Technical report UIUCDCS-R-2008-3017, December 2008

Maximal Causal Models for Multithreaded Systems

Traian Florin Şerbănuţă, Feng Chen and Grigore Roşu

December 11, 2008

Abstract

Extracting causal models from observed executions has proved to be an effective approach to analyze concurrent programs. Most existing causal models are based on happens-before partial orders and/or Mazurkiewicz traces. Unfortunately, these models are inherently limited in the context of multithreaded systems, since multithreaded executions are mainly determined by consistency among shared memory accesses rather than by partial orders or event independence. This paper defines a novel theoretical foundation for multithreaded executions and a novel causal model, based on memory consistency constraints. The proposed model is sound and maximal: (1) all traces consistent with the causal model are feasible executions of the multithreaded program under analysis; and (2) assuming only the observed execution and no knowledge about the source code of the program, the proposed model captures more feasible executions than any other sound causal model. An algorithm to systematically generate all the feasible executions comprised by maximal causal models is also proposed, which can be used for testing or model checking of multithreaded system executions. Finally, a specialized submodel of the maximal one is presented, which gives an efficient and effective solution to on-the-fly datarace detection. This datarace-focused model, still captures more feasible executions than the existing happens-before-based approaches.

1 Introduction

Causal models of concurrent program executions allow to identify causally equivalent executions without rerunning the underlying program and have been widely used in program analysis, e.g., to detect concurrency errors or to reduce the complexity of verification. Many causal models for multithreaded systems have been proposed, most of them inspired from related work on distributed systems [17] or on trace theories [20]. These models are usually defined by means of a causal partial order on the observed events; however, multithreaded systems executions are mainly determined by consistency among shared memory accesses rather than by partial orders or independence among events, limiting the effectiveness of existing causal models.

Contributions. Our first contribution is a novel causal model for multithreaded systems, based on an axiomatization of sequentially consistent memory. The proposed model is *sound* and *maximal* (or *complete*), thus putting an end to the quest for universally "better" causal models[1]. Several case studies are investigated: (1) the usual happens-before causality is subsumed by our models; (2) our models can be applied directly on executable definitions of programming languages—we show that the semantics of a multithreaded language satisfies the axioms; (3) our maximal causal models also subsume (and thus leverage) rather complex causalities encountered in the literature, such as sliced causality[2]. The proposed causal model is practical.

[1] The problem of devising algorithmically "better" representations of causal models for particular properties of interest such as for dataraces or atomicity, will probably always be open.

[2] Sliced causality [4] filters irrelevant events using static (control, data, and aliasing) information about the monitored program.

Thread 1	***Thread 2***	***Thread 1***	***Thread 2***
e_1:$(1, \mathit{acquire}, l)$		acquire(l);	
e_2:$(1, \mathit{write}, x, 1)$		$x := 1$;	
e_3:$(1, \mathit{release}, l)$		release(l);	
e_4:$(1, \mathit{write}, y, 1)$		$y := 1$;	
e_5:$(1, \mathit{acquire}, l)$		acquire(l);	
e_6:$(1, \mathit{write}, x, 1)$		$x := 1$;	
e_7:$(1, \mathit{release}, l)$ $\|$		release(l); $\|$	
	e_8:$(2, \mathit{acquire}, l)$		acquire(l);
	e_9:$(2, \mathit{read}, x, 1)$		if ($x > 0$)
	e_{10}:$(2, \mathit{write}, y, 2)$		then $y := 2$;
(a)	e_{11}:$(2, \mathit{release}, l)$	(b)	release(l);

Figure 1: Example trace and a program generating it.

Our second contribution is an algorithmic characterization of the maximal model: given an observed trace, this algorithm systematically explores all its causally equivalent feasible traces, so it can be used for model-checking executions. Our third contribution regards the on-the-fly detection of dataraces. A specialized submodel of the maximal one is defined and shown to have more coverage than other existing on-the-fly sound techniques. These algorithms have been implemented and experimented with in the context of Java.

Motivating Example. Figure 1(a) shows a two-threaded trace, $e_1e_2e_3e_4e_5e_6e_7e_8e_9e_{10}e_{11}$. An event is a tuple of attributes: the id of the thread, the type of event, the subject (memory location or lock), and possibly a value (if event type is write or read); e_1 is an acquire of lock l by thread 1; e_2, also by thread 1, is a write of x with value 1. A general and rigorous definition of events is given in Section 2. Figure 1(b) shows a program that could generate the trace in Figure 1(a). This is obviously not the only program that can generate that; e.g., an alternative way to generate e_9 and e_{10} is to replace the conditional in Thread 2 by $y := x + 1$. This trace exhibits a datarace on y *in any program* whose execution might have produced it: two writes of y, namely e_4 and e_{10}, can occur next to each other in another execution of the program: if Thread 1 stops between e_3 and e_4, Thread 2 may continue to run until e_{10} is observed, since the read in e_9 will still get the value 1 (this time written by e_2), and then Thread 1 may awake to generate e_4.

However, no existing (sound) causal model can detect this race. Happens-before based models [3, 22, 28] enforce a dependency between the release of lock l and the subsequent acquire of l and thus conclude that e_8 always happens after e_7. Locksets approaches [27], may detect this bug but they are unsound (produce false alarms). More recent hybrid lockset-happens-before models [24], or relaxed happens-before models [30], allow one to permute synchronized blocks in different threads, e.g., $e_5e_6e_7$ and $e_8e_9e_{10}e_{11}$, but enforce at least the read-after-write dependency (a read should always follow the *latest write event* of the same variable). Hence, in Figure 1, these models claim that e_9 always happens after e_6, leading to a total order among events.

Our approach is based on *sequential consistency* of traces, formally defined in Section 2. Intuitively, a multithreaded trace is sequentially consistent if it satisfies the following: (1) every read of a variable has the same value as the latest write of the variable (but does not necessarily depend upon the latest write event!); and (2) no two threads hold the same lock at the same time in the trace. Given a multithreaded execution trace, any sequentially consistent interleaving of the trace's events is a *feasible* trace, i.e., one that can be generated by some execution of any program generating the given trace. This way, one can correctly infer other feasible executions of a multithreaded program from an observed execution trace without running the program again.

For example, $\tau = e_1e_2e_3e_8e_9e_4e_{10}e_{11}e_5e_6e_7$ is a sequentially consistent interleaving of the trace in Figure 1 (a), so τ is a feasible trace of the observed program; moreover, τ exhibits the race between e_4 and e_{10}. On the other hand, the consistency requirements prevent a trace obtained by replacing e_2 by $e'_2 = (1, \mathit{write}, x, 0)$ from being permuted in any order which would exhibit e_4 next to e_{10} (since the read of x at e_9 might guard the write of x at e_{10}, like in Figure 1(b)); therefore, no false alarm is reported. In fact, our model checking algorithm (Section 3) run on the trace in Figure 1(a) can infer 6 other feasible *proper* traces (none is a prefix of another), as opposed to none if using a happens-before model.

Since the model we associate to a trace comprises all traces which can be inferred from that trace using the memory model axioms, we can rightfully claim it is *the maximal model*. Interestingly, maximal-causal-model equivalent executions may not lead to the same state when they terminate, as enforced by happens-before models: it may be the case that one execution ends with y holding value 2 (e.g., the trace in Figure 1(a)), while an equivalent one, with y holding 1 (e.g., trace $e_1e_2e_3e_8e_9e_{10}e_4e_{11}e_5e_6e_7$). Our maximal model also comprises feasible traces with different events than in the observed one, e.g., trace $(2, \mathit{acquire}, l)$ $(2, \mathit{read}, x, 0)$. This trace is maximal w.r.t the original trace, in the sense that it cannot be continued, because: (1) the value read for x is different than the one in the original execution, which is perfectly fine as long as we know the last value held by x, but, after this event, the second thread cannot soundly be continued as in the original execution; and (2) the lock l being acquired by the second thread, the first thread is also prevented from advancing, since its first operation would require acquiring l. The reason for which one cannot continue a thread once a read having a different value than in the original execution is inferred is because the value read could be used in evaluating the condition of a control statement (as it indeed happens in Figure 1), in which case, a different value read can lead to a different path chosen to continue that thread.

Comparison with Past Work. There has been a considerable amount of research on models and techniques to abstract executions for the purpose of inferring causally equivalent executions satisfying/violating particular but important properties, such as dataraces or atomicity/serializability [2, 3, 6, 9, 13, 14, 16, 24, 26–30, 32]. All sound approaches are based on some causal model, usually defined by means of a happens-before causal partial order among the events observed during an execution. Our approach is closest in spirit to [22], which proposes an axiomatization of a happens-before causal order between memory accesses and semaphore operations, and uses that to detect anomalies which could have occurred under equivalent (by the axioms) permutations of the observed execution. An important aim of their work, and ours, is *soundness*, that is, lack of false alarms. As discussed above, we choose to directly axiomatize the memory model and show that our causal model is more general than the happens-before (and any other) causality.

Our theory of multithreaded traces serves as an alternative to Mazurkiewicz traces [7, 20] in the context of multithreaded systems, where a blind instance of it would not capture the intended notion of causal equivalence of traces. Indeed, Mazurkiewicz traces build upon a basic binary relation of independence on events: two traces are causally equivalent iff they differ by permuting consecutive independent events. Obvious dependent events in the context of multithreaded systems are, for example, two accesses to the same shared variable when at least one of the accesses is a write, or two acquire / release events of the same lock. While instantiating the Mazurkiewicz trace theory with the above dependence relation leads to a sound notion of causal equivalence (yielding precisely the equivalence classes of the happens-before causal model), that causal equivalence might be *too strong*, that is, would lead to equivalence classes that are smaller than could/should be. Indeed, as shown in the sequel, one can safely allow atomic permutations of non-contiguous cross-thread blocks of events starting with a write of a shared variable and containing precisely all the reads of the written location, and in some cases also permutations of lock-protected blocks (assuming no write/read atomic block is causally broken). The notion of two-event-independence underlying the theory of Mazurkiewicz traces is too coarse-grained to capture the *maximal causal equivalence class* in the context of multithreaded systems.

The approaches presented above infer equivalent executions by just analyzing the trace, without knowledge about the program generating it. Another interesting and productive line of research attempts to use information about the actual program code to either statically detect potential bad behaviors [11, 21], or to use information about the program and about the property to be checked to further relax the models of an execution [4, 10, 23]. Purely static analysis approaches have to overcome unavoidable undecidability aspects, and typically give up soundness to increase coverage. Dynamic techniques may use static knowledge to enhance their analysis capabilities, so, having underlying models as relaxed as possible is also crucial for them. Our paper is complementary to these approaches: our goal is to establish a rigorous foundation on

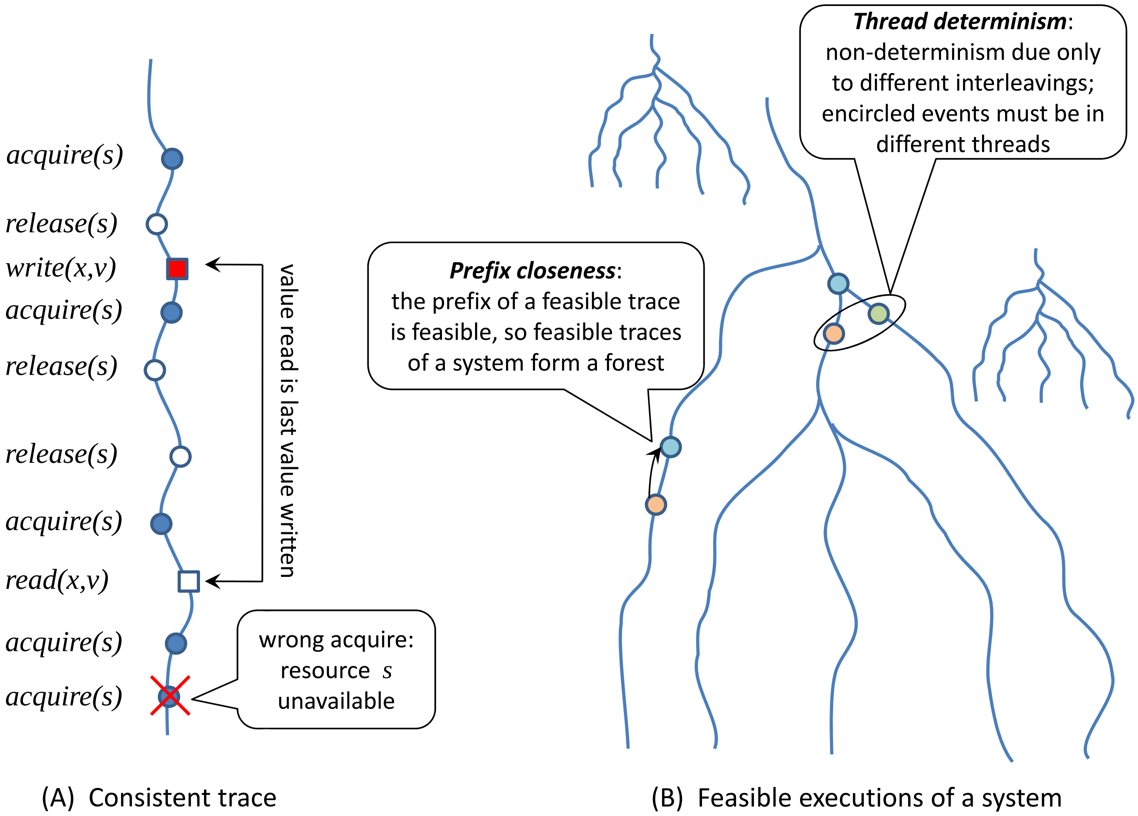

acquire(s)
release(s)
write(x,v)
acquire(s)
release(s)
release(s)
acquire(s)
read(x,v)
acquire(s)
acquire(s)
value read is last value written
wrong acquire: resource s unavailable
(A) Consistent trace
Thread determinism: non-determinism due only to different interleavings; encircled events must be in different threads
Prefix closeness: the prefix of a feasible trace is feasible, so feasible traces of a system form a forest
(B) Feasible executions of a system

Figure 2: Consistent traces and feasible executions

which dynamic code-based techniques can be developed. Section 2.4 details how one could make use of static information to increase the coverage.

Paper Structure. Section 2 defines the maximal causal model for sequentially consistent multithreaded systems and illustrates its generality through several case studies. Section 3 presents a constructive characterization of the maximal model and a model-checking algorithm. Section 4 presents an on-the-fly datarace detection algorithm based on property-specialized submodels of the maximal model. Section 5 presents evaluation results, and Section 6 concludes.

2 A Theory of Multithreaded Traces

We here propose a theory of multithreaded traces, called *MT-traces* for short, which will be used as a framework to formulate and prove the results in this paper. Figure 2 highlights the two major concepts underlying our theory, namely *consistent traces* and *feasible executions*. Consistent traces are those which can be produced by *sequentially consistent* [18] multithreaded systems in general. A consistent trace disallows "wrong" behaviors, such as reading a value different from the one which was written, or proceeding when a lock cannot be acquired. Our second notion, that of feasible executions, refers to *sets* of execution traces and aims at capturing *all* the behaviors that a given multithreaded system or program can manifest. No matter what task a multithreaded system or program accomplishes, its possible traces must obey some basic properties. First, feasible traces are generate-able, meaning that any prefix of any feasible trace is also feasible; this is captured by our first axiom of feasible traces, *prefix closeness*. Second, we here assume that thread interleaving is the only source of non-determinism in producing traces; this is captured by our second axiom of feasible traces, *thread determinism*. This axiomatic approach gives us an elegant and general definition for the *maximal* set of feasible executions which can be inferred from, or are causally equivalent to, an observed one.

Execution Environment. We assume a machine that can execute arbitrarily many threads in parallel. Threads are distinct and can be identified at any time during the execution. The execution environment has shared memory, which can be accessed by the threads to share data and synchronize.

Threads. A thread is a *sequence* of operations. Although in general there could be potential nondeterminism internal to a thread, in this paper we assume that the only source of non-determinism for a thread is its execution environment. That is, if the interaction between a thread and the environment is the same across executions, the thread will execute the same operations. Moreover, there is no direct communication between threads; all interaction between threads happens through the shared environment. Hence, a thread can run independently of the other threads. To simplify the presentation, we assume no dynamic creation of threads.

Semaphores. Synchronization is facilitated by the use of semaphores [8]. Semaphores can be viewed as managers of a number of resources, allowing threads to acquire a resource or release/provide a new resource. Semaphores can only be controlled by means of two indivisible operations: the "P-operation" (here called acquire), and the "V-operation" (here called release). release increases the number of available resources by one, while acquire decreases it by one, blocking if no resource is available and waiting for one to be released. This is not to be confused with re-entrant synchronized monitors, where a counter is maintained to keep track of how many times a mutex was acquired by the same thread. We choose semaphores over other synchronizations constructs since (1) they subsume most of the others (see e.g., [31]); and (2) they are most difficult to handle, due to the varying number of resources. The results in this paper easily adapt to other synchronization constructs[3].

Initial State. For simplicity, we assume that shared variables are initially set to 0 and semaphore counters to 1.

Events. Thread accesses to shared variables and semaphores are recorded as *events*. We consider events to be abstract entities from an infinite "collection" *Events*, and describe them as tuples of *attribute=value* pairs. The only attributes considered here are: *thread*—the unique id of the thread generating the event, *type*—the type of the event (*write*, *read*, *acquire*, or *release*), *target*—the memory location accessed by the event, and *state*—the value read/written by the current event (defined only for the *write*/*read* types of events). For example, $e_1 : (thread = t_1,\ type = write,\ target = x,\ state = 1)$ describes event e_1 as a write of location x with value 1, produced by thread t_1. For any event e and attribute *attr*, we will use $attr(e)$ to denote the projection of e to the attribute *attr*. We only list the attribute values in an event when there is no source of confusion; for example, the above event can be written as $(t_1, write, x, 1)$. Event identity is not reduced to identity of attributes: two events with identical attribute values may be distinct. We purposely considered only variable write/read and semaphore acquire/release events, as these are sufficient both to show the subtleties of the addressed problem and to capture the essence of our general solution.

2.1 Sequential Consistency

For simplicity, in this paper we assume sequential consistency [18]: the result of any execution is the same as if the operations of threads were executed in some sequential order, and the operations of each individual thread appear in this sequence in the order specified by its code.

[3] We have initially proved the subsequent results for synchronized monitors, but then we generalized them for semaphores.

Traces. Sequential consistency allows the observer of an execution to see a total order on all its events. Formally, 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 *events* of τ and let $<_\tau$ be the total order induced by τ on $\mathcal{E}_\tau$. Obviously, not all traces can be obtained as executions (e.g., a one-threaded trace writing 1 to x and then reading 0 from x is inconsistent). We give below two important consistency axioms which capture the essence of sequential consistency.

Write-Read Consistency: *when reading a memory location, the value read is the value written by the most recent write operation on that location.* Given a location x, an event type t, and a trace τ, let $\mathit{latest}^t_\tau(x)$ denote *the most recent event* of type t for x in τ, if such event exists. Formally, $\mathit{latest}^t_\tau(x)$ is an event e such that $\tau = \tau_1 e \tau_2$, $\mathit{target}(e) = x$, $\mathit{type}(e) = t$, and for all events $e' \in \mathcal{E}_{\tau_2}$, either $\mathit{target}(e') \neq x$ or $\mathit{type}(e') \neq t$. Since variables are all initialized with 0, we let $\mathit{state}(\mathit{latest}^t_\tau(x))$ be 0, by convention, if $\mathit{latest}^t_\tau(x)$ is undefined. A trace τ satisfies the *write-read consistency* requirement if for any prefix $\tau' e'$ of τ with $\mathit{type}(e') = \mathit{read}$, it must be that $\mathit{state}(e') = \mathit{state}(\mathit{latest}^{\mathit{write}}_{\tau'}(\mathit{target}(e')))$.

Semaphore Consistency [15]: *the number of resources available for any semaphore location s and any prefix τ of a trace is non negative.* We write that number as $\mathit{available}_\tau(s)$, and formally define it as $\mathit{available}_\tau(s) = 1 + |\{e \in \mathcal{E}_\tau \mid \mathit{type}(e) = \mathit{release} \wedge \mathit{target}(e) = s\}| - |\{e \in \mathcal{E}_\tau \mid \mathit{type}(e) = \mathit{acquire} \wedge \mathit{target}(e) = s\}|$. A trace τ satisfies the *semaphore consistency* requirement if for any prefix trace τ' of τ and any semaphore s, $\mathit{available}_{\tau'}(s) \geq 0$.

Definition 1 *A trace satisfying the write-read and semaphore consistency axioms is **(sequentially) consistent**.*

2.2 Feasibility

Each particular multithreaded system or programming environment has its own notion of feasible execution, given by its specific intended semantics. We prefer to *axiomatize* the set of feasible executions by their crucial properties, to make our results general and applicable to the various special cases. Let us consider a hypothetical multithreaded system $\mathcal{S}$ and an initial state of the memory. All we need from $\mathcal{S}$ is the fact that it can produce a certain (potentially infinite) set of (finite) traces. We deliberately avoid giving a definition of a multithreaded system; this approach pays off in Section 2.4, where an abstract, otherwise nonexistent system is associated to a concrete multithreaded system. $\mathcal{S}$ can be executed in various ways, depending on the scheduling policy of the machine, so many different possible traces can be observed during its executions. A possibly incomplete (i.e., non-terminated, i.e., a prefix of a potentially infinite) trace observed during the execution of $\mathcal{S}$ is called a *feasible trace of* $\mathcal{S}$. Let $\mathit{feasible}(\mathcal{S})$ denote all feasible traces of $\mathcal{S}$. Assuming sequential consistency and that threads operations are deterministic and execute independently of other threads (except for synchronization), $\mathit{feasible}(\mathcal{S})$ must satisfy the following:

Prefix Closeness: events are indivisible and generated in execution order; therefore, $\mathit{feasible}(\mathcal{S})$ is prefix closed. That is, if $\tau_1\tau_2 \in \mathit{feasible}(\mathcal{S})$, then $\tau_1 \in \mathit{feasible}(\mathcal{S})$.

Thread Determinism. *an event is completely determined (except for its state when it is a read event) by the previous events in the same thread, and can be generated at any consistent moment after them.* Let us formalize this. The *projection* of trace τ to thread n, written $\pi_n(\tau)$, is the subsequence of events in τ whose thread is n. Using this definition, thread determinism says that if $\tau e, \tau' \in \mathit{feasible}(\mathcal{S})$ and $\pi_{\mathit{thread}(e)}(\tau) = \pi_{\mathit{thread}(e)}(\tau')$ then: if $\tau' e$ consistent then $\tau' e \in \mathit{feasible}(\mathcal{S})$; and if $\mathit{type}(e) = \mathit{read}$ then there exists e' having the same attributes as e, except $\mathit{state}(e') = \mathit{state}(\mathit{latest}^{\mathit{write}}_\tau(\mathit{target}(e)))$, such that $\tau' e' \in \mathit{feasible}(\mathcal{S})$.

Definition 2 *If $\mathit{feasible}(\mathcal{S})$ is a prefix closed set of consistent traces satisfying the thread determinism, then $\mathcal{S}$ is called a **(sequentially) consistent** multithreaded system.*

Thread 1	***Thread 2***	***Observed Trace***
$q1 := 1;$		$e_1 : (1, \mathit{write}, q1, 1)$
$\mathit{turn} := 1;$		$e_2 : (1, \mathit{write}, \mathit{turn}, 1)$
while $(q2 = 1$ and $\mathit{turn} = 1)$ do skip;		$e_3 : (1, \mathit{read}, q2, 0)$
		$e_4 : (1, \mathit{read}, \mathit{turn}, 1)$
$\mathit{critical} := 1;$		$e_5 : (1, \mathit{write}, \mathit{critical}, 1)$
$q1 := 0;$		$e_6 : (1, \mathit{write}, q1, 0)$
	$q2 := 1;$	$e_7 : (2, \mathit{write}, q2, 1)$
	$\mathit{turn} := 2;$	$e_8 : (2, \mathit{write}, \mathit{turn}, 2)$
	while $(q1 = 1$ and $\mathit{turn} = 2)$ do skip;	$e_9 : (2, \mathit{read}, q1, 0)$
		$e_{10} : (2, \mathit{read}, \mathit{turn}, 2)$
	$\mathit{critical} := 2;$	$e_{11} : (2, \mathit{write}, \mathit{critical}, 2)$
	$q2 := 0;$	$e_{12} : (2, \mathit{write}, q2, 0)$

Figure 3: Peterson's algorithm and a trace generated by it.

Feasibility Closure. A major goal of many of the trace-based analyses discussed in Section 1 is to infer and analyze as many traces as possible using as reference the recorded trace. When one does not know, or does not want to use, the source code of the multithreaded program being executed, one can only infer potential traces of the system resembling the observed trace, in particular no alteration of the state can be inferred if it was not already present in the observed trace; however, one could possibly be allowed to read a different value of the state, in a thread as long as that thread is not further continued. As we previously mentioned, the reason one cannot allow a thread to be advanced as in the original execution once it reads a different value is that, without additional knowledge about the program being executed, no assumption can be made on what statements are affected by that value being changed. On the other hand, a major advantage of this independence from the program is that the inferred traces correspond to real executions of any program which can produce the original trace. Let us now define the *maximal* set of executions which can be inferred from an observed execution as all traces obtainable from τ using the feasibility axioms.

Definition 3 *The* ***feasibility closure*** *of a feasible trace τ, written feasible(τ), is the smallest set of feasible traces containing τ which is prefix-closed and satisfies the thread determinism. A trace in feasible(τ) is called* ***τ-feasible****.*

To emphasize that our model, based on an axiomatization of feasibility, infers more equivalent executions from an observed trace than any other sound similar technique (without assuming language-specific knowledge about the system), let us analyze the example in Figure 3, which recounts Peterson's solution to the mutual exclusion problem between two threads [25], and presents a trace observed for a possible run, in which first thread completes before the second starts. Existing sound techniques, e.g., [22, 30], would only infer feasible traces which are permutations of the observed trace. The feasibility closure does not assume that all inferable traces are permutations of the original trace, but rather uses the axiomatization of the system to gain maximal coverage. Therefore, it should not come as a surprise that the feasibility closure contains all traces inferable through the above mentioned techniques (since all of them, being sound, comply to the axiomatization of the system), and even more. For example, the feasibility closure associated with the observed trace in Figure 3 also contains the trace $\tau' = e_1e_2e_3e_7e_8e'_9e'_4$, where e'_4 is $(1, \mathit{read}, \mathit{turn}, 2)$, and e'_9 is $(2, \mathit{read}, q1, 1)$, which is not a prefix of any permutation of the observed trace. Being solely based on the observed trace, our axiomatization cannot be used to infer additional events for a thread, once this thread departs from the observed trace. Therefore, there is no other trace in the feasibility closure having τ' as a prefix, since the final events for both threads of τ' (e'_4 and e'_9) did not occur in the original trace. However, if one knows more about the (data and control) dependencies induced by the program being executed, one could continue this trace and infer that e_{10} can be generated, since it only depends on e_8. Moreover, if a constraint solver is employed, it could easily infer that the condition in which e_3 and e'_4 are involved is still false, as in the original execution, thus Thread 1 can be continued to generate events e_5 and e_6. At the extreme, if one fully simulates the execution of the code starting from the state obtained upon generating τ', even more possible ways to continue this trace can be discovered (actually, an unbounded number of possible continuations can be obtained). We here prefer to keep the analysis at a more abstract (and fully decidable) level, to find out how much the coverage of a trace analysis technique can be increased without relying on the semantics of a particular language.

This is a good time to formally compare our feasibility closure of a trace τ with the set of traces equivalent with τ in the standard happens-before causal model. The happens-before trace equivalence class can be elegantly captured as the Mazurkiewicz trace associated to the dependence given by the happens-before relation. Let $\pi_x(\tau)$ be the projection of τ to x, keeping only those events whose target is x. The happens-before dependence is a pair $(\mathcal{E}_\tau, T \cup D)$, where $T = \bigcup_t \{(e_1, e_2) \mid \pi_t(\tau) = \tau_1 e_1 e_2 \tau_2\}$ is the intra-thread sequential dependence and $D = \bigcup_x \{(e_1, e_2) \mid \pi_x(\tau) = \tau_1 e_1 e_2 \tau_2 \textit{ such that } e_1 \textit{ or } e_2 \textit{ is a write of } x\}$ is the sequential memory dependence. Given this happens-before dependence, the Mazurkiewicz trace associated with τ is defined as the least set $[\tau]$ of traces containing τ and being closed under permutation of consecutive independent events [20]: if $\tau_1 e_1 e_2 \tau_2 \in [\tau]$ and $(e_1, e_2) \notin T \cup D$, then $\tau_1 e_2 e_1 \tau_2 \in [\tau]$.

The following result, in combination with, e.g., the example based on Peterson's algorithm (Figure 3, or our motivating example (analyzing the trace in Figure 1) —which show that the happens-before causality cannot capture certain causally equivalent traces that our MT-trace theory can—, implies that our MT-trace theory is strictly more general than the theory of Mazurkiewicz traces associated to the happens-before causal order (in multithreaded systems):

Proposition 1 *If $\tau_1 e_1 e_2 \tau_2$ is τ-feasible and $(e_1, e_2) \notin T \cup D$, then $\tau_1 e_2 e_1 \tau_2$ is also τ-feasible.*

This result not only shows that the happens-before equivalence class of the observed trace is included in its feasibility closure, but also that the feasibility closure is closed under the equivalence relation generated by happens-before. In other words, for multithreaded systems, MT-trace theory subsumes the happens-before causality. We next discuss two more complex instances of the MT-trace theory.

2.3 Case Study 1: A Language Definition

We here show an instance of the MT-trace theory above, extracting adequate causal models from the formal semantics of a virtual machine executing multithreaded programs. We therefore present the syntax of a multithreaded language and its operational semantics, including event generation. For this case study, a multithreaded system is a multithreaded program being executed using the defined semantics. We prove that any such system is sequentially consistent.

Syntax. Table 1 shows the syntax of CIMP, a minimal concurrent imperative language. Its imperative part consists of arithmetic expressions with integers and variables, comparison, and statements such as assignment, conditional, or loop. Multiple statements can be executed concurrently using the $\|$ construct. $\mathsf{acquire}$ and $\mathsf{release}$ have the standard semaphore semantics. Semaphore and variable identifiers are disjoint.

Dynamic Semantics. Table 1 presents the semantics of nondeterministic thread interleaving, memory access, and synchronization. The semantics of other constructs is the standard one, not altering the state, nor generating new events, but propagating them. One could add other constructs as well, including state-modifying ones, as long as one follows the shown rules (in altering the state and appending events to the trace). The following types of configuration are used: $\langle \mathsf{Syn}, \mathit{State}, \rangle$, and $\langle \mathsf{Syn}, \mathit{State}, \mathit{Int}\rangle$, with Syn ranging over *AExp*, *BExp*, *Stmt*, and *Pgm*. The final parameter of the second type is the index of the thread currently being executed. X, L, A, B, St, P, σ, and e, are meta-variables ranging over *Var*, *Lock*, *AExp*, *BExp*, *Stmt*,

$$
\begin{array}{rcl}
\mathit{Var} & ::= & \text{variable identifiers} \\
\mathit{Lock} & ::= & \text{semaphore identifiers} \\
\mathit{AExp} & ::= & \text{integer numbers} \mid \mathit{Var} \mid \mathit{AExp} + \mathit{AExp} \mid \ldots \\
\mathit{BExp} & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathit{AExp} \leq \mathit{AExp} \mid \ldots \\
\mathit{Stmt} & ::= & \mathsf{skip} \mid \mathit{Stmt}; \mathit{Stmt} \mid \mathit{Var} := \mathit{AExp} \\
 & \mid & \mathsf{if}\ \mathit{BExp}\ \mathsf{then}\ \mathit{Stmt}\ [\mathsf{else}\ \mathit{Stmt}] \mid \mathsf{while}\ \mathit{BExp}\ \mathsf{do}\ \mathit{Stmt} \\
 & \mid & \mathsf{acquire}(\mathit{Lock}) \mid \mathsf{release}(\mathit{Lock}) \mid \ldots \\
\mathit{Pgm} & ::= & \mathit{Stmt} \mid \mathit{Stmt} \parallel \mathit{Pgm}
\end{array}
$$

SYNTAX

SEMANTICS

$$(0) \qquad \frac{\langle P, \sigma, 1\rangle \xrightarrow{\chi} \langle P', \sigma'\rangle}{\langle P, \sigma\rangle \xrightarrow{\chi} \langle P', \sigma'\rangle}$$

$$(1) \qquad \frac{\langle St, \sigma, tId\rangle \xrightarrow{\chi} \langle St', \sigma'\rangle}{\langle St \parallel P, \sigma, tId\rangle \xrightarrow{\chi} \langle St' \parallel P, \sigma'\rangle} \qquad (2) \qquad \frac{\langle P, \sigma, tId + 1\rangle \xrightarrow{\chi} \langle P', \sigma'\rangle}{\langle St \parallel P, \sigma, tId\rangle \xrightarrow{\chi} \langle St \parallel P', \sigma'\rangle}$$

$$(3) \qquad \frac{\cdot}{\langle X, \sigma, tId\rangle \xrightarrow{e} \langle I, \sigma\rangle},\ \texttt{if}\ \ \sigma(X) = I,\quad e = (\mathit{thread} = tId, \mathit{type} = \mathit{read}, \mathit{target} = X, \mathit{state} = I)$$

$$(4) \qquad \frac{\cdot}{\langle X \texttt{:=} I, \sigma, tId\rangle \xrightarrow{e} \langle \mathsf{skip}, \sigma[I/X]\rangle},\ \texttt{if}\quad e = (\mathit{thread} = tId, \mathit{type} = \mathit{write}, \mathit{target} = X, \mathit{state} = I)$$

$$(5) \qquad \frac{\cdot}{\langle \mathsf{acquire}(L), \sigma, tId\rangle \xrightarrow{e} \langle \mathsf{skip}, \sigma[\sigma[L] - 1/L]\rangle},\ \texttt{if}\quad \sigma(L) > 0, e = (\mathit{thread} = tId, \mathit{type} = \mathit{acquire}, \mathit{target} = L)$$

$$(6) \qquad \frac{\cdot}{\langle \mathsf{release}(L), \sigma, tId\rangle \xrightarrow{e} \langle \mathsf{skip}, \mathit{locks}[\mathit{locks}[L] + 1/L]\rangle},\ \texttt{if}\quad e = (\mathit{thread} = tId, \mathit{type} = \mathit{release}, \mathit{target} = L)$$

$$\frac{\cdot}{\langle P, \sigma\rangle \xrightarrow{\epsilon}{}^{*} \langle P, \sigma\rangle} \qquad \frac{\langle P, \sigma\rangle \xrightarrow{\tau}{}^{*} \langle P'', \sigma''\rangle \ \langle P'', \sigma''\rangle \xrightarrow{\chi} \langle P', \sigma'\rangle}{\langle P, \sigma\rangle \xrightarrow{\tau\chi}{}^{*} \langle P', \sigma'\rangle}$$

Table 1: Language CIMP: Syntax and Trace Semantics.

Pgm, *State*, and *Event*, respectively. Names (variables and semaphores) are shared among all threads. We let σ_ϵ denote the initial state, assigning 0 to variable names and 1 to semaphore names.

The dynamic semantics is based on interleaving: at any moment, exactly one thread is advanced one step (rules 0–2). Threads are indexed from left to right in their original parallel construct list, starting with 1. *tId* identifies the thread currently executed. Synchronization is achieved through rules (5) and (6). Rule (5) says a thread can acquire one semaphore resource iff the counter associated to the semaphore in the store is positive, in which case it decreases the number of locks available for that variable. Rule (6) increases the number of available locks for a semaphore.

Trace Semantics. Events are emitted for reads/writes of variables and acquire/release of locks (rules 3–6). Metavariable χ stands for either an event, or the empty word ϵ (internal transition). Other transitions simply propagate events using the χ variable, or are internal transitions, as the dissolution of skip in the semantics of sequential composition:

$$\frac{\langle St_1, \sigma\rangle \xrightarrow{\chi} \langle St_1', \sigma'\rangle}{\langle St_1; St_2, \sigma\rangle \xrightarrow{\chi} \langle St_1'; St_2, \sigma'\rangle}$$

$$\frac{\cdot}{\langle \mathsf{skip}; St, \sigma\rangle \xrightarrow{\epsilon} \langle St, \sigma\rangle}$$

Relation $\xrightarrow{\tau}^*$ is the reflexive and transitive closure of relation $\xrightarrow{\chi}$. If $(p, \sigma) \xrightarrow{\tau}^* (p', \sigma')$ can be derived using the rules, we say that (p, σ) *evolves to* (p', σ') *emitting trace* τ, and write it as $\mathsf{CIMP} \vdash (p, \sigma) \xrightarrow{\tau}^* (p', \sigma')$. If $\mathsf{CIMP} \vdash (p, \sigma_\epsilon) \xrightarrow{\tau}^* (p', \sigma')$, we say that p *yields* τ and write it as $p \leadsto \tau$. Let $\mathit{Traces}(p)$ be the set of all traces yielded by p.

Adequacy. The definition above satisfies the consistency and the feasibility axioms. First, the semantics guarantees that the state is uniquely determined by the trace:

Proposition 2 *If CIMP* $\vdash \langle p, \sigma_\epsilon\rangle \xrightarrow{\tau}^* \langle p', \sigma\rangle$ *then:*
1. $\sigma(x) = \mathit{state}(\mathit{latest}_\tau^{\mathit{write}}(x))$*, for any variable* x*; and*
2. $\sigma(s) = \mathit{available}_\tau(s)$*, for any semaphore name* s*.*

Let σ_τ denote the unique state corresponding to τ. Since the state attribute of a read event generated by CIMP is the current value for that event—rule (3), write-read consistency is ensured. Moreover, since acquire events can only be generated if the current counter for that semaphore is positive—rule (5), the semaphore invariant is also maintained.

Corollary 1 *All CIMP traces are sequentially consistent.*

It is clear that if $p \leadsto \tau e$, then also $p \leadsto \tau$, so $\mathit{Traces}(p)$ is prefix closed. Semantic rules can only deterministically advance a thread once it it selected by rules (0–2). However, due to internal transition rules, there could be multiple programs p' obtained by a derivation of p producing a trace τ. A *maximally executed program* for a trace $\tau \in \mathit{Traces}(p)$, is a program p' such that $\mathsf{CIMP} \vdash \langle p, \sigma_\epsilon\rangle \xrightarrow{\tau}^* \langle p', \sigma_\tau\rangle$ and any possible step forward will produce a new event; that is, if $\mathsf{CIMP} \vdash \langle p', \sigma_\tau\rangle \xrightarrow{\chi} \langle p'', \sigma_{\tau\chi}\rangle$, then $\chi = e$ for some event e. Because all rules not accessing the state are just syntax transformations, and because all rules accessing the state are recorded in the trace, the trace uniquely determines the maximal program obtained upon producing it.

Proposition 3 *Given a trace* $\tau \in \mathit{Traces}(p)$*, there exists a unique maximally executed program* p_τ *for that trace.*

Given a positive integer n, let $\pi_n(p)$ be the *projection of* p *to thread* n, the syntactic counterpart of the trace-to-thread projection $\pi_n(\tau)$, defined as the nth statement in p's "$\|$" list. The execution of a thread must pass through a unique maximally executed program before emitting a new event:

Proposition 4 *If* $(p,\sigma_\epsilon) \xrightarrow{\tau}{}^{*} (p_1,\sigma_\tau) \xrightarrow{e} (p_2,\sigma_{\tau e})$*, then* $\pi_{thread(e)}(p_1) = \pi_{thread(e)}(p_\tau)$*. Moreover if* $\tau_1, \tau_2 \in \mathsf{Traces}(p)$ *such that* $\pi_n(\tau_1) = \pi_n(\tau_2)$*, then* $\pi_n(p_{\tau_1}) = \pi_n(p_{\tau_2})$*.*

The above implies that if the values read by a thread are the same across executions, then that thread will evolve similarly. This allows us to prove the thread determinism:

Proposition 5 *If* $\tau e, \tau' \in \mathsf{Traces}(p)$*,* $t = thread(e)$*, and* $\pi_t(\tau) = \pi_t(\tau')$ *then: (1) if* $\tau e' \in \mathsf{Traces}(p)$ *then either* $e = e'$ *or* $thread(e') \neq t$*; (2) if* $\tau' e$ *is consistent then* $\tau' e \in \mathsf{Traces}(p)$*; (3) if* $type(e) = read$ *then there exists* e' *with* $type(e') = read$*,* $target(e') = target(e)$*,* $thread(e') = t$*, and* $\tau' e' \in \mathsf{Traces}(p)$*.*

Since, CIMP traces are consistent, and the sets of program executions also satisfy prefix closeness, we conclude that:

Theorem 1 *Any CIMP program, regarded as a multithreaded system whose feasible executions are given by CIMP's operational semantics, is sequentially consistent.*

2.4 Case Study 2: Sliced Causality

We here discuss another instance of the MT-trace theory, in which only a subset of events (those related to a property of interest) is recorded. This way, the generality of our theory allows not only to recast prior work on sliced causal models into the same uniform theory of MT-traces, but more importantly to apply the general results developed in the rest of the paper to a much more complex causal setting.

Recording all the memory access events can have two major drawbacks: (1) prohibitive runtime overhead, and (2) low analysis coverage. As (1) is clear, we only discuss (2).

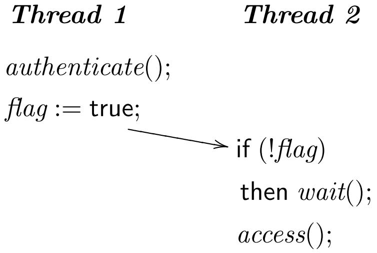
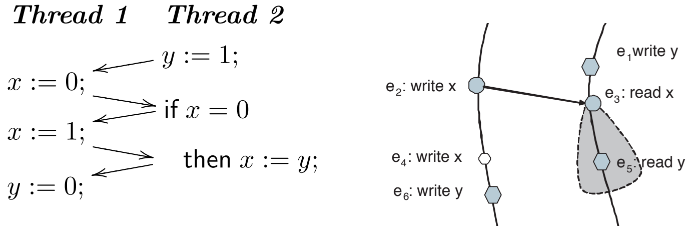


Figure 4: Buggy Synchronization Pattern

Consider a simple and common safety property for a shared resource, that any access should be authenticated. Figure 4 shows a buggy program. Thread 1 authenticates and Thread 2 uses the authenticated resource. They communicate via the *flag*. Synchronization is unnecessary, since only the main thread modifies *flag*. However, the developer makes a (rather common [12]) mistake, using if instead of while in the task thread. Suppose now that we observed a successful run of the program, as shown by the arrow. If all events are recorded in execution traces like in Section 2.3, then the feasibility closure of the observed trace will contain only the observed trace, because its events are totally ordered. Consequently, a low level observer of this execution has no chance to see the potential violation of the property.

Sliced causality [4, 5] is a causal partial order relation that increases the coverage of runtime analysis by dropping irrelevant events and dependencies, such as the write/read dependence caused by accesses to *flag*. The sliced causality is constructed by making use of dependence information obtained both statically and dynamically. Instead of considering all the events produced by the running system, it first slices the trace according to the desired property and then computes the causal partial order on the achieved slice; the slice contains all the property events, i.e., events referred by the property, as well as all the relevant events, i.e., events upon which the property events depend, directly or indirectly. This way, irrelevant causality on events is trimmed without breaking soundness, allowing more permutations of relevant events to be analyzed and resulting in better coverage. In our example, since *access()* is *not* controlled by if, sliced-causality techniques



(a) Example Program (b) Example Trace

Figure 5: Example for relevance dependence

can predict the error from the successful execution. When the bug is fixed replacing if with while, *access()* is controlled by while (since it is a potentially non-terminating loop), so no violation is reported.

We recall some notions and results from [4, 5] and then show that the sliced traces employed by sliced causality also form an instance of our MT-trace theory, provided that the original, unsliced traces form one. Event e' *depends upon* event e, written $e \sqsubset e'$, iff a change of e may change or eliminate e'; in other words, *e should occur before e' in any consistent permutation of events.* We distinguish: (1) *control dependence*, $e \sqsubset_{ctrl} e'$, when a change of the state of e may eliminate e'; and (2) *data-flow dependence*, written $e \sqsubset_{data} e'$, when a change of the state of e may lead to a change in the state of e'. Relation $\sqsubset_{ctrl}$, unlike $\sqsubset_{data}$, only relates events generated by the same thread.

An additional dependence, *relevance dependence*, is also needed. In Figure 5, threads 1 and 2 are executed as shown (a), yielding trace $e_1e_2e_3e_4e_5e_6$ (b)—this trace is incomplete, being observed before the final write of x in Thread 2. Suppose the property to check refers only to y; the property events are then e_1, e_5, and e_6. Events e_2 and e_3 are clearly relevant, as $e_2 \sqsubset_{data} e_3 \sqsubset_{ctrl} e_5$. If only closures under $\sqsubset_{ctrl}$ and $\sqsubset_{data}$ were used to compute the relevant events, like in dynamic slicing [1], then e_4 would be irrelevant, so one may conclude that $e_2e_6e_1e_3e_5$ is a feasible permutation; there is, obviously, no execution that can produce that trace, so one reported a false alarm. Consequently, e_4 is also relevant and $e_3 \sqsubset_{rlvn} e_4$.

Sliced causality, written $\sqsubset$, is the transitively closed union of $\sqsubset_{ctrl}$, $\sqsubset_{data}$, $\sqsubset_{rlvn}$. Given a trace τ and a property φ, let $\mathcal{E}_\tau\restriction_\varphi$ be the subset of $\mathcal{E}_\tau$ containing the φ-property events. Given a set (of property events) $P \subseteq \mathcal{E}_\tau$, let $sliced_P(\tau)$ be the "subtrace" of τ containing the events from P and all the events they $\sqsubset$-depend on; these events will be called *relevant* for P in τ. It has been shown that any permutation of P consistent with $\sqsubset\restriction_{\mathcal{E}_{sliced_P(\tau)}\times\mathcal{E}_{sliced_P(\tau)}}$ and the thread ordering corresponds to some execution of the multithreaded system [4, 5].

We will next briefly sketch how one could capture the sliced causality as an instance of our feasibility framework, further extending the coverage obtainable by using it. *Sliced traces* are obtained from the original ones by using $\sqsubset$ to obtain the events relevant to a set of (property) events, filtering out the non-relevant ones. To be able to apply our feasibility model to the sliced traces of a system, we regard these sliced traces as traces of a virtual multithreaded system.

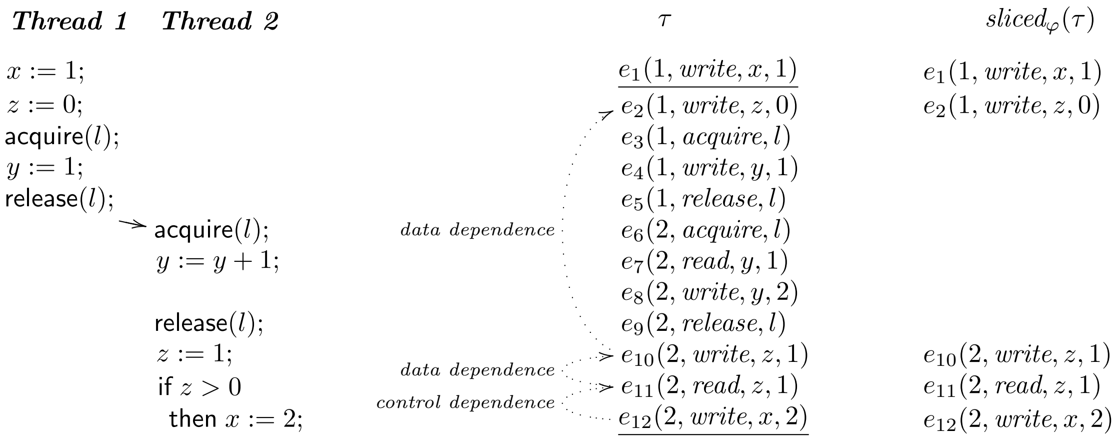
Let $\mathcal{S}$ be the original multithreaded system whose traces are to be sliced according to some arbitrary but fixed property φ. Recall that the multithreaded system is hypothetical in Sections 2.1 and 2.2, in the sense that all it matters is the set of traces that $\mathcal{S}$ can produce; how traces are effectively produced, or how $\mathcal{S}$ is implemented, or even whether $\mathcal{S}$ exists at all as an actual running system, is irrelevant (which is a strong point of MT-trace theory, allowing it to be used here). The *φ-slice of a trace* τ, written $sliced_\varphi(\tau)$, is $sliced_{\mathcal{E}_\tau\restriction_\varphi}(\tau)$, the trace obtained from τ by erasing all non-φ-relevant events.

Let $sliced_\varphi(\mathcal{S})$ be the virtual multithreaded system "producing" the φ-sliced traces, that is,

$$feasible(sliced_\varphi(\mathcal{S})) = sliced_\varphi(feasible(\mathcal{S})).$$

For any trace $\tau' \in feasible(sliced_\varphi(\mathcal{S}))$, let $feasible'(\tau')$ be the set of permutations of τ' consistent with $\sqsubset\restriction_{sliced_\varphi(\mathcal{E}_\tau)\times sliced_\varphi(\mathcal{E}_\tau)}$. The main result of [4] can be restated here as:

Theorem 2 *$feasible'(sliced_\varphi(\tau)) \subseteq sliced_\varphi(feasible(\mathcal{S}))$, for any trace $\tau \in feasible(\mathcal{S})$.*



```
Thread 1      Thread 2
x := 1;
z := 0;
acquire(l);
y := 1;
release(l);
              acquire(l);
              y := y + 1;

              release(l);
              z := 1;
              if z > 0
                then x := 2;
```

(a) Program

τ

$\underline{e_1(1, write, x, 1)}$
$e_2(1, write, z, 0)$
$e_3(1, acquire, l)$
$e_4(1, write, y, 1)$
$e_5(1, release, l)$
$e_6(2, acquire, l)$
$e_7(2, read, y, 1)$
$e_8(2, write, y, 2)$
$e_9(2, release, l)$
$e_{10}(2, write, z, 1)$
$e_{11}(2, read, z, 1)$
$\underline{e_{12}(2, write, x, 2)}$

(b) Trace

$sliced_\varphi(\tau)$

$e_1(1, write, x, 1)$
$e_2(1, write, z, 0)$
$e_{10}(2, write, z, 1)$
$e_{11}(2, read, z, 1)$
$e_{12}(2, write, x, 2)$

(c) Sliced trace

Figure 6: Slicing an observed trace based on given (underlined) property events

feasible$'$ was subsequently relaxed to handle synchronized blocks by means of lock sets [5]. Nevertheless, the class of equivalent feasible sliced traces can be further extended in the sense of the feasibility model proposed in this paper. Consider the trace τ in Figure 6(b), and suppose that our property φ is "e_1 *and* e_{12} *are in a race*". Let $e'_7(2, read, y, 1)$ be the event similar to e_7, but in which the value is replaced by 1. Either manually, or by running the algorithm for model-checking the feasibility closure of a trace (Algorithm 1, Section 3), one can verify that the feasibility closure of τ consists of the following proper traces (and their prefixes): τ, $e_1e_2e_6e'_7$, $e_1e_6e_2e'_7$, $e_1e_6e'_7e_2$, $e_6e_1e_2e'_7$, $e_6e_1e'_7e_2$, $e_6e'_7e_1e_2$, neither of these traces satisfying φ. Suppose now we know that the trace was generated by executing the program in Figure 6(a). Sliced causality computes the closure of control and data dependencies based on the trace and the program, yielding the "sliced" trace in Figure 6(c). However, because e_{10} data depends on e_2, no race can be detected between e_1 and e_{12} using the technique in [4, 5]; one can only detect a race between e_2 and e_{10}, which could not be detected by the feasibility closure of the original observed trace τ, since it corresponds to a trace like $e_1e_6e'_7e'_8e_9e_2$, not inferable without additional knowledge about the program, where e'_8 is $(2, write, y, 1)$. However, assuming the virtual system generating sliced traces, we can compute the feasibility closure directly on $sliced_\varphi(\tau)$, the sliced trace in Figure 6(c). Let e'_{11} denote event $(2, read, z, 0)$; then $feasible(sliced_\varphi(\tau))$ consists of the following traces and their prefixes[4]: $sliced_\varphi(\tau)$, $e_1e_{10}e_2e'_{11}$, $e_1e_{10}e_{11}e_2e_{12}$, $e_1e_{10}e_{11}e_{12}e_2$, $e_{10}e_1e_2e'_{11}$, $e_{10}e_1e_{11}e_2e_{12}$, $e_{10}e_1e_{11}e_{12}e_2$, $e_{10}e_{11}e_1e_2e_{12}$, $e_{10}e_{11}e_1e_{12}e_2$, $e_{10}e_{11}e_{12}e_1e_2$. The latter two exhibit the race between e_1 and e_{12}; for example, the latter trace can be obtained as the sliced trace of the feasible trace $e_6e'_7e'_8e_9e_{10}e_{11}e_{12}e_1e_2$.

Corollary 2 *The solid line inclusions in the diagram below hold for any trace τ in feasible$(\mathcal{S})$:*

$$\begin{tikzcd}
sliced_\varphi(feasible(\tau)) \arrow[r, hook] \arrow[d, hook] & sliced_\varphi(feasible(\mathcal{S})) \\
feasible(sliced_\varphi(\tau)) \arrow[ur, hook, dashed] & feasible'(sliced_\varphi(\tau)) \arrow[l, hook'] \arrow[u, hook, "Theorem\ 2"']
\end{tikzcd}$$

PROOF. The topmost inclusion comes from the fact that both $feasible(\mathcal{S})$ and $feasible(\tau)$ are closed under prefixes, thread determinism, and consistency, both contain τ, and $feasible(\tau)$ is the smallest set with those properties. The left and bottom inclusions are both consequences of the forthcoming Theorem 3, which constructively characterizes the feasibility closure. Intuitively, the left inclusion holds because, since $sliced_\varphi(\cdot)$ removes elements from the input trace, there are less constraints when permuting the remaining events; also,

[4] One can obtain these traces by running Algorithm 1 on the sliced trace.

all events which could be permuted originally, can still be permuted in the sliced trace. The intuition for the bottom inclusion is that the axioms defining $\mathit{feasible}'(\mathit{sliced}_\varphi(\tau))$ imply those generating $\mathit{feasible}(\mathit{sliced}_\varphi(\tau))$ □

Since $\sqsubset_{rlvn}$ was defined with the notion of a partial order in mind, one should not expect that the dashed inclusion holds as-is. The example above yields feasible traces because there is no relevance dependence involved. However, our experiments show that unsoundness due to unconsidered relevance dependencies is rather rare: no false alarms were generated during our evaluation presented in Section 5. We conjecture that, given a proper notion of relevance, the dashed inclusion will also hold, enabling any trace analysis technique to be applied to the feasible (sliced) executions of $\mathit{sliced}_\varphi(\mathcal{S})$ with the guarantee of soundness w.r.t. the original system $\mathcal{S}$. In particular, any of the techniques developed for MT-traces, including the model-checking and race-detection techniques in Sections 3 and 4, could be used in combination with sliced causality to improve the coverage of the analysis by first dropping irrelevant events.

3 Model Checking the Feasibility Closure

Section 2 showed how the maximal causal model can be naturally defined by characterizing feasibility through closure properties rather than constructively. This section presents a constructive characterization of the feasibility closure, concretized in a model checking algorithm. Similarly to happens-before model checking algorithms, the complexity of our algorithm is dominated by the number of causally equivalent feasible traces being explored; however, its coverage is considerably greater, as shown by Table 2. We here don't fix any particular type of properties to be checked; we simply assume a generic procedure φ to check whether a given trace satisfies the desired property (next section will present a specialized algorithm which can check a particular type of property, dataraces, without generating all the checked traces). When all events in an execution are recorded, this algorithm fully becomes an explicit state model checker, since, as discussed in Section 2.3, a feasible trace completely determines the state obtainable upon producing it, therefore φ could also be used check state assertions in addition to trace properties.

Interleavings. At a closer look at the axioms generating the feasibility closure, specifically the thread determinism, one can see that, to advance a thread t in an equivalent execution, one must guarantee that the projection of the trace to t is the same as in the original execution. Therefore, a good starting point in approximating the feasibility closure is characterizing traces whose projection on each thread is the same as in the observed execution (or a prefix of it). This is faithfully captured by the notion of *interleaving (prefix)*:

Definition 4 *Trace τ' is an **interleaving** of τ if their projections to any of the threads are equal, that is, $\pi_t(\tau) = \pi_t(\tau')$ for each thread t. τ' is an **interleaving prefix** of τ if there exists τ'' such that $\tau'\tau''$ is an interleaving of τ.*

Using the fact that all prefixes of the original trace are in the feasibility closure, and the first part of the thread determinism axiom, one can prove by induction that

Proposition 6 *Any consistent interleaving prefix of a feasible trace τ is τ-feasible.*

Interleaving prefixes cover all possibilities of generating a τ-feasible trace using only the events in τ. However, the definition of interleaving (prefix) overlooks the final part (regarding *read* events) of the thread determinism axiom. To achieve a complete constructive characterization of feasibility closures, we have to go beyond prefixes of interleavings, more exactly, one read per thread beyond. This is because, as guaranteed by thread determinism, whenever all events before a read event have been generated in a thread, the read event can also be generated, but its state might differ than the state it had in the observed trace. However, once a read event whose state is different than the one in the original trace is derived, the execution cannot be continued for that thread, because the read might have guarded (some of) the statements following it.

An *extended interleaving prefix* is a trace which on all threads behaves similarly to the observed trace up to the last read, which might have a different value:

```
    Input: Feasible trace $\tau_0$ of size $n$ over $\mathsf{k}$ threads.
    Maps: $\mathsf{thread} : \{1, \ldots, \mathsf{k}\} \to \mathit{Stack}, \sigma : \mathit{Locations} \to \mathit{Int}$
 1  for $t = \overline{1, k}$ do $\mathsf{thread}[t] \leftarrow \pi_t(\tau_0)$          // first event at top
 2  $\mathsf{Enabled} \leftarrow \{1, \ldots, \mathsf{k}\}$                                  // advanceable threads
 3  Initialize $\sigma$ with 0 for variables and 1 for semaphores
 4  $\tau \leftarrow \epsilon$; $t \leftarrow 0$
 5  while $t < \mathsf{k}$ do
 6      $t \leftarrow t + 1$
 7      if $t \in \mathsf{Enabled}$ then
 8          $e \leftarrow \mathtt{top}(\mathsf{thread}[t])$
 9          if $\mathit{type}(e) \neq \mathit{acquire} \vee \sigma[\mathit{target}(e)] > 0$ then          // advance
10              $l \leftarrow \mathit{target}(e)$
11              if $\mathit{type}(e) = \mathit{read} \wedge \mathit{state}(e) \neq \sigma[l]$ then // extended prefix
12                  $\mathsf{Enabled} \leftarrow \mathsf{Enabled} \setminus \{t\}$
13                  $\mathit{state}(e) \leftarrow \sigma[l]$
14              else                                                   // update state
15                  $\mathtt{pop}(\mathsf{thread}[t])$
16                  if $\mathit{type}(e) = \mathit{write}$ then $\sigma[l] \leftarrow \mathit{state}(e)$
17                  if $\mathit{type}(e) = \mathit{acquire}$ then $\sigma[l] \leftarrow \sigma[l] - 1$
18                  if $\mathit{type}(e) = \mathit{release}$ then $\sigma[l] \leftarrow \sigma[l] + 1$
19              $\mathtt{push}(\tau, e)$; check $\tau$ against $\varphi$
20              $t \leftarrow 0$
21      while $t = \mathsf{k} \wedge \tau \neq \epsilon$ do                                // backtrack
22          $e \leftarrow \mathtt{pop}(\tau)$; $t \leftarrow \mathit{thread}(e)$; $l \leftarrow \mathit{target}(e)$
23          if $t \notin \mathsf{Enabled}$ then                              // extended prefix
24              $\mathsf{Enabled} \leftarrow \mathsf{Enabled} \cup \{t\}$
25          else                                                       // restore state
26              $\mathtt{push}(\mathsf{thread}[t], e)$
27              if $\mathit{type}(e) = \mathit{write}$ then $\sigma[l] \leftarrow \mathit{state}(\mathtt{latest}(\tau, l))$
28              if $\mathit{type}(e) = \mathit{acquire}$ then $\sigma[l] \leftarrow \sigma[l] + 1$
29              if $\mathit{type}(e) = \mathit{release}$ then $\sigma[l] \leftarrow \sigma[l] - 1$
```

Algorithm 1: Model Checking the Feasibility Closure

Definition 5 *Trace $\tau' = \tau_1 e'$ is an **extended prefix** of $\tau = \tau_1 e \tau_2$ if $e = e'$ or $type(e) = type(e') = read$ and $thread(e) = thread(e')$ and $target(e) = target(e')$. τ' is an **extended interleaving prefix** of τ if $\pi_t(\tau')$ is an extended prefix of $\pi_t(\tau)$ for any thread t.*

Consistent extended interleaving prefixes give us a sound and complete characterization for the τ-feasible traces:

Theorem 3 *Given a feasible trace τ, a trace τ' is τ-feasible iff it is a consistent extended interleaving prefix of τ.*

Model Checking Algorithm. Algorithm 1 can be used to explore (and check properties against) the feasibility closure of a given trace. It takes as input a trace τ_0 and a procedure φ saying whether a property is satisfied by a (partial) trace (and state), and checks whether all traces in the feasibility closure of τ_0 (and their corresponding states) satisfy the property of φ. In the initialization phase (lines 1–4), the original trace is split into threads and each thread projection is loaded into a stack, with first events in the thread at top of the stack, and the store initialized with 0 for variables and 1 for semaphores. We additionally maintain a set of enabled threads, that is, threads for which all events generated had the same state as in the original execution, therefore they can still be advanced. The trace being created, τ, is also maintained as a stack, initially empty, but with first events at bottom of the stack. Variable t keeps track of the index of the thread which should be advanced next. The main loop (lines 5–29) is a backtracking loop, exiting only when the entire space has been explored. Inside the loop, first part (lines 6–9) checks whether next thread can be advanced. If a thread is found, the state is modified accordingly (lines 15–18), disabling further advances to the thread, if the state of the added event differs from the one in the observed trace (lines 11–13); note that in the latter case, the top event in the corresponding thread needs not be removed, since the thread is disabled. Then, τ is advanced and added to the result set, property φ is being checked (line 19), and the search for next advanceable thread is restarted (line 20). If no additional threads can be advanced from this state, the algorithm backtracks, undoing the effects on the state of previous advances (lines 21–29).

A simple amortized analysis of our model-checking algorithm shows that, without any additional knowledge about the property to check φ, it essentially performs a minimal amount of work: it generates and checks against φ each consistent extended interleaving prefix of τ_0, searching for each next event through the tops of at most k thread stacks. Supposing that φ is a simple safety property taking constant time and memory to evaluate in any given state σ, which is frequently the case in many situations, the time complexity of our algorithm is $\mathcal{O}(|\mathit{feasible}(\tau_0)| \times k)$ and its memory complexity is $\mathcal{O}(|\tau_0|)$; recall that $\mathit{feasible}(\tau_0)$ is closed under prefixes. On the other hand, if the property can specify arbitrarily complex history-based behaviors taking time $f(n, m)$ and memory $g(n, m)$ to check a trace of size n against a property of size m for some arbitrarily complex f and g functions, then the complexity of our algorithm can also be arbitrarily high: $\mathcal{O}(|\mathit{feasible}(\tau_0) \times (k + f(|\tau_0|, |\varphi|)))$ timewise and $\mathcal{O}(|\tau_0| + g(|\tau_0|, |\varphi|))$ memory-wise. Calculating exact bounds for different specification formalisms is beyond our scope.

4 On-The-Fly Datarace Detection

The trace model-checking algorithm above is expensive and likely unfeasible for on-the-fly analysis. As discussed in Section 1, many techniques have been proposed for detecting dataraces on-the-fly: some of them are based on extracting happens-before causal partial orders and tend to be sound, others are based on lock-sets maintained for each location and tend to be unsound, while others, called hybrid techniques, combine the two to gain coverage from lock-sets and precision from happens-before. We propose a novel hybrid technique below, based on a causal model which is maximal for an axiomatics which is stronger (so less general) then the one underlying MT-trace theory. The models of this new axiomatics resemble happens-before causalities, so we call them causal orders. Interestingly, the causal models underlying each of the existing on-the-fly datarace detection techniques that we are aware of already satisfy these stronger axioms, which means that the maximal causal order proposed below already subsumes the existing ones. Moreover, unlike our maximal causal models, it is amenable for on-the-fly datarace detection. An algorithm is also given.

Causal Dataraces. Trace $\tau = \tau_1 e_1 e_2 \tau_2$ is in a datarace on e_1 and e_2 if $\mathit{thread}(e_1) \neq \mathit{thread}(e_2)$, $\mathit{target}(e_1) = \mathit{target}(e_2)$, and $\mathit{write} \in \{\mathit{type}(e_1), \mathit{type}(e_2)\}$. Since the feasibility closure of τ comprises all the traces which are causally equivalent to τ, we introduce the next:

Definition 6 *Feasible trace $\tau = \tau_1 e_1 \tau_2 e_2 \tau_3$ has a* ***causal datarace*** *on e_1 and e_2 iff it has an extended interleaving prefix $\tau' e'_1 e'_2$ in a datarace on e'_1 and e'_2, such that $\pi_{\mathit{thread}(e_1)}(\tau_1) = \pi_{\mathit{thread}(e'_1)}(\tau')$ and $\pi_{\mathit{thread}(e_2)}(\tau_1 \tau_2) = \pi_{\mathit{thread}(e'_2)}(\tau')$.*

A causal datarace identifies the events being in a potential race in the original execution by their thread history, even though their *state* might differ in the execution exhibiting the race. By the thread determinism axiom, and Theorem 3 we have that e_i is identical with e'_i, $i \in \{1, 2\}$, except its state attribute, which could be different. Using the causal datarace property above in combination with Algorithm 1 is expensive, particularly for on-the-fly analysis, since it generates the entire feasibility closure for the trace, when one only looks whether two events can be brought next to each other. Therefore, we prefer a different approach here, based on a stronger axiomatization.

Mutex Discipline. Semaphores are rarely used in their full generality in practice, since they can be acquired in one thread an released in another. More restricted synchronization mechanisms are common, such as locks, which can be seen as special semaphores respecting the following:

Definition 7 *A trace τ respects the* ***mutex discipline*** *if for any prefix τ' of τ, any thread t, and any semaphore s, $available_{\pi_t(\tau')}(s) \in \{0,1\}$ and $available_{\tau'}(s) \in \{0,1\}$.*

The mutex discipline implies semaphore consistency.

Write-Read Dependence. The feasibility closure of a trace only requires that each read event reads the same value as in the original trace. While this generality increases the coverage and is necessary to capture the maximal causality, it also increases the need for search. We next replace this expensive consistency requirement with the more restrictive but easier to compute write-read event dependence:

Definition 8 *e_2* ***write-read depends on*** *e_1 in $\tau = \tau_1 e_1 \tau_2 e_2 \tau_3$, written $e_1 <_\tau^{wr} e_2$, if $target(e_1) = target(e_2)$, $type(e_1) = write$, $type(e_2) = read$, and for all $e \in \mathcal{E}_{\tau_2}$, either $target(e) \neq target(e_1)$, or $type(e) \neq write$.*

$e_1 <_\tau^{wr} e_2$ iff the value read by e_2 is the value written by e_1.

Recall that an extended interleaving of τ may contain some read events with different states than their counterparts in τ. Given an extended interleaving τ' of τ, let the *$\tau-$prefix* of τ' be the largest prefix of τ' which is an interleaving prefix of τ (obtained by erasing the final read events having a different *state* than the one in τ.)

Definition 9 *A interleaving prefix τ' of τ preserves the write-read dependence of τ iff $<_{\tau'}^{wr} = <_\tau^{wr} \cap (\mathcal{E}_{\tau'} \times \mathcal{E}_{\tau'})$. An extended interleaving prefix of τ preserves the write-read dependence if its $\tau-$prefix preserves it.*

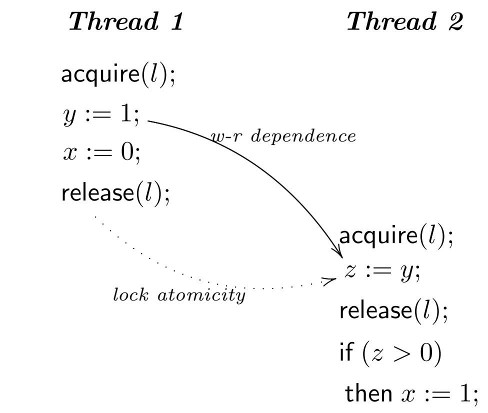
We can now prove the sequential consistency property:

Proposition 7 *An extended interleaving prefix of a feasible trace τ that satisfies the mutex discipline and preserves write-read dependence is τ-feasible.*

The implicit maximal causal model obtained like in Section 2 but using the mutex discipline and write-read dependence axioms above is reminiscent to the one proposed in [30]. However, we prefer not to reduce locking to write/read events and we do not promote model-checking this race-driven model, as [30] proposes. For model-checking purposes we propose the maximal causal model like in Section 3, while for datarace detection we propose a specialized algorithm in the sequel. We next transform the above causal model into a causal partial order, similar to the hybrid approach in [24], but more general and sound. This causal order can be used to detect dataraces without generating feasible traces.

Thread Ordering. Sequentially consistent interleavings preserve the order in which operations of each thread are executed. The *thread ordering* $<_\tau^t$ of τ is given by $e <_\tau^t e'$ if $e <_\tau e'$ and $thread(e) = thread(e')$. One could equivalently define an interleaving of a trace τ as being a trace τ' such that $\mathcal{E}_{\tau'} = \mathcal{E}_\tau$, and $<_{\tau'}^t = <_\tau^t$.

Lock Atomicity. Lock-set approaches, including hybrid ones [9, 24, 27], focus on the mutual exclusion imposed by locks but ignore the write-read dependence, and thus may yield false alarms. In Figure 7, the write(x) in thread 1 is protected by l, while the write(x) in thread 2 is not. There is no obvious dependence between the two writes and, indeed, any lockset algorithm (hybrid or not) reports a race here incorrectly (the two acquire/release blocks must be ordered, due to the depicted write-read dependence). A safe approach is to enforce a total order between operations regarding the same lock [28], but that is over-restrictive, as seen



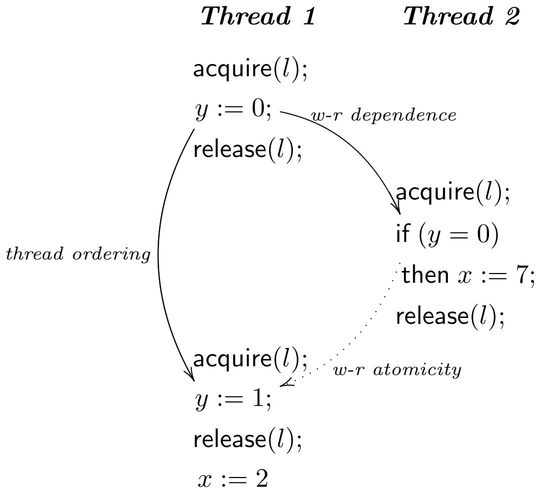
```
Thread 1            Thread 2

acquire(l);
y := 1;
x := 0;
release(l);
                    acquire(l);
                    z := y;
                    release(l);
                    if (z > 0)
                     then x := 1;
```

Figure 7: Motivating Lock Atomicity

in the example in Figure 1. We propose a more general solution: closure under *lock atomicity*. Let *locks*(e) be the set of locks protecting e. For any $s \in$ *locks*(e), let *acquire*$_s(e)$ and *release*$_s(e)$ be the acquire/release event protecting e.

Definition 10 *A relation $\prec$ is closed under **lock atomicity** iff for all events e_1, e_2 and lock s, $e_1 \prec e_2$ and $s \in$ locks$(e_1) \cap$ locks(e_2) implies that release$_s(e_1) \prec e_2$.*

An equivalent formulation is: "*if acquire$_s(e_1) \prec e_2$ and $s \in$ locks(e_2) then release$_s(e_1) \prec e_2$.*"



```
Thread 1        Thread 2

acquire(l);
y := 0;
release(l);
                acquire(l);
                if (y = 0)
                 then x := 7;
                release(l);
acquire(l);
y := 1;
release(l);
x := 2
```

Figure 8: Motivating the Write-Read Atomicity

Write-Read Atomicity. A similar situation arises for the write-read dependence: the atomicity between writes and their corresponding reads must be preserved. Consider the execution in Figure 8. Even if we consider a relation closed under lock atomicity, we still have no dependence between the write(x) in thread 2 and the write(x) in thread 1. However, there is no race on the two events, as the write(x) in thread 2 is protected by the read(y) which would be invalidated by any attempt to bring the two writes on x together. In traditional happens-before approaches, this is solved by enforcing an ordering between a write and its preceding reads in the observed trace. We relax this requirement: a write depends on a previous read only if it also depends on the write that read depends on. Given a read event e, let $write(e)$ be the event such that $write(e) <_{\tau}^{wr} e$.

Definition 11 *Partial order $\prec$ is closed under write-read atomicity iff for all e_1, e_2, if $type(e_1) = read$, $type(e_2) = write$, $target(e_1) = target(e_2)$, and $write(e_1) \prec e_2$, then $e_1 \prec e_2$.*

Datarace Causal Order. We can now define a causal order which can be used to soundly detect dataraces.

Definition 12 *The **datarace causal order** of τ is the smallest partial order containing the thread ordering and the write-read dependence, which is closed under lock atomicity and write-read atomicity. Let $\prec_\tau$ denote this order.*

The main result of this section guarantees that any two events which are not separated by a third event through the causal order and have disjoint lock sets can be brought together in an alternative feasible execution.

Theorem 4 (Causal Datarace) *If $\tau = \tau_1 e_1 \tau_2 e_2 \tau_3$ is a feasible trace, $thread(e_1) \neq thread(e_2)$, $target(e_1) = target(e_2)$, $\{write\} \subseteq \{type(e_1), type(e_2)\}$, $locks(e_1) \cap locks(e_2) = \emptyset$, and there is no $e \in \mathcal{E}_{\tau_2}$ such that $e_1 \prec_\tau e \prec_\tau e_2$, then τ has a causal datarace on e_1 and e_2.*

Online Causal Datarace Checking. Algorithm 2 computes the datarace-driven causal model on the fly, representing it by means of *vector clocks (VCs)* associated with events.

Definition 13 *A **vector clock (VC)** [19] is a mapping $VC : \{1, \ldots, \#\mathsf{Threads}\} \rightarrow Integer$. $VC_1 \geq VC_2$ iff $VC_1(t) \geq VC_2(t)$ for any thread t. If $VC_1 \not\geq VC_2$ and $VC_2 \not\geq VC_2$, they are **incomparable**. Two vector clocks can be **merged**: $merge(VC_1, VC_2)[t] = \max(VC_1[t], VC_2[t])$.*

VCs encode partial orders: if e_1's VC is larger than e_2's in τ, then $e_1 \prec_\tau e_2$. Algorithm 2 updates the causal model by computing the VC of the received event. eventVC stores the event VCs. threadVC keeps the thread VCs. A counter for every shared location, stored in writeCounts, differentiates writes to the location; every write can be uniquely identified as a pair (location, integer). For every write, two VCs are needed: writeVC associates the write with a VC and readVC keeps the VC for the latest read following the write. To enforce lock-atomicity, if a write or read is protected by a lock, we keep two VCs for the corresponding write-lock combination: writeLockVC keeps the VC for the release of the lock that protects a certain write and readLockVC gives the VC for the release of the lock that protects the latest read following a certain write. locks keeps the locks currently held by a thread and lockedWrites and lockedReads track the writes/reads currently protected by a certain lock.

Algorithm 2 works as follows. Suppose an event is received for thread t. We first advance t's VC (line 2) to indicate that a new event is observed in this thread. This way, the intra-thread ordering is ensured. If the event is an acquire of lock l, we add l to the lock set of t (line 4). If the event is a release of l, we go through every write/read (only the latest read is considered for a write) currently protected by l to associate each of them with the VC of the current thread, which is also the VC of the release event (lines 6 to 9). Then the protected write/read set for l is cleared and l is removed from the thread (lines 10 and 11). If the event is a read of location x, we first locate the latest write of x by fetching the writer counter for x (line 13). The counter is used to get the VC for the particular write of x, which is used to update the current thread's VC (line 14, note that $VC.merge(VC')$ is syntactic sugar for $VC = merge(VC, VC')$). The write-read dependency is thus guaranteed. We also need to go through every lock held by the current thread (lines 15 to 17) to ensure (1) if the corresponding write was protected by the same lock, VC is with the one for the release that protected the write (line 16); and (2) add the read into the protected read set for the lock (line 17). (1) is needed to enforce lock-atomicity. Then we associate the read event with the thread's VC (line 18) and use it to update readVC since it is the latest read for the corresponding write (line 19).

When the event is a write of x, we first increase the write counter (line 21). Then the current thread's VC clock is compared with other writes of x (lines 22 to 27). If the former is greater than some write, it

```
Input: A newly received event $e$.
eventVC: $Event \rightarrow \mathsf{VC}$
threadVC: $\{1, \ldots, \#\mathsf{Threads}\} \rightarrow \mathsf{VC}$
writeVC, readVC: $Location \times Integer \rightarrow \mathsf{VC}$
writeLockVC, readLockVc: $Location \times Integer \times$
                          $Location \rightarrow \mathsf{VC}$
locks: $\{1, \ldots, \#\mathsf{Threads}\} \rightarrow lock\ set$
lockedWrites, lockedReads: $Location \rightarrow$
                           $(Location,\ Integer)\ set$
writeCounts: $Location \rightarrow Integer$
$t \leftarrow \mathsf{thread}(e)$, $x \leftarrow target(e)$
$\mathsf{threadVC}[t][t] \leftarrow \mathsf{threadVC}[t][t] + 1$
switch $type(e)$ do
    case $acquire$: $\mathsf{locks}$ [t].add(x)
    case $release$
        forall $(x', i) \in \mathsf{lockedWrites}$ do
            $\mathsf{writeLockVC}[x'][i][x] \leftarrow \mathsf{threadVC}[t]$
        forall $(x', i) \in \mathsf{lockedReads}$ do
            $\mathsf{readLockVC}[x'][i][x].\mathtt{merge}(\mathsf{threadVC}[t])$
        $\mathsf{lockedWrites} \leftarrow \emptyset$, $\mathsf{lockedReads} \leftarrow \emptyset$
        $\mathsf{locks}$ [t].remove(x)
    case $read$
        $c \leftarrow \mathsf{writeCounts}[x]$
        $\mathsf{threadVC}[t].\mathtt{merge}(\mathsf{writeVC}[x][c])$
        forall $l \in \mathsf{locks}[t]$ do
            $\mathsf{threadVC}[t].\mathtt{merge}(\mathsf{writeLockVC}[x][c][l])$
            $\mathsf{lockedReads}[l].\mathtt{add}(x, c)$
        $\mathsf{eventVC}[e] \leftarrow \mathsf{threadVC}[t]$
        $\mathsf{readVC}[x][c].\mathtt{merge}(\mathsf{threadVC}[t])$
    case $write$
        $c \leftarrow \mathsf{writeCounts}[x] \leftarrow \mathsf{writeCounts}[x] + 1$
        forall $i < c$ do
            if $\mathsf{threadVC}[t] \geq \mathsf{writeVC}[x][i]$ then
                $\mathsf{threadVC}[t].\mathtt{merge}(\mathsf{readVC}[x][c])$
                forall $l \in \mathsf{locks}[t]$ do
                    $\mathsf{threadVC}[t].\mathtt{merge}(\mathsf{writeLockVC}[x][c][l])$
                    $\mathsf{threadVC}[t].\mathtt{merge}(\mathsf{readLockVC}[x][c][l])$
        forall $l \in \mathsf{locks}[t]$ do $\mathsf{lockedWrites}[l].\mathtt{add}(x, c)$
        $\mathsf{writeVC}[x][c] \leftarrow \mathsf{eventVC}[e] \leftarrow \mathsf{threadVC}[t]$
```

Algorithm 2: Datarace Causal Order Computation

Program	LOC	Th.	S.V.	H.B.	Maximal	Races
Bank	1.4k	3	7	83	83	1
StringBuffer	1.4k	3	7	8	11	0
Vector	12.1k	18	49	8801	2723993	1
hedc	39.9k	10	119	246924	12083622	4
tsp	706	4	648	N/A	N/A	1
sor	17.7k	4	106	N/A	N/A	0

Table 2: Maximal model v.s. happens-before causality

means that the new write depends on the previous write. Therefore, the VC clock for the previous write's latest read is used to update the current thread's VC clock to enforce the write-read atomicity (line 24). The releases that protected the previous write and its latest read are also considered to ensure the lock atomicity (lines 25 to 27). The write is then added to the protected write set for every lock held by the current thread (line 28). Finally, we associate the write event with the thread's VC and record it in writeVC (line 29).

Proposition 8 *After Algorithm 2 processes trace τ, for any two events $e_1, e_2 \in \mathcal{E}_\tau$ such that $thread(e_1) \neq thread(e_2)$, $target(e_1) = target(e_2)$, and $\{write\} \subseteq \{type(e_1), type(e_2)\}$, if $\mathsf{eventVC}(e_1)$ and $\mathsf{eventVC}(e_1)$ are incomparable and $locks(e_1) \cap locks(e_2) = \emptyset$ then τ has a causal datarace on e_1 and e_2.*

5 Preliminary Evaluation

We implemented Algorithms 1 and 2 and evaluated them using several known datarace benchmarks. Programs were instrumented to log their executions[5]. Each program was executed only once. For every logged trace, we first evaluated the effectiveness of the maximal model, by comparing the number of feasible traces it contains versus the same number for the happens-before causal model. Then we used both the maximal model and the datarace causal model to detect dataraces. The results are presented in Table 2.

First four columns in the table show the benchmarks, along with their size (lines of code), number of threads (Th.) created, and number of shared variables (S.V.) detected. Banking [12] contains common concurrent bug patterns. StringBuffer and Vector are standard library classes of Java 1.4.2. hedc, tsp and sor are from [33]. hedc implements a meta-crawler for searching multiple Internet archives concurrently. tsp is a parallelized solution to the traveling salesman problem. sor is a scientific computation application synchronized by barriers instead of locks.

Fifth and sixth columns show the number of feasible traces causally equivalent to a given execution, using the happens-before (H.B.) causality and our maximal causality, respectively; the N/A means it did not finish in 1 hour. We can see that, for simple programs, like Bank and StringBuffer, both causal models produced similar results since there are few causal dependencies. When the programs are more complex, like Vector and hedc, the maximal model inferred many more feasible executions than the happens-before causality from the same observed execution trace, showing that our causality is not only maximal but indeed has the potential to detect more concurrency errors. However, the results also show that Algorithm 1 may run into a trace-explosion problem quicker than specialized techniques; even though this is good news wrt coverage of the analysis, in practice one may need to devise property-specific techniques to effectively analyze the maximal model, e.g., the datarace causal model.

Last column shows the number of races detected. Every logged execution trace was checked for dataraces using both the maximal model and the datarace causal model; the former reports a race when an (extended) interleaving (prefix) exhibiting the race was inferred. *All known races* in the benchmark programs were detected by both the maximal model, when applicable, and the datarace causal model, some of which undetectable using the sound happen-before causality [4], but detectable by unsound techniques [24]. Timewise,

[5] Since program instrumentation and trace logging are not important for this paper, we omit the details.

race detection using the datarace model was comparable with logging the trace; races in Table 2 were found in seconds.

6 Conclusion

We presented a novel theory of multithreaded traces, based on axioms of memory consistency. Even though we followed the sequentially consistent memory model, we believe that the ideas are general enough to transport to other memory models. A benefit of our approach is that a maximal causal model can be naturally associated to each trace, capturing and allowing to analyze all the feasible and causally equivalent traces without re-executing the system. Two analysis algorithms were presented, a heavy-weight model-checking one for arbitrary properties and a light-weight one for dataraces. Several case studies and experiments show that our approach is not only theoretically compelling, but also compares favorably in practice with existing techniques.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.

[2] T. R. Allen and D. A. Padua. Debugging parallel fortran on a shared memory machine. In *ICPP*, pages 721–727, 1987.

[3] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. A theory of data race detection. In *PADTAD '06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 69–78, New York, NY, USA, 2006. ACM.

[4] F. Chen and G. Roşu. Parametric and Sliced Causality. In *Computer Aided Verification (CAV'07)*, 2007.

[5] F. Chen, T. F. Serbanuta, and G. Rosu. jpredictor: a predictive runtime analysis tool for java. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 221–230, New York, NY, USA, 2008. ACM.

[6] M. Christiaens and K. D. Bosschere. Trade, a topological approach to on-the-fly race detection in java programs. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[7] V. Diekert and G. Rozenberg, editors. *The Book of Traces.* World Scientific, Singapore, 1995.

[8] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University of Eindhoven, The Netherlands, 1965. Reprinted in Programming Languages, F. Genuys, Ed., Academic Press, New York, 1968, 43-112.

[9] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. *SIGPLAN Not.*, 26(12):85–96, 1991.

[10] P. A. Emrath, S. Chosh, and D. A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 580–588, New York, NY, USA, 1989. ACM.

[11] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 89–99, New York, NY, USA, 1988. ACM.

[12] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[13] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in cilk programs. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 1–11, New York, NY, USA, 1997. ACM.

[14] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM.

[15] D. P. Helmbold, C. E. McDowell, and J.-Z. Wang. Analyzing traces with anonymous synchronization. In D. A. Padua, editor, *ICPP (2)*, pages 70–77. Pennsylvania State University Press, 1990.

[16] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Determining possible event orders by analyzing sequential traces. *IEEE Trans. Parallel Distrib. Syst.*, 4(7):827–840, 1993.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of ACM*, 21(7):558–565, 1978.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[19] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[20] A. Mazurkiewicz. Trace theory. In *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[21] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 2006. ACM.

[22] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. In *Advances in Languages and Compilers for Parallel Computing*, pages 109–129. MIT Press, 1990.

[23] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 133–144, New York, NY, USA, 1991. ACM.

[24] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, 2003.

[25] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115 – 116, 1981.

[26] M. Ronsse and K. D. Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[28] E. Schonberg. On-the-fly detection of access anomalies (with retrospective). In K. S. McKinley, editor, *Best of PLDI*, pages 313–327. ACM, 1989.

[29] A. Sen and V. K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *Proceedings of the Seventh International Conference on Principles of Distributed Systems (OPODIS)*, 2003.

[30] K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *FMOODS*, volume 3535 of *LNCS*, pages 211–226. Springer, 2005.

[31] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2008.

[32] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 334–345, New York, NY, USA, 2006. ACM.

[33] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.