( \text{thread, type, target, and state} \).

Let us next discuss the semantics definition presented in Table 1. We do that in three stages: first the dynamic semantics, then the trace semantics and finally the operations on the abstract state.

3.1 The dynamic semantics

The semantics of thread execution is based on interleaving and is specified by the rules (0 – 2) from the definition. This rules specify that, for each small step of the system, exactly one thread will be chosen and advanced one step. Threads are numbered based on their occurrence in the parallel-construct list, from left to right, starting with 1. tId is used to keep track of which thread is currently scheduled for execution.

The semantics for synchronization is given through rules (5 – 7) from the definition. The value of a lock can be either 0, meaning the lock has not been
taken, or the id of the thread holding it. Rule (5) says a thread can acquire a
lock if it is about to execute a synchronized block protected by that lock and the
lock is not being held by any other thread. If the current thread holds the lock
to an empty block, then the block can be dissolved, and the lock released – rule
(6). A synchronized block can be advanced one step only if the current thread
holds its lock, as specified by rule (7). From the point of view of the dynamic
semantics, the other rules are common for any imperative language, so we won’t
discuss them.

3.2 Trace Semantics

To generate traces, we will log all accesses to memory. For variables, these are
generated by the rules for reading/assigning them; for locks we will log lock ac-
quires and releases. Each time a memory access is performed, a new event is
 appended to the existing trace. For this language, we log the following informa-
tion: id of the thread, type of event, location accessed, and the contents of the
state/locks for that location. For write events we will record the state after the
execution of the step; for read events, the one before the execution.

3.3 Abstract State

As mentioned before, we want to analyze the execution from the point of view
of the programming language, rather than from that of the program since it is
well known [1] that this is generally undecidable. From the language point of
view, without additional information about the program being run, one cannot
derive from observing the state how the values were computed. However, one can
certainly track reads and writes of memory locations and know when a read event
reads the value written by a write event. To achieve this, we use the abstract
state and each time a location is written, a fresh abstract value is recorded in
the state, and also in the state attribute of the state, together with the concrete
value written. When a location is read, the state of the read location, including
the abstract value, is again recorded into the trace. Because we record a fresh
abstract value for each write, the only events which will have that abstract value
in the state are the write itself and the reads which read it.

4 Abstract Soundness

An initial configuration of an execution is a triple \langle p, \sigma_0, \tau_0 \rangle, such that \sigma_0 is an
initial abstract state and \tau_0 = e_1 e_2 \cdots e_k, where k is the number of variable
names appearing in p, and \(e_i = (\text{thread}(e_i) = 0, \text{type}(e_i) = \text{write}, \text{target}(e_i) =
X_i, \text{state}(i) = \sigma_0(X_i)), where X_i is the ith variable name in an arbitrary (but
fixed) ordering. We have used thread 0 for these initialization events to signal
that they are not generated by any thread in particular.
Definition 7. Given a program $p$ and an abstract trace $\tau$, we say that $p$ yields $\tau$, and write $p \xrightarrow{\cdot} \tau$, if there exists a derivation such that \( \text{ACIMP} \vdash \langle p, \sigma_0, \tau_0 \rangle \xrightarrow{\cdot} \langle p', \sigma, \tau \rangle \) for some program $p'$ and abstract state $\sigma$.

Let $\text{Traces}(p)$ denote the set of all traces yielded by $p$. We call these traces the feasible traces of $p$.

Since all rules of ACIMP only append at most one event to the existing trace without modifying it, it follows immediately that $\text{Traces}(p)$ is prefix closed, that is, $\tau_0 \tau \tau' \in \text{Traces}(p)$ implies that $\tau_0 \tau \in \text{Traces}(p)$. Moreover, since each time state is written, the value written and the location to which is written are recorded in the trace, it follows that there exists only one final state for any derivation yielding a trace.

Proposition 2. If $\text{ACIMP} \vdash \langle p, \sigma_0, \tau \rangle \xrightarrow{p, \sigma, \tau} \langle p_1, \sigma_1, \tau \rangle$ and $\text{ACIMP} \vdash \langle p, \sigma_0, \tau \rangle \xrightarrow{p, \sigma_2, \tau} \langle p_2, \sigma_2, \tau \rangle$, then $\sigma_1 = \sigma_2$. We use $\sigma_\tau$ to refer to that state.

Without using the abstract values, one could use prove, for example, that our language satisfies mutual exclusion, that is, that one cannot execute statements from two blocks synchronized by the same lock at the same time.

Proposition 3. ACIMP satisfies mutual exclusion. That is, for any program $p$ and trace $\tau \in \text{Traces}(p)$, $\tau$ satisfies the mutual exclusion property.

The following result holds, showing that abstract values indeed track state changes.

Lemma 1. Let $\emptyset \tau_0 \tau_1 \tau_2$ be an abstract trace yielded by $p$ and let $X$ be a variable name. Then $\sigma_{\emptyset \tau_0 \tau_1 \tau_2}(X) \neq \sigma_{\emptyset \tau_1 \tau_2}(X)$ iff there exists $e \in \mathcal{E}_{\tau_2}$ such that type($e$) = write and target($e$) = $X$. For lock names type($e$) can be either read or write.

The following result shows that abstract traces naturally capture the write-read dependence and atomicity.

Proposition 4. Let $\tau \in \text{Traces}(p)$. Then $\tau \vdash e_1 \sqsubseteq_{\text{wr}} e_2$ iff type($e_1$) = write, and state($e_1$) = state($e_2$).

A maximally executed program for a trace $\tau \in \text{Traces}_{\text{ACIMP}}(p)$, is a program $p'$ such that ACIMP $\vdash \langle p, \sigma_0, \epsilon \rangle \xrightarrow{\cdot} \langle p', \sigma_{\tau'}, \tau' \rangle$ and any possible step forward will produce a new event; that is, if ACIMP $\vdash \langle p', \sigma_{\tau'}, \tau' \rangle \xrightarrow{\cdot} \langle p'', \sigma_{\tau''}, \tau'' \rangle$, then $\tau'' = \tau' e$ for some $e$. Similarly to the way the trace determines the state, it turns out that it is also sufficient to express how the program evolved in order to produce that trace.

Proposition 5. Given a trace $\tau \in \text{Traces}_{\text{ACIMP}}(p)$, there exists a unique maximal executed program for that trace.

In the sequel, we let $p_\tau$ denote the maximal executed program for $\tau$. As it turns out, the uniqueness property holds even when restricted to a thread, as the next result shows.
**Lemma 2.** Let $p_{\tau_1}$ and $p_{\tau_2}$ be the maximally executed programs for $\tau_1$ and $\tau_2$, respectively. If $\pi_i(\tau_1) = \pi_i(\tau_2)$, then $\pi_i(p_{\tau_1}) = \pi_i(p_{\tau_2})$.

The result above allows us to reorder silent transitions of an execution to enforce its passing through the maximally executed program for each prefix of the trace.

**Lemma 3.** If $\langle p, \sigma_0, \epsilon \rangle \rightarrow^* \langle p_1, \sigma_{\tau_1}, \tau_1 \rangle \rightarrow^* \langle p_2, \sigma_{\tau_1\tau_2}, \tau_1\tau_2 \rangle$, then $\langle p_{\tau_1}, \sigma_{\tau_1}, \tau_1 \rangle \rightarrow^* \langle p_2, \sigma_{\tau_1\tau_2}, \tau_1\tau_2 \rangle$.

The main result of this section shows that causal atomicity is sound for the abstract state semantics.

**Theorem 1 (Abstract Soundness).** Let $\tau$ be a trace of $p$, and $\tau'$ be a causally atomic feasible trace of $\tau$. Then $\tau' \in \text{Traces}_{\text{ACIMP}}(p)$.

## 5 Concrete Soundness and maximality

### 5.1 Abstract-Concrete Correspondence

Let us prove that abstract values added to the concrete values in the state are indeed for auxiliary purposes only. Consider the definition in Table 1 in which states are now mappings from names to integer values only, the `state` attribute of an event only contains an integer, and all abstract values are erased. This basically only modifies rules for reading/writing a variable from the store (3,4) in the following way:

\[
\langle X, \sigma, \tau, \text{Id} \rangle \rightarrow \langle I, \sigma, \tau_e \rangle, \quad \text{if } \sigma(X) = I \quad (3')
\]

\[
\langle X := I, \sigma, \tau, \text{Id} \rangle \rightarrow \langle \text{skip}, \sigma[I/X], \tau_e \rangle, \quad \text{if } e = (\text{thread} = \text{Id}, \text{type} = \text{write}, \text{target} = X, \text{state} = I) \quad (4')
\]

We will name the definition obtained after these transformations the **concrete semantics of CIMP**, and denote it as $\text{CIMP}$. Consider also several forgetful mappings, all termed $\hat{\cdot}$ which transform abstract values states, events, traces, and configurations to their concrete counterparts by forgetting the abstract part of any concrete-abstract value pair. Formally, this can be defined as the free extension of the projection on the first component mapping defined on the set of concrete-abstract value pairs with values to the set of integer numbers.

Definition 7 naturally applies for the concrete semantics and concrete traces as well. When in danger of confusion we will index the relation by the name of the language definition to clearly state which definition we are using. The following shows that working with abstract state we do not add or loose semantic behaviours.

**Proposition 6.** $\hat{\text{Traces}}_{\text{ACIMP}}(p) = \text{Traces}_{\text{CIMP}}(p)$.
5.2 \(\alpha\)-equivalence of abstract traces

Although we generate a fresh abstract value each time we write a variable value in the store, we are not interested in the value itself, but just use the fact that it is fresh as a way to uniquely identify it in the subsequent read events of the same location. Our setting is in some sense similar with the one in functional languages, where one only needs a binding variable for its identity, to know where the values should replace it when the function is applied. Therefore, as in the theoretical treatment of functional languages, it makes a lot of sense to equate traces up to a renaming of abstract values, and work with equivalence classes of such traces.

**Definition 8.** Two abstract traces \(\tau_1\) and \(\tau_2\) are termed \(\alpha\)-equivalent, written \(\tau_1 \equiv \alpha \tau_2\), iff there exists an automorphism \(b\) on the set of abstract values such that its free extension to events and traces yields \(b(\tau_1) = \tau_2\).

Since \(b\) is a bijection, it quickly follows that \(\alpha\)-equivalence is an equivalence relation. The following shows that \(\text{Traces}(p)\) is closed under \(\alpha\)-equivalence:

**Proposition 7.** For all abstract traces such that \(\tau \equiv \alpha \tau'\),

\[
\tau \in \text{Traces}(p) \text{ iff } \tau' \in \text{Traces}(p).
\]

This enables us to factor the abstract traces of a program and work modulo \(\alpha\)-equivalence. Let \(\mathcal{T}\) denote the class of abstract traces \(\alpha\)-equivalent with \(\tau\), that is, \(\mathcal{T} = \tau /\equiv \alpha = \{\tau' \mid \tau' \equiv \alpha \tau\}\). Let \(\mathcal{T}(p)\) denote the factorization of \(\text{Traces}(p)\) by the alpha equivalence, that is, \(\mathcal{T}(p) = \text{Traces}(p)/\equiv \alpha = \{\mathcal{T} \mid p \Rightarrow \tau\}\). Let us now show that \(\alpha\)-equivalence of traces yields \(\alpha\)-equivalence of states, and thus of configurations, for the obvious definitions of those concepts.

**Lemma 4.** Let \(\text{Syn}_1, \text{Syn}_2\) be syntax elements generated by the syntax of the language. If \(\text{ACIMP} \vdash \langle \text{Syn}_1, \sigma_1, \tau_1 \rangle \rightarrow \langle \text{Syn}'_1, \sigma'_1, \tau'_1 \rangle\) and \(\text{ACIMP} \vdash \langle \text{Syn}_2, \sigma_2, \tau_2 \rangle \rightarrow \langle \text{Syn}'_2, \sigma'_2, \tau'_2 \rangle\) such that \(b(\sigma_1) = \sigma_2\) and \(b(\tau'_1) = \tau'_2\), then \(b(\sigma'_1) = \sigma'_2\).

We can use the Lemma 4 to show that there is a one-to-one correspondence between \(\mathcal{T}(p)\) and \(\text{Traces}_{\text{CIMP}}(p)\).

**Theorem 2.** Let \(\tau_1, \tau_2 \in \text{Traces}_{\text{ACIMP}}(p)\). If \(\hat{\tau}_1 = \hat{\tau}_2\) then \(\tau_1 \equiv \alpha \tau_2\). Therefore, \(\mathcal{T}(p)\) and \(\text{Traces}_{\text{CIMP}}(p)\) are isomorphic.

The identification of \(\mathcal{T}(p)\) and \(\text{Traces}_{\text{CIMP}}(p)\) allows us to give a unified definition of a trace, and of a feasible trace.

**Definition 9.** A *feasible trace* of a program \(p\) is a class of \(\alpha\)-equivalent abstract traces yielded by \(p\). The alphabet of a feasible trace \(\mathcal{T}\), written \(\mathcal{E}_\mathcal{T}\) is the set \(\{\mathcal{E}_{\tau'} \mid \tau' \in \mathcal{T}\}\).

Therefore \(\mathcal{T}(p)\) becomes the set of feasible traces of \(p\). Regarding the alphabet, one should notice that, while \(\mathcal{E}_\mathcal{T} = \mathcal{E}_{\hat{\tau}}\) implies that \(\mathcal{E}_{\hat{\tau}} = \mathcal{E}_{\hat{\tau'}}\), the converse is not generally true.
5.3 Maximality Result

Let us define the language-based causal equivalence on the feasible traces of a program. We say that two feasible traces of the program are causally equivalent based on the language if one of them can be obtained as an interleaving of the other. Formally,

**Definition 10.** Let $\tau_1, \tau_2 \in \mathcal{T}_p$. $\tau_1$ is $\mathcal{L}$-causal equivalent to $\tau_2$, written $\tau_1 \equiv^{\mathcal{L}}_p \tau_2$, iff $\tau_1 \in \text{interleavings}(\tau_2)$.

The following could be considered as an alternative definition of language-based causal equivalence.

**Proposition 8.** $\tau_1 \equiv^{\mathcal{L}}_p \tau_2$ iff $E_{\tau_1} = E_{\tau_2}$ and $\leq^{\tau_1} = \leq^{\tau_2}$. Therefore $\equiv^{\mathcal{L}}_p$ is an equivalence relation.

The result above shows that the feasible traces of a program can be partitioned into equivalence classes w.r.t. the language-based causality.

**Definition 11.** The $\mathcal{L}$-based causal equivalence class of a feasible trace $\tau$, written $[\tau]^{\mathcal{L}}_p$, is the set $[\tau]^{\mathcal{L}}_p = \{ \tau' \mid \tau' \in \text{interleavings}(\tau) \}$.

The causal atomic class of a concrete trace $\hat{\tau}_0 \tau \in \text{Traces}_{\text{ACIMP}}(p)$, written $[\hat{\tau}_0 \tau]_{\text{atomic}}$, is the set of causally atomic feasible traces $\hat{\tau}_0 \tau'$ of $\hat{\tau}_0 \tau$.

The main result of this paper, presented below, shows that the language-based causal equivalent class of a feasible trace $\tau$ is effectively captured as the set of causally atomic feasible traces of $\tau$. This not only shows that the causal atomicity is a sound technique, but also shows that it maximally identifies with the language-based causality which we have argued to be the most relaxed causality one could get without knowledge about the semantics of the program itself.

**Theorem 3 (Concrete Soundness and maximality).** For any trace $\tau \in \text{Traces}_{\text{ACIMP}}(p)$, $[\tau]^{\mathcal{L}}_p = [\hat{\tau}]_{\text{atomic}}$.

6 Preliminary Evaluation

7 Conclusion

References


This appendix contains proof to the theorems presented in the paper and is included for reviewers’ convenience. In case of acceptance, it will be removed and a reference to a technical report containing the proofs will be provided.

A Proofs of the main results

Theorem 1 (Abstract Soundness). Let $\tau$ be a trace of $p$, and $\tau'$ be a causally atomic feasible trace of $\tau$. Then $\tau' \in Traces_{\text{ACIMP}}(p)$.

Proof. We will prove that for any prefix $\tau'_1 = \langle 0 \rangle \tau''_1$ of $\tau'$, $\tau'_1 \in Traces_{\text{ACIMP}}(p)$, by induction on the length of $\tau''_1$. The base case trivially holds. Assume now that $\tau'_1 \in Traces_{\text{ACIMP}}(p)$ and let $e$, $\tau'_2$ be such $\tau' = \tau'_1 \tau'_2$. Let $\tau_1$, $\tau_2$ be such $\tau = \tau_1 \tau_2$. Since $\tau'$ is consistent with the thread ordering of $\tau$, it means that the thread ordering of $\tau'$ is the same as the one of $\tau$; therefore it must be that $\pi_{\text{thread}}(e)(\tau_1) = \pi_{\text{thread}}(e)(\tau'_1)$. By Lemma 2 it follows that $\pi_{\text{thread}}(e)(p_{\tau_1}) = \pi_{\text{thread}}(e)(p_{\tau'_1})$. Let ACIMP $\vdash \langle p_{\tau_1}, \sigma_{\tau_1}, \tau_1 \rangle \rightarrow \langle p_{\tau_1}, \sigma_{\tau_1}, \tau_1 e \rangle$ be the step producing $e$ in the derivation of $\tau$. Let us discuss the proof tree for this transition. It would have $\pi_{\text{thread}}(e) - 1$ applications of rule (2), followed by one of rule (1). Therefore $\pi_i(p_1) = \pi_i(p_{\tau_1}), if i \neq \text{thread}(e)$. After that, the instance of the top goal will look like: $\langle \pi_{\text{thread}}(e)(p_{\tau_1}), \sigma_{\tau_1}, \{\langle \text{thread}(e), \perp \rangle / tId \}, \tau_1 \rangle \rightarrow \langle \pi_{\text{thread}}(e)(p_1), \sigma(\tau_1 e), \tau_1 e \rangle$. On top of that will be 0 or many instances of the unnumbered, non-axiom rules, or of rule (7) until finally, one of the axioms (3 – 6) would complete the proof.

Let $\sigma_1 = \sigma_{\tau_1}, \{\langle \text{thread}(e), \perp \rangle / tId \}$. We will try to rebuild the proof starting with ACIMP $\vdash \langle p_{\tau_1}, \sigma_{\tau_1}, \tau_1 \rangle \rightarrow \langle p'_1, \sigma_{\tau'_1}, \tau'_1 \rangle$ as the root, where $p'_1 = p_{\tau'_1} / \pi_{\text{thread}}(e)(p_1)$. We can apply rule (2) for $\pi_{\text{thread}}(e) - 1$ times, then since $\pi_{\text{thread}}(e)(p_{\tau_1}) = \pi_{\text{thread}}(e)(p_{\tau'_1})$ we can apply rule (1) to get to an instance of the top goal like: $\langle \pi_{\text{thread}}(e)(p_{\tau_1}), \sigma'_1, \tau'_1 \rangle \rightarrow \langle \pi_{\text{thread}}(e)(p_1), \sigma(\tau'_1 e), \tau'_1 e \rangle$, where $\sigma'_1 = \sigma_{\tau'_1}[\langle \text{thread}(e), \perp \rangle / tId]$. Since all unnumbered rules only depend on the syntax, we can apply them as for the original proof. Suppose, though, that an instance of rule (7) must be applied, which, besides decomposing the program, also requires that $(\sigma'_1)_X = tId = \text{thread}(e)$. Since the instance of the original proof held, it must be that $(\sigma_1)_X = \text{thread}(e)$. Now, this is only possible if the latest event $e'$ in $\tau_1$ such that $\text{target}(e') = X$ has $\text{type}(e') = \text{write}$ and $\text{thread}(e') = \text{thread}(e)$, and $\text{state}(e') = \sigma_1 (X)$. Since $\pi_{\text{thread}}(e)(\tau'_1) = \pi_{\text{thread}}(e)(\tau_1)$, it follows that $e' \in E_{\tau'_1}$. Let us show that $e'$ is also the latest in $\tau'_1$ such that $\text{target}(e') = X$. Since $\tau'_1$ is an execution, if there exists another events greater than $e'$ in $\tau'_1$ with target $X$, the corresponding release of $e'$ must have occurred before that. Therefore, there exists $e'' \in \mathcal{E}_{\tau'_1}, \tau'_1 \vdash e' \sqsubseteq_{\text{wr}} e'', \text{such that } \text{target}(e'') = X, \text{type}(e'') = \text{read}, \text{thread}(e'') = \text{thread}(e), \text{and state}(e'') = \sigma_1 (X)$. But this would lead to contradiction, because it follows that $e'' \in \pi_{\text{thread}}(e)(\tau'_1) = \pi_{\text{thread}}(e)(\tau_1)$, and $\tau'_1 \vdash e' \sqsubseteq_{\text{wr}} e''$ implies $\tau_1 \vdash e' \sqsubseteq_{\text{wr}} e''$, thus $e'$ is not the latest in $\tau_1$ with $\text{target}(e') = X$. Therefore $\sigma'_1 (X) = \text{state}(e')$ and we have an instance of rule (7).

Let us now show for each of the axioms (3 – 6) that if one of them made the original proof complete, it will also make the new proof complete.

Rule (3): $\text{type}(e) = \text{read}, \text{target}(e) = X$, and $\text{state}(e) = \sigma_1 (X)$. Then there exists $e' \in \mathcal{E}_{\tau_1}$ such that $\tau_1 \vdash e' \sqsubseteq_{\text{wr}} e$, and $\text{state}(e) = \sigma_1 (X)$. Since $\tau'_1$ is
consistent with the atomic causality, it means that \( \tau' \vdash e' \sqsubseteq_{\text{wr}} X \), whence \( e' \) is the latest event in \( \tau'_1 \) such that \( \text{type}(e') = \text{write} \) and \( \text{target}(e') = X \). Therefore, \( \sigma'_1(X) = \text{state}(e') \), whence the rule can also be applied for the new proof to complete it.

**Rule (4):** \( \text{type}(e) = \text{write} \), \( \text{target}(e) = X \), and \( \text{state}(e) = \sigma_1(X) = (I, V) \) with \( V \) fresh value. Then, since the value written is part of the statement, and \( V \) is also fresh for \( \tau'_1 \) (since \( \tau'_1 \) is consistent with the causal atomicity), it follows that the rule can also be applied to complete the new proof.

**Rule (5):** \( (\sigma_1)_X = 0, V \) fresh, \( \text{thread}(e) = \text{Id} \), \( \text{type}(e) = \text{write} \), \( \text{target}(e) = X \), and \( \text{state}(e) = (\text{Id}, V) \). Suppose there exists \( e <_{\tau'_1} e' \) such that \( \text{target}(e') = X \) and \( \text{type}(e) = \text{write} \). Since \( \tau \) is complete, there must be \( e'' \) such that \( \tau \vdash e' \sqsubseteq_{\text{wr}} e'' \). Therefore, since \( \tau' \) is consistent with the atomic causality, \( \tau' \vdash e' \sqsubseteq_{\text{wr}} e'' \). But this precisely means that \( e'' <_{\tau'} e' \). It follows that \( (\sigma'_1)(X) \) is either \( \phi_0(X) \) of the value left by the last release, in both cases being \( (0, \bot) \). This means that axiom (5) also applied to complete the new proof.

**Rule (6):** Same reasoning as for Rule (3).

**Theorem 2.** Let \( \tau_1, \tau_2 \in \text{Traces}_{\text{ACIMP}}(p) \). If \( \tilde{\tau}_1 = \tilde{\tau}_2 \) then \( \tau_1 \equiv_\alpha \tau_2 \). Therefore, \( \mathbb{T}(p) \) and \( \text{Traces}_{\text{CIMP}}(p) \) are isomorphic.

**Proof.** We will prove by induction on the length of the trace that if \( \text{ACIMP} \vdash \langle p, \phi_1, \phi_0 \rangle \vdash^* \langle p_1, \sigma_1, \tau_1 \rangle \) and \( \text{ACIMP} \vdash \langle p, \phi_0, \phi_0 \rangle \vdash^* \langle p_2, \sigma_2, \tau_2 \rangle \) such that \( \tilde{\tau}_1 = \tilde{\tau}_2 \) then there exists a renaming \( b \) of abstract values such that \( b(\tau_1) = \tau_2 \) and \( b(\sigma_1) = \sigma_2 \).

For the base case, \( \tau_1 = \phi_0 \) and \( \tau_2 = \phi_0 \), and, as we already noticed, it means no state-affecting step was performed. Therefore \( \sigma_1 = \phi_0 \) and \( \sigma_2 = \phi_0 \), and we can choose \( b \) such that for each variable name \( X \) such that \( \phi_0(X) = (0, V_{X,1}) \) and \( \phi_0(X) = (0, V_{X,2}) \), \( b(V_{X,1}) = V_{X,2} \), and arbitrary distinct values for the other abstract values. Since all abstract values in \( \phi_0 \) are distinct, \( b \) is well defined. Moreover, \( b(\phi_0) = \phi_0 \). Since all abstract values in \( \phi_0 \) are distinct, \( b \) is a bijection. For the induction case, making abstraction of transitions not generating events since they do also not modify the state we need to prove that, if \( \text{ACIMP} \vdash \langle p_1, \sigma_1, \tau_1 \rangle \vdash^* \langle p_1', \sigma'_1, \tau_1 e_1 \rangle \) and \( \text{ACIMP} \vdash \langle p_2, \sigma_2, \tau_2 \rangle \vdash^* \langle p_2', \sigma'_2, \tau_2 e_2 \rangle \) such that \( b(\sigma_1) = \sigma_2 \), \( b(\tau_1) = \tau_2 \), and \( e_1 = e_2 \), then there exists \( b' \) such that \( b'(\tau_1 e_1) = \tau_2 e_2 \) and \( b'(\sigma'_1) = \sigma'_2 \).

If \( \text{type}(e_1) = \text{read} \) then we already have \( b(e_1) = e_2 \) and we can use Lemma 4 to prove that also \( b(\sigma'_1) = b(\sigma'_2) \). If \( \text{type}(e_1) = \text{write} \), assume then that \( \text{state}(e_1) = (I, V_1) \) and \( \text{state}(e_2) = (I, V_2) \). Moreover, let \( V'_1 = b(V_1) \) and \( V'_2 = b(V_2) \) be such \( b(V'_2) = V_2 \). Then let \( b' \) be defined as \( b'(X) = \begin{cases} V_2, & \text{if } X = V_1 \\ V'_2, & \text{if } X = V'_1 \\ b(X), & \text{if } X \notin \{V_2, V'_1\} \end{cases} \)

Since \( V_1 \) is fresh for first derivation, it must be that \( V'_1 \) is fresh for the second derivation. That is because whenever fresh names for the first derivation are generated, write events are recorded in the trace; therefore if not fresh, \( V'_1 \) would have appeared in \( \tau_2 \) and since \( b(\tau_2) = \tau_2 \), it would mean that \( V_1 \) appears in
\[ \tau_1 \text{ so it would not be fresh. Similarly, } V'_2 \text{ is fresh because } V_2 \text{ is fresh. Therefore } b'(\tau_1 e_1) = b'(\tau_2 e_2) \text{ and } b'(\sigma_1) = b'(\sigma_2). \] We are now in the conditions of Lemma 4 and by applying it our proof is complete.

**Theorem 3 (Concrete Soundness and maximality).** For any trace \( \tau \in \text{Traces}_{\text{ACIMP}}(p) \), \( [\tau]_p^L = [\tilde{\tau}]_\text{atomic}. \)

**Proof.** Let \( \tau' \in [\tau]_p^L \). Let \( \tilde{\tau} \in [\tau]_p^L \) such that \( \tilde{\tau} = \tau' \). From Proposition 8 we have that \( E_{\tilde{\tau}} = E_\tau \) and \( <_t^{e_1} = <_t^{e_2} \). Therefore \( \tau' \) is an interleaving of \( \tilde{\tau} \). We need to show that it is consistent with the write-read dependence and the write-read atomicity. Let \( e_1, e_2 \in E_\tau \) such that \( \text{type}(e_1) = \text{write}, \text{type}(e_2) = \text{read}, \text{target}(e_1) = \text{target}(e_2) = X, \) and \( \tilde{\tau} \vdash e_1 \sqsubseteq_X e_2 \). Let \( e'_1 \) and \( e'_2 \) be the corresponding events in \( \tau' \), and let \( \tau_0 \in \tau \) be such \( E_{\tilde{\tau}} = E_{\tau_0}. \) Since \( \tilde{\tau}_0 = \tilde{\tau} \), it must be that \( \tau_0 \vdash e'_1 \sqsubseteq_X e'_2 \). From Proposition 4 this is equivalent with \( \text{state}(e'_1) = \text{state}(e'_2) \). Applying again Proposition 4, this time for \( \tau' \) this is equivalent with \( \tau' \vdash e'_1 \sqsubseteq_X e'_2 \). Since \( e_1 \) and \( e_2 \) were arbitrarily chosen, we have that \( \sqsubseteq_X^{\tau'} = \sqsubseteq_X^{\tilde{\tau}} \). But this precisely means that \( \tau' \) is consistent with the atomic causality.

Conversely, let \( \tilde{\tau} \) be a trace in \( [\tilde{\tau}]_\text{atomic} \). We need to prove that there exists a trace \( \tau' \in \text{Traces}_{\text{ACIMP}}(p) \) such that \( \tilde{\tau} = \tau' \) (therefore, by Proposition 6, \( \tilde{\tau} = \tau' \) is in \( \text{Traces}_{\text{CIMP}}(p) \)), and that \( E_{\tilde{\tau}} = E_\tau \) (since \( <_t^{e_1} = <_t^{e_2} \) to begin with). Suppose \( \tilde{\tau} = e_1 e_2 \cdots e_n \). Let \( \rho \) be a permutation of size \( n \) such that \( \tilde{\tau} \tau' = \tilde{\tau} \circ \rho \), that is, \( \tau' = e_\rho(1) e_\rho(2) \cdots e_\rho(n) \). This would obviously be the identity for the first \( k \) events, where \( k \) is the number of variable names of the program. Let \( \tau' = \tau \circ \rho \). Obviously, \( E_{\tau'} = E_\tau \). By Theorem 1, \( \tau' \in \text{Traces}_{\text{ACIMP}}(p) \), and our proof is complete, since this also implies that \( E_{\tilde{\tau}} = E_\tau \) (their intersection is not empty).