Technical Report http://hdl.handle.net/2142/27708, October 2011

Maximal Causal Models for Sequentially Consistent Systems

Traian Florin Șerbănuță Feng Chen Grigore Roșu

Univeristy of Illinois at Urbana-Champaign
{tserban2,grosu}@illinois.edu

Abstract

This paper shows that it is possible to build a *maximal and sound* causal model for concurrent computations from a given execution trace. It is sound, in the sense that any program which can generate a trace can also generate all traces in its causal model. It is maximal (among sound models), in the sense that by extending the causal model of an observed trace with a new trace, the model becomes unsound: there exists a program generating the original trace which cannot generate the newly introduced trace. Thus, the maximal sound model has the property that it comprises *all* traces which *all* programs that can generate the original trace can also generate. The existence of such a model is of great theoretical value. First, it can be used to prove the soundness of non-maximal, and thus smaller, causal models. Second, since it is maximal, the proposed model allows for natural and *causal-model-independent definitions* of trace-based properties; this paper proposes maximal definitions for causal dataraces and causal atomicity. Finally, although defined axiomatically, the set of traces comprised by the proposed model are shown to be effectively constructed from the original trace. Thus, maximal causal models are also amenable for developing practical analysis tools.

1 Introduction

Traces of events describing concurrent computations have been employed in a plethora of methods for testing and analyzing concurrent systems. A common characteristic of all these methods is that one uses an abstraction of a trace, i.e., a *model*, to "predict" (problematic) event patterns occurring in other traces abstracted by the model. Consider, for example, the conventional happens-before causality: if two conflicting accesses to an object are not causally ordered, then a data-race is reported [22]. But is this the best one can do? Of course, not. A series of papers propose more relaxed happens-before causal models where one can also permute blocks protected by the same lock, provided that they access disjoint variables [18], thus discovering new concurrency bugs not observable with plain happens-before. But is this the best one can do? Of course, not.

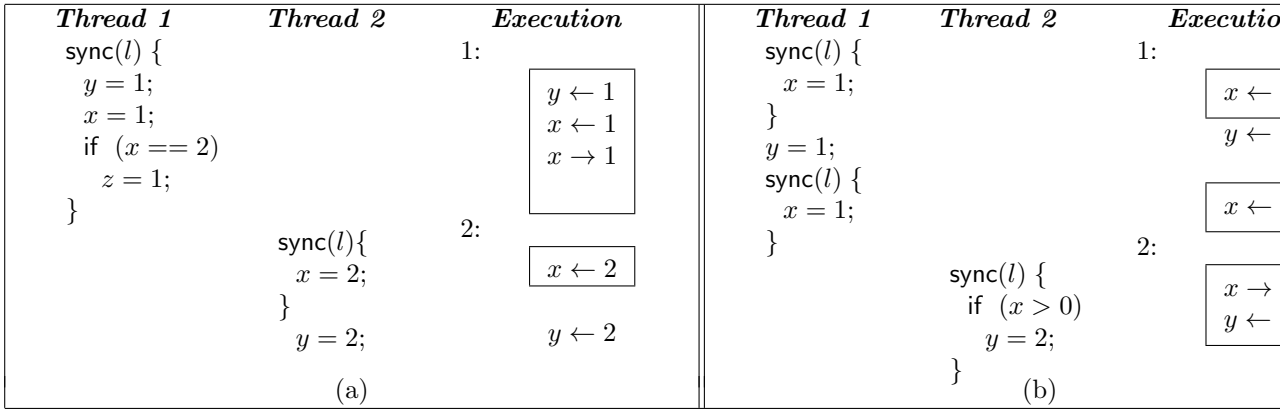

Figure 1: Motivating examples.

Other papers propose models where one can also permute semantic blocks (whose actions are possibly generated by different threads) provided that each read access continues to correspond to the same write access [23, 25]. Others could go even further: Section 5 discusses a series of existing causal models (we only study *sound* models here, i.e., ones which only report real problems in the analyzed systems, allowing developers to focus on fixing those real problems and not on additionally sorting them out from false positives). We would naturally like to know whether there is an end to the question "Is this the best we can do?", that is, whether there is any causal model that can be associated to a given execution trace which comprises the maximum number of causally equivalent traces.

Although most runtime analysis techniques are built upon some underlying sound causal model, possibly relaxed for efficiency reasons, each effort seems to focus more on how to capture it efficiently rather than proving its soundness (often implicitly assumed) or studying its relationship to existing models (other than empirically comparing the number of found bugs). Moreover, since such approaches attempt to extract information from *one* observed trace and to find property violations, they actually deal with *causal properties* (e.g., causal datarace, causal atomicity), which are instances of desired system-wise properties that can be detected using only the causal information gathered from the observed trace. Since what can be inferred from a trace intrinsically depends on the chosen causal model, definitions of causal properties differ from technique to technique, with the undesirable effect that a causal property (e.g., a datarace) in one model might not be recognized by another model.

1.1 Motivating Examples

Each example in Figure 1(a) and (b) shows a two-threaded program, together with one of its possible executions, in which Thread 1 is executed completely before Thread 2 starts. In this representation of executions, synchronized blocks are boxed, while write and read operations on shared locations are denoted by

$\leftarrow$ (receiving a value), and $\rightarrow$ (yielding a value), respectively. Both *programs* exhibit a race condition between the two write operations on y. However, are the observed executions also exhibiting a causal datarace?

When analyzing the observed execution in Figure 1(a), a simple happens-before approach ordering all accesses to concurrent objects [22] cannot observe a causal datarace: the release operation of the lock in Thread 1 is required to happen-before the acquire of the lock in Thread 2. Happens-before with lock atomicity [18] is not able to infer a causal datarace either: although the lock atomicity would allow for the two lock-blocks to be permuted, the read of x in Thread 1 is still required to happen-before the write of x in Thread 2. Yet, the race condition can be captured as a causal datarace of the observed execution by weaker happens-before models [23, 25], since in those models, one can additionally permute a write before a read of same location, as long as it is permuted before the write corresponding to that read. Thus, the trace generated by the program in Figure 1(a) *has or does not have* a causal datarace, depending upon the particular causal model employed.

However, we were not able to find any existing (sound) causal model able to detect the race condition in Figure 1(b) as a causal datarace for the observed execution. The reason for this is that all models enforce at least the read-after-write dependency (i.e., a read should always follow the *latest write event* of the same variable), and therefore would not allow the permutation of the last two lock-blocks of the execution, since the read of x in Thread 2 must follow the last write of x in Thread 1. Nevertheless, *there is enough information in the observed execution to be able to detect the race*: since both writes of x in Thread 1 write the same value, it is actually possible to permute the last two lock blocks, and thus detect the race. Moreover, since one could conceive a technique specialized for finding such cases, it can be rightfully claimed that the observed execution *has in fact a causal datarace*, although not captured by any existing definition!

Given this ever increasing (regarding coverage) sequence of causal models and definitions for causal properties, it is only natural to ask the following questions:

> *Is there any causal model that generalizes all existing models, and which cannot be surpassed? Also, is there a unified definition for a causal property, which all present and future causal models can relate to?*

We give positive answers to these questions in the context of sequential consistency [14]. While we believe the presented approach can be applied to other memory models, we chose sequential consistency here for three reasons: (1) it is broadly accepted, popular and intuitive; (2) it is subsumed by other memory models: errors detected under sequential consistency are also errors for other memory models; (3) recent research in computer architecture (e.g., [4]) shows that it actually *can* be efficiently supported and implemented in multiprocessor hardware, strengthening the applicability of our approach.

Contributions. Our first contribution is a novel axiomatization for concurrent computations, based on consistency and feasibility axioms, which yields sound (by

definition) causal models for observed executions. We then show the theoretical significance of a provable sound maximal model, as a means to unify existing definitions of causal properties, but also as a tool for proving soundness of runtime analysis techniques. We exemplify the latter by using our causal model to easily (re)prove the soundness for a series of existing causal models. The main result of the paper is that the obtained model is indeed the maximal causal model for the observed execution, in the sense that it comprises precisely *all* traces which can be generated by all programs which can generate the observed trace. Concretely, we show that: (1) all programs which can produce the observed execution can generate all traces in the model; and (2) for any trace not in the model there exists a program generating the observed trace which cannot generate it.

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 [3, 9, 12, 17, 18, 21–24]. Section 5 discusses the relation between our model and the sound causal models upon which the above mentioned techniques were based [11, 18, 22, 23, 25]. Our axiomatic approach is closest in spirit to that of Netzer and Miller [17], which proposes an axiomatization of a happens-before causal order between memory accesses and semaphore operations. Instead, we directly axiomatize legal sequentially consistent execution traces.

Although our focus here is rather foundational, attempting to unify existing causal models and causal definitions of execution-dependent properties by building a provable maximally sound model to support them, this should not imply that axiomatic approaches to concurrent executions have an exclusively theoretical value. For example, Ganai and Gupta [10] apply a similar technique for softawre model checking, attempting to reduce the state space to be explored using sequential consistency constraints. Moreover, building on a previous draft of this paper [1], Said et al. [20] encoded the axioms of our proposed model (extended with constructs for thread creation and wait/notify) into an SMT solver and used that to effectively search the model for potential dataraces in Java programs.

Another interesting and productive line of research attempts to use information about the actual program code to either statically detect potential bad behaviors [7, 16], or to use information about the program and about the property to be checked to further relax the models of executions [5, 6]. Our approach is complementary to these, establishing a foundation on which code-based techniques can be developed.

Paper structure. Section 2 introduces some notation and discusses sequential consistency. Section 3 axiomatizes consistent concurrent systems and defines our proposed causal models. Section 4 uses these models to give uniform semantic definitions of trace-related properties, such as causal dataraces and atomicity. Section 5 shows how existing models are included in ours, thus proving their soundness. Section 6 presents a constructive characterization of the maximal

model. Section 7 formally defines the maximality claim and proves our model maximal among sound models, and Section 8 concludes.

2 Execution Model

Assume a machine that can execute arbitrarily many threads in parallel. The execution environment contains a set of *concurrent objects* (shared memory locations, locks, ...), which are accessed by threads to share data and synchronize. *Threads*, which can only interact through the execution environment, are abstracted as *sequences of operations on concurrent objects*. The only source of thread non-determinism is the 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, in the same order. To simplify the presentation, we assume no dynamic creation of threads (this presents no technical difficulty).

2.1 Concurrent Objects, Serial Specification

We adopt the Herlihy and Wing [13] definition of concurrent objects and serial specifications. A concurrent object is behaviorally defined through a set of atomic operations, which any thread can perform on it, and a serial specification of its legal behavior in isolation. The serial specification describes the valid sequences of operations which can be performed on the object. We next describe two common types of concurrent objects.

Shared memory locations. Each shared memory location can be regarded as a shared object with read and write operations, whose serial specification states that each read yields the same value as the one of the previous write. Moreover, to avoid non-determinism due to the initial state of the memory, we will further require that all memory locations are initialized, that is, the first operation for each location is a write.

Mutexes. Each mutex can be regarded as a concurrent object providing *acquire* and *release* operations. Their mutual exclusion property is achieved through the serial specification which accepts only those sequences in which the difference between the number of *acquire* and *release* operations is either 0 or 1 for each prefix, and all consecutive pairs of *acquire-release* share the same thread.

To keep the proofs simple and the concepts clear we refrain here from adding more concurrency constructs (such as spawn/join, wait/notify, or semaphores). Note, however, that this would not introduce additional complexity, but just constrain further the notion of consistency.

2.2 Events and Traces

Operations performed by threads on concurrent objects 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, *op*—the operation performed (e.g., *write*, *read*, *acquire*, or *release*), *target*—the concurrent object accessed by the event, and *data*—the value sent/received by the current event, if such exists (e.g., for the *write*/*read* operations). For example, (*thread*=t_1, *op*=*write*, *target*=x, *data*=1) describes an event recording a write operation by thread t_1 to memory location x with value 1. When there is no confusion, we only list the attribute values in an event, e.g., $(t_1, write, x, 1)$. For any event e and attribute *attr*, *attr*(e) denotes the value corresponding to the attribute *attr* in e, and $e[v/attr]$ to denote the event obtained from e by replacing the value of attribute *attr* by v. An *execution trace* is abstracted as a sequence of events. Given a trace τ, a concurrent object o and a thread t, let $\tau\restriction_o$ and $\tau\restriction_t$ denote the restriction of τ to events involving only o, and only t, respectively. Let $latest_o(\tau)$ be the latest event of τ having the *op* attribute o. If o is omitted, it simply means the latest event in τ.

Sequential consistency can be now elegantly defined:

Definition 1 (Attiya and Welch [2]) *Let τ be any trace.*
*(1) τ is **legal** if and only if $\tau\restriction_o$ satisfies o's serial specification for any object o;*
*(2) An **interleaving** of τ is a trace σ such that $\sigma\restriction_t = \tau\restriction_t$ for each thread t.*
*(3) A trace σ is **(sequentially) consistent** if it admits a legal interleaving.*

Since we restrict ourselves to sequential consistency, from here on when we say that a trace is sequentially consistent we automatically mean that it is also legal.

3 Feasibility Model

This section introduces a novel axiomatization for a machine producing consistent executions, and uses it to associate a sound-by-definition causal model to any observed execution, comprising all executions which can potentially be inferred from that execution alone, without additional knowledge of the system generating it.

Figure 2 highlights the two major concepts underlying our approach, namely *consistent traces* and *feasible executions*. A consistent trace (Definition 1) disallows "wrong" behaviors, such as reading a value different from the one which was written, or proceeding when a lock cannot be acquired. Our novel notion, that of feasible executions, refers to *sets* of execution traces and aims at capturing *all* the behaviors that a given system or program can manifest. No matter what task a concurrent system or program accomplishes, its set of 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 closedness*. Second, we assume that thread interleaving is the only source of non-determinism in producing traces; this is captured by our second axiom of feasible traces, *local determinism*.

Each particular multithreaded system or programming environment, say $\mathcal{S}$, has its own notion of feasible execution, given by its specific intended semantics. Let us call all (possibly incomplete) traces that $\mathcal{S}$ can yield $\mathcal{S}$-*feasible*, and let

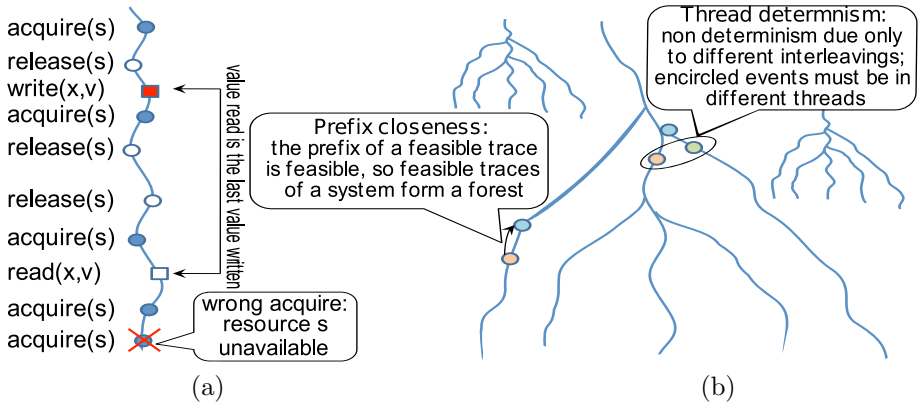

Figure 2: Feasibility model: (a) trace consistency; and (b) feasible executions.

feasible($\mathcal{S}$) be their set. Instead of defining *feasible*($\mathcal{S}$), which requires a formal definition of $\mathcal{S}$ and is therefore $\mathcal{S}$-specific (and tedious), we here *axiomatize* it by what we believe are its crucial properties:

Prefix Closedness: *Events are indivisible and generated in execution order; hence, feasible*($\mathcal{S}$) *must be prefix closed*: if $\tau_1\tau_2$ is $\mathcal{S}$-feasible, then τ_1 is $\mathcal{S}$-feasible.

Local Determinism: *The execution of a concurrent operation is determined by the previous events in the same thread, and can happen at any consistent moment after them.* Formally, if $\tau e, \tau' \in \mathit{feasible}(\mathcal{S})$ and $\tau{\restriction_{\mathit{thread}(e)}} = \tau'{\restriction_{\mathit{thread}(e)}}$ then: if $\tau' e$ is consistent then $\tau' e \in \mathit{feasible}(\mathcal{S})$; moreover, if $\mathit{op}(e) = \mathit{read}$ and there exists an event e' such that $e = e'[\mathit{data}(e)/\mathit{data}]$ and $\tau' e'$ is consistent, then $\tau' e \in \mathit{feasible}(\mathcal{S})$. The second part says that if a *read* operation is enabled, i.e., all previous events have been generated, then it can be executed at any consistent time (despite the fact that the value it receives might be different from that observed in the original trace).

Definition 2 *$\mathcal{S}$ is **consistent** if and only if feasible*($\mathcal{S}$) *satisfies the axioms above.*

A major goal of trace-based analysis is to infer/analyze as many traces as possible using a 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. Let us now define the proposed causal model, termed *feasibility closure*, as the set of executions which can be inferred from an observed execution—they correspond to the traces obtainable from τ using the feasibility axioms.

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

The following result formalizes the soundness of the proposed model, showing how closure properties guarantee that all traces in our causal model are feasible. Moreover, it shows that *any* system/program which can generate one trace, can also generate *all* traces comprised by its causal model.

Proposition 1 *If $\mathcal{S}$ consistent and $\tau \in feasible(\mathcal{S})$ then $feasible(\tau) \subseteq feasible(\mathcal{S})$. Moreover, if σ is consistent and $\tau \in feasible(\sigma)$, then $feasible(\tau) \subseteq feasible(\sigma)$.*

PROOF. Both $feasible(\mathcal{S})$ and $feasible(\sigma)$ are closed under the feasibility axioms. Since τ belongs to both of them, and $feasible(\tau)$ is the smallest set closed under the same axioms, it follows that it must be included in both. □

The intuition for $\tau \in feasible(\sigma)$ is that if a run of any program executed on $\mathcal{S}$ can produce σ, then there is also some run of the same program executed also on $\mathcal{S}$ that can produce τ. Since $feasible(\sigma)$ was chosen to be the smallest set of traces closed under the axioms above, it follows, also intuitively, that if $\tau \notin feasible(\sigma)$ then there is some program that yields σ but which cannot yield τ. This maximality claim will be formalized and proved in Section 7.

Therefore, observing an execution trace τ, $feasible(\tau)$ comprises *all* traces that can be obtained by *all* programs which can produce τ.

As shown later, Proposition 1 can be further strengthened using the characterization of the feasibility closure, in the sense that if τ is a consistent interleaving of σ then $feasible(\sigma) = feasible(\tau)$ (Corollary 1). This basically means that the feasibility closure does not depend on the representative legal trace chosen for an observed sequentially consistent trace σ in Definition 1, and thus it can be rightfully called the *legal feasibility closure* of the sequentially consistent trace σ.

4 Formal Definitions for Causal Properties

In this section we revisit some definitions of causal properties for traces, namely dataraces and atomicity, and claim that the proposed model make those uniform and consistent for models. Indeed, having a maximal model comprising all possible sound models allows for unique semantical definitions which can be shared among all such models.

4.1 Dataraces

Two events have a *data conflict* if they belong to different threads, both access the same memory location, and at least one access is a *write*. In our notation, events e_1 and e_2 have a data conflict if $thread(e_1) \neq thread(e_2)$, $target(e_1) = target(e_2)$, and $write \in \{op(e_1), op(e_2)\}$. A *datarace* occurs when an execution contains two events having a data-conflict, without proper synchronization between them [22]. An *obvious datarace* between events e_1 and e_2 in a consistent trace τ can be observed when the two data-conflicting events (e_1 and e_2) are consecutively generated (i.e., $\tau = \tau_1 e_1 e_2 \tau_2$), so the second part of the definition above is trivially satisfied. However, this definition, although "model independent", is rather restrictive, since the chances of noticing the two accesses occurring consecutively

are really low. For this reason, the notion of *causal datarace* is more appropriate. Informally, an execution admits a causal datarace between two memory accesses if the two accesses could have been executed concurrently under an alternative scheduling, inferable from the observed execution.

As previously discussed in Section 1.1, many techniques have been proposed for finding causal dataraces. However, the formal definition of a datarace for an execution in such a model is typically operational, that is, constrained by the model itself. For example, in the techniques based on happens-before, the datarace is defined as two events having a data conflict which are not ordered by the happens-before causal order induced by the observed execution [22]; if considering happens before with locksets [18], the happens-before ordering is relaxed to only order memory location accesses, with the additional requirement that the lock-protected blocks be maintained atomic. Therefore, each causal model encountered in the literature defines its own model-dependent definition for a causal datarace, to take full advantage of its particularities.

Since, as shown in Section 7, our causal model is maximal, we can precisely give a definition of causal datarace which only depends on the observed execution:

Definition 4 *A trace $\tau = \tau_1 e_1 \tau_2 e_2$ admits a **causal datarace** on data-conflicting events e_1 and e_2 if and only if there exists a τ-feasible trace σ such that $\sigma\restriction_{thread(e_1)} = \tau_1\restriction_{thread(e_1)}$ and $\sigma\restriction_{thread(e_2)} = (\tau_1 e_1 \tau_2)\restriction_{thread(e_2)}$.*

The above definition states that we can predict a datarace from an observed trace τ if there exists a τ-feasible trace which makes the datarace apparent. We chose as a witness a trace stopped at the moment when both threads are about to execute the events in a race. This is indeed a clear witness for the race, since, by the local determinism axiom, the execution of the conflicting operations is allowed to proceed in any order from this point; moreover this saves us the trouble of specifying that the value of an event involved in the trace might change if it corresponds to a *read* operation. Nevertheless, one should note that, unlike in other causal models, the witness traces containing the events in an observable race, in both orderings, are also part of the feasibility closure of τ.

With our causal datarace definition, the race in Figure 1(b) is finally captured, having $(1, acquire, l)$ $(1, write, x, 1)$ $(1, release, l)$ $(2, acquire, l)$ $(2, read, x, 1)$ as a witness belonging to the feasibility closure of the observed execution.

4.2 Atomicity

Assume the existence of an additional concurrent object, named *transaction monitor*, with two operations *begin* and *end* and the serial specification requiring that for each thread the first transaction monitor operation is a *begin* and, for each thread, there are no two transaction *begin* operations without a transaction *end* between them. That is, transaction monitors are similar to but weaker than locks, in the sense that the mutual exclusion is not enforced, although desired. A *transaction* of a consistent trace τ is then a subsequence of events σ of τ having the same thread, starting with a transaction *begin* operation and ending with the next transaction *end* operation.

Within this framework, one can either define global atomicity, which amounts to serializability of transactions [25], or local atomicity [8], which requires each transaction be serializable, but not necessarily the entire execution.

Definition 5 *A transaction σ of τ is* ***atomic*** *for consistent trace τ if there exists a τ-feasible trace $\tau_1\sigma\tau_2$. τ is* ***locally atomic*** *if each of its transactions are atomic for τ. τ is* ***(globally) atomic****, or* ***serializable****, if there exists a τ-feasible trace τ' such that each transaction σ of τ is a contiguous subsequence of τ'.*

Although these definitions are similar to those found in the above mentioned papers, they are now model-independent. Being defined using the maximal causal model, they become universal, applicable to all conceivable sound causal models.

5 Relationship with Existing Models

In this section we analyze the relationships between our model and other existing (sound) models for (consistent) multithreaded computations. Proving that existing models are faithfully captured by our model shows that these rather ad hoc (from a theoretical perspective) models are indeed sound. We start with the following result, which can be regarded as a sufficient criterion for feasibility:

Theorem 1 *Any consistent prefix σ_1 of an interleaving $\sigma_1\sigma_2$ of τ is τ-feasible.*

PROOF. Induction on the length of the interleaving prefix. The base case is trivial. Let $\tau' e$ be a consistent interleaving prefix of τ, and assume that τ' is τ-feasible. Let $t = thread(e)$, and let τ_1, τ_2 be such that $\tau = \tau_1 e \tau_2$. By prefix closedness, it follows that $\tau_1 e$ is feasible. Moreover, since $(\tau' e)\restriction_t = \tau'\restriction_t e$ is a prefix of $\tau\restriction_t$, it follows that $\tau'\restriction_t = \tau_1\restriction_t$. Using the local determinism for $\tau_1 e$ and τ', we obtain that $\tau' e$ is τ-feasible (since it is consistent). □

The remainder of this section shows that existing sound causal models are captured by the feasibility closure as simple instances of Theorem 1.

5.1 Happens Before Relation on Mazurkiewicz Traces

One elegant way to capture the happens-before trace equivalence is the Mazurkiewicz trace associated to the dependence given by the happens-before relation [11].

The happens-before dependence is a set $T \cup D$, where $T = \bigcup_t \{(e_1, e_2) : \tau\restriction_t = \tau_1 e_1 e_2 \tau_2\}$ is the intra-thread sequential dependence relation and $D = \bigcup_x \{(e_1, e_2) : \tau\restriction_x = \tau_1 e_1 e_2 \tau_2$ *such that* e_1 *or* e_2 *is a write of* $x\}$ is the sequential memory dependence relation. 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 [15]: 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 shows that the feasibility closure is closed under the equivalence relation generated by happens-before, that is, happens-before is captured by our model, and thus re-shown sound for consistent executions:

Proposition 2 *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 τ-feasible. Given any τ-feasible trace τ', $[\tau'] \subseteq$ feasible(τ). Hence, $[\tau] \subseteq$ feasible(τ).*

PROOF. Let $\tau_1 e_1 e_2 \tau_2$ be a τ-feasible trace such that $(e_1, e_2) \notin T \cup D$. We will show that $\tau_1 e_2 e_1 \tau_2$ is also τ-feasible. First, all prefixes of $\tau_1 e_1 e_2 \tau_2$, including τ_1, $\tau_1 e_1$, $\tau_1 e_1 e_2$, $\tau_1 e_1 e_2 \tau_2' e_2'$ (for any prefix $\tau_2' e_2'$ of τ_2), are τ-feasible, since *feasible*(τ) is prefix closed. Now, we can iteratively use closedness under local determinism (1) for $\tau_1 e_1 e_2$ and τ_1, to derive that $\tau_1 e_2$ is τ-feasible; (2) for $\tau_1 e_1$ and $\tau_1 e_2$ to derive that $\tau_1 e_2 e_1$ is also τ-feasible; (3) by finitary induction for each prefix $\tau_2' e_2'$ of τ_2, for $\tau_1 e_1 e_2 \tau_2' e_2'$ and $\tau_1 e_2 e_1 \tau_2'$ to derive that $\tau_1 e_2 e_1 \tau_2' e_2'$ is also τ-feasible.

Therefore, for any τ-feasible trace τ', *feasible*(τ') is closed under permutation of consecutive independent events; hence, $[\tau'] \subseteq$ *feasible*$(\tau') \subseteq$ *feasible*(τ). □

5.2 Weak Happens Before

Several more recent trace analysis techniques [23, 25] argue that the happens-before model can be further relaxed, noticing that the only purpose of the write-after-read happens-before order is to guarantee that a read event always reads the same write event as before in any feasible interleaving of the original trace. Therefore, one only needs to preserve the *read-after-write dependence*:

Definition 6 *Suppose $\tau = \tau_1 e_1 \tau_2 e_2 \tau_3$. Then e_2* ***write-read depends on*** *e_1 in τ, written $e_1 <_\tau^{wr} e_2$, if target(e_1) = target(e_2), op(e_1) = write, op(e_2) = read, and for all $e \in \mathcal{E}_{\tau_2}$, either target$(e) \neq$ target(e_1), or op$(e) \neq$ write.*

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

Sen et al. [23] introduce the notion of *atomic* sets associated to each *write* event, containing itself and all read events which write-read depend on it, accepting as feasible executions all linearizations of the transitive closure of the combined $<_\tau^{wr}$ and thread ordering, satisfying the additional requirement that the atomic sets are preserved. However, as noticed by Wang and Stoller [25], this can be simply restated as follows:

Definition 7 *$\tau \sim \sigma$ if τ is an interleaving of σ and $<_\tau^{wr} = <_\sigma^{wr}$.*

That is, the $\sim$-equivalence class of τ contains all interleavings of τ which have exactly the same write-read dependence relation. Next result shows that this model is also captured by our model.

Proposition 3 *If σ_1 is τ-feasible, and $\sigma_1 \sim \sigma_2$, then σ_2 is also τ-feasible.*

PROOF. We show that we are in the conditions of Theorem 1: Since σ_1 is consistent, and $<_{\sigma_2}^{wr} = <_{\sigma_1}^{wr}$, it follows that σ_2 must also be consistent, since all *read* events follow the same *write* events as in the σ_1, which, by the consistency of σ_1, precisely implies that each *read* event returns the value of the previous *write* event. □

5.3 Happens-Before with synchronization

A conservative, sound, and requiring no implementation changes approach to handling locks in happens-before-based trace analysis techniques is to assume that *acquire* and *release* operations on the same lock yield the same happens-before dependence as if they were particular *write* and *read* operations (on the lock variable) [22]. However, this prevents synchronized blocks from being permuted, and thus imposes coverage limitations. The lock-set approaches, also called hybrid happens-before [18], propose to handle locks separately, associating to each event the set of locks [21] protecting them, hereby not enforcing any particular order between synchronized blocks.

We here group the events protected by locks in *atomic blocks*. Events e_1 and e_2 from a consistent trace τ, both generated by thread t, are *l-atomic* in τ, written $e_1 \Updownarrow_l^\tau e_2$, if and only if there is some *acquire* event e on lock l generated by t before both e_1 and e_2, and there is no *release* event e' on l generated by t between e and either of e_1, e_2. For each lock l, let $[e]_l$ denote the *l-atomic equivalence class of e*. A trace τ' is *consistent with the lock atomicity of τ* if there exists no lock l and decomposition $\tau_1 e_1 \tau_2 e_2 \tau_3 e_3 \tau_4 e_4 \tau_5$ such that $e_1 \Updownarrow_l^\tau e_3$ and $e_2 \Updownarrow_l^\tau e_4$ and $[e_1]_l \neq [e_2]_l$. Let $\prec_{hb}^\tau$ be the transitive closure of the union between happens-before and thread orderings of τ. The following holds:

Proposition 4 *Let σ be a τ-feasible trace. Any linearization of $\prec_{hb}^\sigma$ consistent with the lock atomicity of σ is τ-feasible.*

PROOF. Again, we reduce our proof to Theorem 1. First, any linearization of $\prec_{hb}^\sigma$ is an interleaving of σ. Moreover, since σ is consistent, preservation of happens-before ensures that the serial specification of the memory locations is satisfied. Finally, consistency with lock atomicity implies that the serial specification for mutexes is also satisfied. Therefore, any linearization of $\prec_{hb}^\sigma$ consistent with the lock atomicity of σ, is a consistent interleaving of σ, thus σ-feasible. □

5.4 Weak-Happens-Before with synchronization

We next present two approaches to handling synchronization in weak-happens-before models and show they are both embeddable in our model.

Lock atomicity via write-read atomicity [23]. Since the notion of write-read atomicity already allows atomic sets to be permuted, it seems reasonable to use the conservative idea from standard happens-before methods, and treat *acquire* as a *write* event and *release* as a read event. Formally, given the consistent trace τ, one could additionally introduce an atomic dependence relation $<_\tau^a$ given by $e_1 <_\tau^a e_2$ if $\tau = \tau_1 e_2 \tau_2 e_2 \tau_3$, $\mathit{target}(e_1) = \mathit{target}(e_2)$, $\mathit{op}(e_1) = \mathit{acquire}$, $\mathit{op}(e_2) = \mathit{release}$, and there is no event e in τ_2 such that $\mathit{target}(e) = \mathit{target}(e_1)$, and $\mathit{op}(e) = \mathit{acquire}$. With this definition, equivalent traces to an observed trace τ are those interleavings of τ having the same write-read and atomic dependencies.

However, this definition needs a careful approach. Consider the example in Figure 1(b), and suppose that we observe a similar execution, but that the program is stopped after the *read* of x in Thread 2. Since no *release* event has been generated, the *acquire* in Thread 2 has no event depending on it, and thus it can be permuted (without the *read* event on x it was supposed to protect) before the last lock-block of Thread 1. Then, the final *read* of x itself can be permuted past the final *release* of l in Thread 1, exhibiting a spurious causal datarace.

Nevertheless, these models are sound for *synchronization complete* traces, that is, traces in which each acquired lock is eventually released.

Proposition 5 *Let σ be a synchronization complete τ-feasible trace. Any interleaving σ' of σ satisfying that $<^{wr}_{\sigma'}=<^{wr}_{\sigma}$ and $<^{a}_{\sigma'}=<^{a}_{\sigma}$ is τ-feasible.*

PROOF. Since we already shown that $<^{wr}_{sigma'}=<^{wr}_{\sigma}$ implies that the serial specification of memory locations is verified, we only need to show that $<^{a}_{\sigma'}=<^{a}_{\sigma}$ implies the satisfaction of the mutex specification for synchronization complete traces, that is, that any prefix of σ' has at most one more *acquire* operations than *release* operations, and all consecutive pairs of *acquire-release* have the same thread. The second part is easily guaranteed by the fact that $<^{a}_{\sigma'}=<^{a}_{\sigma}$, since $<^{a}$ enforces the *acquire-release* in relation are consecutive, and, since σ is consistent, this definition additionally implies that they have the same thread. The first part comes from the fact that, since σ is synchronization complete, every *acquire* has a corresponding *release*, with whom it is in the $<^{a}_{\sigma}$ relation. □

Lock atomicity via locksets. Wang and Stoller [25] propose a weak-happens-before model based on write-read dependence, while using locksets to handle locks as individual objects. In this model, a trace τ' is equivalent with a consistent trace τ if τ' is an interleaving of τ having the same write-read dependence relation and being consistent with the lock atomicity of τ.

Proposition 6 *Let σ be a τ-feasible trace. Any interleaving σ' of σ, consistent with the lock atomicity of σ and satisfying that $<^{wr}_{\sigma'}=<^{wr}_{\sigma}$ is τ-feasible.*

PROOF. From the proof of Proposition 3, $<^{wr}_{\sigma'}=<^{wr}_{\sigma}$ implies the serial specification of memory locations is obeyed in σ'. Additionally, from the proof of Proposition 4, consistency with the lock atomicity of a consistent trace implies that the serial specification of mutexes is obeyed. We can therefore apply Theorem 1. □

6 Characterizing the Feasibility Closure

Section 3 showed how our causal model can be naturally defined by characterizing feasibility axiomatically rather than constructively. Closure axioms guarantee that all equivalent traces which can be derived based on the consistency axioms are considered. This section presents a constructive characterization of the feasibility closure.

As might have been suggested by Theorem 1, consistent interleaving prefixes cover all possibilities of generating τ-feasible traces using only the events in τ. However, the definition of interleaving (prefix) overlooks the final part of the local determinism axiom, that is, the one regarding operations which might receive different values from their objects. To achieve a complete constructive characterization of the feasibility closure, we have to go beyond prefixes of interleavings, more exactly, one *read* operation per thread beyond. This is because, as guaranteed by local determinism, whenever all events before an event have been generated in a thread, the operation on the concurrent object specified in that event can also take place, but its *data* attribute might now retrieve a value different from the one it had in the observed trace. However, once such an event whose *data* is different from the one in the original trace is derived, the execution cannot be continued for that thread, because that event might influence/prevent the generation of the following events. An *extended interleaving prefix* is a (partial) trace which behaves similarly to the observed trace up to its final event for each thread, which might have a different value:

Definition 8 *Trace $\tau' = \tau_1 e'$ is an **extended prefix** of $\tau = \tau_1 e \tau_2$ if either $e = e'$, or $op(e) = read$ and $e = e'[data(e)/data]$. τ' is an **extended interleaving prefix** of τ if $\tau'\restriction_t$ is an extended prefix of $\tau\restriction_t$ for any thread t.*

We can now give a complete constructive characterization for τ-feasible traces:

Theorem 2 *Given τ consistent, a trace τ' is τ-feasible iff it is a consistent extended interleaving prefix of τ.*

PROOF. Proving that any consistent extended interleaving prefix of τ is τ-feasible proceeds similarly to the proof of Theorem 1. For the reverse, one needs to show that the set of consistent extended interleaving prefixes of τ contains τ, is prefix closed, and closed under local determinism. First two are obvious: τ is an interleaving prefix of itself, and any prefix of an extended interleaving prefix of τ is an extended interleaving prefix of τ by the definition. Now let $\tau_1 e$ and τ_2 be consistent interleaving prefixes of τ such that $thread(e) = t$, and $\tau_1\restriction_t = \tau_2\restriction_t$. Since $(\tau_1 e)\restriction_t$ is an extended prefix of $\tau\restriction_t$, then either $(\tau_1 e)\restriction_t$ is a prefix of τ, or $op(e) = read$, and there exists e', such that $thread(e') = n$, $op(e') = read$, $target(e') = target(e)$, and $\tau_1\restriction_t e'$ is a prefix of $\tau\restriction_t$.

Let e'' be e, if $op(e) \neq read$, or $e'' = e[data(e^w)/data]$, if $e^w = latest_{write}(\tau_2\restriction_{target(e)})$. Then $\tau_2 e''$ is an extended interleaving prefix. If $op(e) \neq read$, and $\tau_2 e$ is consistent, then it also is a consistent extended interleaving prefix. If $op(e) = read$, then, since $\tau_2 e''$ is consistent (by the choice of e''), the property follows. □

An important corollary of Theorem 2 is that the feasibility closure does not depend on the legal trace chosen to represent a sequentially consistent trace, as it contains all the representative legal traces.

Corollary 1 *If τ is a consistent interleaving of consistent trace σ then $feasible(\tau) = feasible(\sigma)$.*

Input: Trace τ_0 of size n over k threads.
Maps: thread : $\{1,\ldots,\mathsf{k}\} \to \mathit{Stack}$
$\sigma : \mathit{Locations} \to \mathit{Int}$
Initial: $\sigma[x] \leftarrow \perp$, for all variables and mutexes
thread$[t] \leftarrow \tau_0{\restriction_t}$, for all threads
Advanceable $\leftarrow \{1,\ldots,\mathsf{k}\}$

```
1  τ ← ϵ; t ← 0;
2  while t < k do
3  |  t ← t + 1;
4  |  if t ∈ Advanceable then
5  |  |  e ← top(thread[t]);
6  |  |  if (op(e) ≠ acquire ∨ σ[target(e)] = ⊥) ∧ (op(e) ≠ read ∨
   |  |  σ[target(e)] ≠ ⊥) then                                  // advance
7  |  |  |  l ← target(e);
8  |  |  |  if op(e) = read ∧ data(e) ≠ σ[l] then        // extended prefix
9  |  |  |  |  Advanceable ← Advanceable \ {t} ;
10 |  |  |  |  data(e) ← σ[l];
11 |  |  |  else                                          // update state
12 |  |  |  |  pop(thread[t]);
13 |  |  |  |  if op(e) = write then σ[l] ← data(e);
14 |  |  |  |  if op(e) = acquire then σ[l] ← σ[l] − 1;
15 |  |  |  |  if op(e) = release then σ[l] ← σ[l] + 1;
16 |  |  |  end
17 |  |  |  push(τ,e); check τ against φ;
18 |  |  |  t ← 0;
19 |  |  end
20 |  end
21 |  while t = k ∧ τ ≠ ϵ do                                  // backtrack
22 |  |  e ← pop(τ); t ← thread(e); l ← target(e);
23 |  |  if t ∉ Advanceable then                       // extended prefix
24 |  |  |  Advanceable ← Advanceable ∪ {t};
25 |  |  else                                            // restore state
26 |  |  |  push(thread[t],e);
27 |  |  |  if op(e) = write then σ[l] ← data(latest(τ↾l));
28 |  |  |  if op(e) = acquire then σ[l] ← t;
29 |  |  |  if op(e) = release then σ[l] ← ⊥;
30 |  |  end
31 |  end
32 end
```

Algorithm 1: Model checking the feasibility closure

Model checking the feasibility closure. 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, 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 $\perp$ for both variables and mutexes. We additionally maintain a set of enabled threads (Advanceable), 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 created, τ, is also maintained as a stack, but with first events at bottom of the stack; it is initially empty. Variable t keeps track of the index of the thread which should be advanced next. The main loop is a backtracking loop, exiting only when the entire space has been explored. Inside the loop, the first part (lines 3–6) checks whether the next thread can be advanced. If a thread is found, the state is modified accordingly (lines 12–15), disabling further advances to the thread if the state of the added event differs from the one in the observed trace (lines 8–10); 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 checked (line 17), and the search for the next advance-able thread is restarted (line 18). If no additional thread can be advanced from this state, the algorithm backtracks, undoing the effects of previous advances (lines 21–31).

A simple amortized analysis of Algorithm 1 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 prefix-closed.

Happens-before based model checkers can exploit the property being checked or the structure of the program to gain efficiency (but not coverage). We envision similar techniques could potentially be applied to our models. However, here we are not trying to propose an *optimal* model checker but rather to show that the maximum model is algorithmically analyzable, not just an existential entity, and it can be the basis on which other analysis techniques can be built.

7 Maximality

Proposition 1 showed that our causal model is sound in the sense that the same program which generated a trace can also generate its entire feasibility closure. Section 5 further showed that the causal model is quite comprehensive, being able to capture the existing sound causal models.

In this section we show that the model is also maximal among sound models,

$Proc$	$::=$	$Proc \mid\mid Proc$	$\frac{\langle p_1, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_1', \sigma', \delta', \rho'\rangle}{\langle p_1 \mid\mid p_2, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_1' \mid\mid p_2, \sigma', \delta', \rho'\rangle}\ (Par_1)$ $\quad \frac{\langle p_2, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_2', \sigma', \delta', \rho'\rangle}{\langle p_1 \mid\mid p_2, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_1 \mid\mid p_2', \sigma', \delta$
	$\mid$	$Int : Stmt$	$\frac{\langle s, \sigma, \delta, \rho, t\rangle \xrightarrow{\tau} \langle s', \sigma', \delta', \rho', t\rangle}{\langle t : s, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle t : s', \sigma', \delta', \rho'\rangle}$
$Stmt$	$::=$	$Stmt\ ;\ Stmt$	$\frac{\langle s_1, \sigma', \delta', \rho', t\rangle \xrightarrow{\tau} \langle s_1', \sigma', \delta', \rho', t\rangle}{\langle s_1\ ;\ s_2, \sigma, \delta, \rho, t\rangle \xrightarrow{\tau} \langle s_1'\ ;\ s_2, \sigma', \delta', \rho', t\rangle}$
	$\mid$	nop	$\frac{\cdot}{\langle \mathsf{nop}\ ;\ s, \sigma, \delta, \rho, t\rangle \xrightarrow{\epsilon} \langle s, \sigma, \delta, \rho, t\rangle}$
	$\mid$	if Int then $Stmt$	$\frac{\cdot}{\langle \mathsf{if}\ i\ \mathsf{then}\ s, \sigma, \delta, \rho, t\rangle \xrightarrow{\epsilon} \langle s, \sigma, \delta, \rho, t\rangle}$ **if** $\rho($
			$\frac{\cdot}{\langle \mathsf{if}\ i\ \mathsf{then}\ s, \sigma, \delta, \rho, t\rangle \xrightarrow{\epsilon} \langle \mathsf{nop}, \sigma, \delta, \rho, t\rangle}$ **if** $\rho($
	$\mid$	load Loc	$\frac{\cdot}{\langle \mathsf{load}\ x, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, read, x, i)} \langle \mathsf{nop}, \sigma, \delta, \rho[t \leftarrow i], t\rangle}$ **where** $i =$
	$\mid$	$Loc := Int$	$\frac{\cdot}{\langle x := i, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, write, x, i)} \langle \mathsf{nop}, \sigma[x \leftarrow i], \delta, \rho, t\rangle}$
	$\mid$	acquire Loc	$\frac{\cdot}{\langle \mathsf{acquire}\ x, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, acquire, x)} \langle \mathsf{nop}, \sigma, \delta[x \leftarrow t], \rho, t\rangle}$ **if** $\delta($
	$\mid$	release Loc	$\frac{\cdot}{\langle \mathsf{release}\ x, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, release, x)} \langle \mathsf{nop}, \sigma, \delta[x \leftarrow \bot], \rho, t\rangle}$ **if** $\delta($

Figure 3: Syntax and SOS semantics for the CONC language

in the sense that any extension to it is done at the expense of soundness. We will prove therefore that given a trace τ' which is not in the feasibility closure of a trace τ, there exists a program p which can generate τ but not τ'; therefore, if the model were extended to include τ' and used τ' as a witness that a property is satisfied/invalidated by a program generating τ, this would be a false witness if the program which generated τ was p.

To prove our claim, we propose CONC, a very simple (not even Turing complete) concurrent language which can conceivably be simulated in any real language. Figure 3 presents the grammar and SOS semantics of CONC. The grammar specifies a parallel composition of named threads. Each thread is a succession of statements and uses one internal register to load data from the shared memory. load x loads the value at location x into the internal register of the thread, $x := i$ stores integer i at location x, acquire and release have the straight-forward semantics, and if i then s executes s only if the internal register has value i. A running configuration of CONC is a tuple $\langle p, \sigma, \delta, \rho\rangle$ where p is the remainder of the program being executed, σ maps variables to values, δ maps each lock to the id of the thread holding it, and ρ gives for each thread the value of its internal register. Assuming p has n threads, the initial configuration of the system is $START(p) = \langle p, \sigma_\epsilon, \delta_\epsilon, \rho_\epsilon\rangle$ where σ_ϵ, δ_ϵ, and ρ_ϵ^n, initialize all locations, locks, and registers for the n threads with $\perp$, respectively.

We have chosen this minimal language both because it is sufficiently expressive to generate all (finite) legal traces and because it is quite easy to simulate in any other language. In Java, for example, each thread would be modeled by a thread object, and all threads could be started in a loop by the main thread. Since beginning of threads do not generate events, this is as-if all threads start together in parallel. The running method of each Java thread object would declare a local variable r to stand for the register, and then the two CONC instructions dealing with the register translate as follows: load l becomes $r = l$, and if i then s becomes if $(r == i)$ s.

It is straightforward to associate to each event an instruction producing it. Let *code* be the mapping defined on events as follows:

$$code(e) = \begin{cases} \mathsf{load}\ x & \textbf{if } e = (t, read, x, i) \\ x := i & \textbf{if } e = (t, write, x, i) \\ \mathsf{acquire}\ x & \textbf{if } e = (t, acquire, x) \\ \mathsf{release}\ x & \textbf{if } e = (t, release, x) \end{cases}$$

Given a program p, let $p\restriction_t$ be its projection on thread t, that is, the statement labeled by t in the parallel composition.

The following result shows that, except for the code, the running configuration is completely determined by the trace generated up to that point, basically allowing us to work just with code and traces in the sequel.

Proposition 7 (τ-configurations) *Suppose that n is the number of threads of p and*

$$\text{CONC} \vdash START(p) \xrightarrow{\tau}{}^{*} \langle p', \sigma_\tau, \delta_\tau, \rho_\tau^n\rangle.$$

Then (1) $\sigma_\tau(x) = data(latest_{write}(\tau\restriction_x))$;
(2) $\delta_\tau(x) = \begin{cases} thread(latest(\tau\restriction_x)) \textbf{ if } op(latest(\tau\restriction_x)) = acquire \\ \bot \qquad\qquad\qquad\qquad\qquad\qquad \textbf{otherwise} \end{cases}$;
and (3) $\rho^n_\tau(t) = data(latest_{read}(\tau\restriction_t))$.

PROOF. First note that ϵ-transitions only affect the code-part of configurations. Therefore, the base case, when $\tau = \epsilon$, is trivial. Suppose now that

$$START(p) \xrightarrow{\tau}{}^{*} \langle p', \sigma_\tau, \delta_\tau, \rho^n_\tau\rangle \xrightarrow{e} \langle p'', \sigma, \delta, \rho\rangle.$$

Suppose that $thread(e) = t_i$. Then in order for e to be generated, $p'\restriction_{t_i}$ must be $code(e)$; p''' and $p'' = \mathsf{nop}$; p'''.

The proof tree is built by applying in order $i-1$ steps of Par_2 and then a step of (Par_1) (or none, if $i = n$), then (*Thread*), (*Seq*), and finally one of the (*Read*), (*Write*), (*Acq*), or (*Rel*); we therefore need to show that

$$\langle code(e), \sigma_\tau, \delta_\tau, \rho^n_\tau, t_i\rangle \xrightarrow{e} \langle \mathsf{nop}, \sigma_{\tau e}, \delta_{\tau e}, \rho^n_{\tau e}, t_i\rangle$$

If $e = (t_i, read, x, i)$, then $code(e) = \mathsf{load}\ x$ so we can apply the (*Read*) rule, which updates only ρ^n_τ to $\rho^n_\tau[r \leftarrow i] = \rho^n_{\tau e}$.

If $e = (t_i, write, x, i)$, then $code(e) = x := i$, and rule *Write* applies updating only σ to $\sigma_\tau[x \leftarrow i] = \sigma_{\tau e}$.

If $e = (t_i, acquire, x)$, then $code(e) = \mathsf{acquire}\ x$ and only δ is updated to $\delta_\tau[x \leftarrow t] = \delta_{\tau e}$.

If $e = (t_i, release, x)$, then $code(e) = \mathsf{release}\ x$ and only δ is updated to $\delta_\tau[x \rightarrow \bot] = \delta_{\tau e}$. □

Therefore, in the sequel we will use $\text{CONC} \vdash p \xrightarrow{\tau}{}^{*} p'$ instead of $\text{CONC} \vdash START(p) \xrightarrow{\tau}{}^{*} \langle p', \sigma_\tau, \delta_\tau, \rho^n_\tau\rangle$

Now, let us prove that the semantics of CONC does indeed satisfy the sequential consistency axioms. Let p be a CONC program and let $feasible(p)$ be the set of all p-feasible traces; that is τ is p-feasible if there exists a program p' such that $\text{CONC} \vdash p \xrightarrow{\tau}{}^{*} p'$. We sill show that $feasible(p)$ satisfies the *strong local determinism* property, namely, not only that an enabled event can be generated at any point by its thread, but that it also must be the next event generated by that thread (ignoring the *data* attribute for *read* events). Formally, $feasible(p)$ satisfies strong local determinism if it satisfies local determinism and if $\tau_1 e_1$ and $\tau_2 e_2$ are p-feasible, $thread(e_1) = thread(e_2) = t$, and $\tau_1\restriction_t = \tau_2\restriction_t$, then $op(e_1) = op(e_2)$, $target(e_1) = target(e_2)$; if additionally $op(e_i) = write$, then also $data(e_1) = data(e_2)$.

The *restriction of a derivation* $\text{CONC} \vdash p_0 \xrightarrow{\tau_1} p_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} p_n$ *to thread* t is the maximal subsequence of statements $p_0\restriction_t, p_1\restriction_t, \ldots, p_n\restriction_t$ such that any two consecutive statements in the sequence are distinct, and we'll denote it by $\left[\text{CONC} \vdash p_0 \xrightarrow{\tau_1\cdots\tau_n}{}^{*} p_n\right]\restriction_t$.

The following result shows that every CONC program p is a consistent system in the sense of Definition 2.

Proposition 8 (CONC consistency) *feasible(p) satisfies prefix closedness and strong local determinism.*

PROOF. Prefix closedness is obvious, since the semantics can emit at most one event for each execution step.

Now, notice that every transition in CONC modifies the code for precisely one of the threads. Let us prove the following stronger local result which basically shows that the evolution of a thread does not depend on events which are not directly related to it.

> Suppose $\text{CONC} \vdash p \xrightarrow{\tau_1}{}^{*} p_1 \xrightarrow{\tau_1'} p_1'$ (where τ_1' is either ϵ or an event) and $\text{CONC} \vdash p \xrightarrow{\tau_2}{}^{*} p_2$, such that $\tau_1{\restriction_t} = \tau_2{\restriction_t}$, $p_1{\restriction_t} = p_2{\restriction_t}$, $p_1'{\restriction_t} \neq p_1{\restriction_t}$, and either $\tau_2\tau_1'$ is consistent or τ_1' is a *read* event. Then there exist an unique p_2' such that $p_2'{\restriction_t} \neq p_2{\restriction_t}$ and $\text{CONC} \vdash p \xrightarrow{\tau_2}{}^{*} p_2 \xrightarrow{\tau_2'} p_2'$, and, moreover $p_1'{\restriction_t} = p_2'{\restriction_t}$, and either $\tau_1' = \tau_2'$ or both of them are reading from the same target but with different values.

Note that uniqueness holds trivially, as once a thread was chosen, at most one rule can apply. Now, for the existence part, if the transition for τ_1' is

(Nop), then it can be applied on p_2 with the same effect;

(If_{true}) or (If_{false}), then since $\rho_\tau^n(t) = data(latest_{read}(\tau{\restriction_t}))$ and $\tau_1{\restriction_t} = \tau_2{\restriction_t}$, it follows that $\rho_{\tau_1}^n(t) = \rho_{\tau_2}^n(t)$, therefore the exact same transition can be applied on p_2;

($Read$), then $\tau_1' = (t, read, x, i)$, which must be generated by an instruction $\mathsf{load}\ x$, whence the same rule can be applied on p_2, generating event $(t, read, x, i')$, where $i' = \sigma_{\tau_2}(x)$.

($Write$), then $\tau_1' = (t, write, x, i)$, which must be generated by an instruction $x := i$, whence and same rule can be applied on p_2, generating the same event;

(Acq), then $\tau_1' = (t, acquire, x)$, which must be generated by $\mathsf{acquire}\ x$; since $\tau_2\tau_1'$ is consistent, it must be that $\delta_{\tau_2}(x) = \bot$, hence the rule can be applied on p_2 generating the same event;

(Rel), then $\tau_1' = (t, release, x)$, which must be generated by $\mathsf{release}\ x$; since $\tau_2\tau_1'$ is consistent, it must be that $\delta_{\tau_2}(x) = t$, hence the rule can be applied on p_2 generating the same event.

Let us now use the above lemma to prove the local determinism property. Assume $\tau_1 e$ and τ_2 are p-feasible traces such that $thread(e) = t$ and $\tau_1{\restriction_t} = \tau_2{\restriction_t}$. Let p_1, p_2 be such that $\text{CONC} \vdash p \xrightarrow{\tau_1 e} p_1$ and $\text{CONC} \vdash p \xrightarrow{\tau_2} p_2$. Then we can use the lemma proved above to show that $\left[\text{CONC} \vdash p \xrightarrow{\tau_2}{}^{*} p_2\right]{\restriction_t}$ is a prefix of $\left[\text{CONC} \vdash p \xrightarrow{\tau_1 e}{}^{*} p_2\right]{\restriction_t}$, by induction on the length of $\left[\text{CONC} \vdash p \xrightarrow{\tau_2}{}^{*} p_2\right]{\restriction_t}$.

Moreover, using the same lemma, we can (uniquely) continue the execution of p_2 on thread t using the same steps from the execution of p_1 and the fact that either $\tau_2 e$ is consistent or e is a read event.

For strong local determinism, if $\tau_1 e_1$ and $\tau_2 e_2$ are p-feasible, $\mathit{thread}(e_1) = \mathit{thread}(e_2) = t$, and $\tau_1{\restriction_t} = \tau_2{\restriction_t}$, then we can use an argument similar to the one above to match the statements corresponding to thread t from the beginning of the execution and until e_1 and e_2 are generated, and for this step, applying the part of the lemma referring to the generated events. □

Now, given a trace τ, let us build the canonical CONC program generating it. *code* can be naturally extended on traces by $\mathit{code}(e\tau) = \mathit{code}(e)$; $\mathit{code}(\tau)$. Let $\{t_1, t_2, \ldots, t_n\}$ be the set of thread ids appearing in τ. Then the program $\mathit{program}(\tau)$ associated to trace τ is defined by

$$\mathit{program}(\tau) = t_1 : \mathit{code}(\tau{\restriction_{t_1}}) \mid\mid \cdots \mid\mid t_n : \mathit{code}(\tau{\restriction_{t_n}})$$

Let us also define the empty program with n threads as $\mathit{program}^n(\epsilon) = t_1 : \mathsf{nop} \mid\mid \cdots \mid\mid t_n : \mathsf{nop}$. The following result shows that the program corresponding to a consistent trace can indeed generate that trace.

Proposition 9 *If τ is a consistent trace with n threads, then* $\mathrm{CONC} \vdash \mathit{program}(\tau) \xrightarrow{\tau}{}^{*} \mathit{program}^n(\epsilon)$.

PROOF. We prove by induction on the length of τ that $\mathrm{CONC} \vdash \mathit{program}(\tau\tau') \xrightarrow{\tau}{}^{*} \mathit{program}^n(\tau')$ where

$$\mathit{program}^n(\tau) = t_1 : \overline{\mathit{code}}(\tau{\restriction_{t_1}}) \mid\mid \cdots \mid\mid t_n : \overline{\mathit{code}}(\tau{\restriction_{t_n}}),\ \overline{\mathit{code}}(\tau) = \begin{cases} \mathit{code}(\tau) & \textbf{if } \tau \neq \epsilon \\ \mathsf{nop} & \textbf{otherwise} \end{cases}.$$

The base case, when $\tau = \epsilon$, is trivial: $\mathit{program}^n(\tau') = \mathit{program}(\tau)$. Suppose now that $\mathrm{CONC} \vdash \mathit{program}(\tau e\tau') \xrightarrow{\tau}{}^{*} \mathit{program}^n(e\tau')$. We need to show that

$$\langle \mathit{program}^n(e\tau'), \sigma_\tau, \delta_\tau, \rho^n_\tau \rangle \xrightarrow{e}{}^{*} \langle \mathit{program}^n(\tau'), \sigma_{\tau e}, \delta_{\tau e}, \rho^n_{\tau e} \rangle.$$

Suppose that $\mathit{thread}(e) = t_i$. Then:

$$\mathit{program}^n(e\tau'){\restriction_{t_j}} = \begin{cases} \mathit{program}^n(\tau'){\restriction_{t_j}}, & \textbf{if } j \neq i \\ \mathit{code}(e)\ ;\ \mathit{program}^n(\tau'){\restriction_{t_i}}, & \textbf{otherwise} \end{cases}$$

We can build a proof for the above assertion in two execution steps: one generating the event and affecting the configuration by turning the instruction $\mathit{code}(e)$ into nop, and the other dissolving the nop and obtaining the desired configuration. For the first step, the proof tree is build by applying in order $i-1$ steps of Par_2 and then a step of (Par_1) (or none, if $i = n$), then (Thread), (Seq), and finally one of the (Read), (Write), (Acq), or (Rel) to deduce

$$\langle \mathit{code}(e), \sigma_\tau, \delta_\tau, \rho^n_\tau, t \rangle \xrightarrow{e} \langle \mathsf{nop}, \sigma_{\tau e}, \delta_{\tau e}, \rho^n_{\tau e}, t \rangle$$

If $e = (t, \mathit{read}, x, i)$, then $\mathit{code}(e) = \mathsf{load}\ x$ so we can apply the (Read) rule, which updates the register for t of ρ^n_τ to $\sigma_\tau(x) = \mathit{data}(\mathit{latest}_{\mathit{write}}(\tau{\restriction_x}))$. However, due to consistency constraints, $\mathit{data}(e) = \mathit{data}(\mathit{latest}_{\mathit{write}}(\tau{\restriction_x}))$,

whence $\rho^n_\tau[r \leftarrow i] = \rho^n_{\tau e}$ and the rule generates e. Moreover, $\sigma_{\tau e} = \sigma_\tau$ and $\delta_{\tau e} = \delta_\tau$, whence our claim is proven.

If $e = (t, write, x, i)$, then $code(e) = x := i$, and rule *Write* applies generating e; we have that $\sigma_\tau[x \leftarrow i] = \sigma_{\tau e}$, $\delta_{\tau e} = \delta_\tau$, and $\rho^n_{\tau e} = \rho^n_\tau$.

If $e = (t, acquire, x)$, then $code(e) = \mathsf{acquire}\ x$ and because of the consistency requirements for mutexes (the difference between the number of *acquire* and *release* operation is either 0 or 1 for each prefix), it must be that $op(latest(\tau\restriction_x)) \neq acquire$, whence $\delta_\tau(x) = \perp$ and rule *Acq* can apply generating e; we have that $\delta_\tau[x \leftarrow t] = \delta_{\tau e}$, $\sigma_\tau = \sigma_{\tau e}$, and $\rho^n_\tau = \rho^n_{\tau e}$.

If $e = (t, release, x)$, then $code(e) = \mathsf{release}\ x$ and because of the consistency requirements for mutexes, it must be that $op(latest(\tau\restriction_x)) = acquire$, whence $\delta_\tau(x) = thread(latest(\tau\restriction_x))$. Again, from consistency requirements, since all consecutive *acquire-release* pairs share the same thread, it follows that $\delta_\tau(x) = t$ and so the rule *Rel* can apply generating e; we have that $\delta_\tau[x \leftarrow \perp] = \delta_{\tau e}$, $\sigma_\tau = \sigma_{\tau e}$, and $\rho^n_\tau = \rho^n_{\tau e}$. □

The following theorem justifies the maximality claims for the proposed causal model.

Theorem 3 (Maximality) *For any τ' consistent but not τ-feasible there exists a program generating τ but not τ'.*

PROOF. We can assume, without loss of generality, that $\tau' = \tau_1 e'$ such that τ_1 is τ-feasible (potentially empty).

Let $t = thread(e')$.

(1) Suppose $\tau_1\restriction_t$ is a prefix of $\tau\restriction_t$. If $\tau\restriction_t = \tau_1\restriction_t$ then τ' cannot be $program(\tau)$-feasible because that would mean there exists a derivation $\text{CONC} \vdash program(\tau) \xrightarrow{\tau_1}{}^* p_1 \xrightarrow{e'} p'_1$; however since $\text{CONC} \vdash program(\tau) \xrightarrow{\tau}{}^* p$ with $p = program^n(\epsilon)$, from the proof of Proposition 8, $[\text{CONC} \vdash program(\tau) \xrightarrow{\tau}{}^* p]\restriction_t$ must be a proper prefix of $[\text{CONC} \vdash program(\tau) \xrightarrow{\tau_1}{}^* p_1 \xrightarrow{e'} p'_1]\restriction_t$, which is not possible as $p\restriction_t = \mathsf{nop}$.

Otherwise it must be that $\tau\restriction_t = \tau_1\restriction_t e\tau_2$, and we can apply the strong local determinism to deduce that if τ' is $program(\tau)$-feasible, then either $e = e'$ or their type is *read* and their states are different. But both these lead to contradiction because they imply that $\tau_1 e' \in feasible(\tau)$.

(2) If $\tau_1\restriction_t = \tau_0 e'_0$ is not a prefix of $\tau\restriction_t$, then, because τ_1 is τ-feasible, it must be that $\tau\restriction_t = \tau_0 e_0 e \tau_2$ such that $op(e_0) = op(e'_0) = read$, $target(e_0) = target(e'_0)$, and $data(e_0) \neq data(e'_0)$.

Let us consider a special event $?^t = (t, ?)$ (with the meaning that $thread(?^t) = t$ and $op(?^t) = ?$) and for each statement s, an extension $code^s$ of $code$ with $code^s(?^t) = s$, and $program^s(\tau) = code^s(\tau\restriction_{t_1}) \mid\mid \cdots \mid\mid code^s(\tau\restriction_{t_n})$.

Let $x = target(e_0)$, $i = data(e_0)$ and let

$$j = \begin{cases} data(e') - 1, \text{ if } data(e') \text{ defined} \\ 0, \text{ otherwise} \end{cases}$$

Let τ_l, τ_r be such that $\tau = \tau_l e \tau_r$ and $\tau_l\restriction_t = \tau_0 e_0$, and let $p' = program^s(\tau_l ?^t e \tau_r)$, obtained by inserting $s = \mathsf{if}\ i\ \mathsf{then}\ x := j$ in the code for thread t, between

the code for e_0 and that for e. This new program can still generate τ, as the state read by e_0 is different than that read by e'_0 and thus the conditional is not executed, but it cannot generate τ', as the next event for thread t upon generating $\tau_0 e'_0$ must be $(t, write, x, j)$ which is guaranteed to be distinct from e'. □

8 Conclusion and Future Work

We have shown that, by axiomatizing basic properties of (sequentially consistent) concurrent systems, one can obtain maximally sound causal models for concurrent executions, which can be naturally associated to each observed trace, capturing all feasible traces which could be inferred from it. The maximality result has two important theoretical implications. First, verifying the soundness claims for any causal model is reduced to proving that it is a submodel of the maximal one. Second, since the maximal model captures all causally equivalent traces, it allows for universal, model-independent definitions for causal properties. Finally, the model can be explored for finding concurrency anomalies either directly through its axioms (as shown practical by Said et al. [20]) or through its constructive characterization presented in this paper.

We strongly believe that the ideas developed here in the context of sequential consistency can be generalized to more relaxed memory models. However, we expect this task to be more complex, as the axiomatization of these memory models is also more complex. For example, one of the simplest relaxed memory models, the x86-TSO [19], has no less than nine axioms *interrelating* the behavior of concurrent objects. To model it in our framework, one would have to adjust both the consistency criteria—to account for the usage of write buffers—and the local determinism axiom—to account for the fact that commits from the write buffer to memory happen relatively independent of the thread execution. The main difficulty introduced by this new paradigm is that a concurrent object's behavior is no longer independent, but constrained by other concurrent objects: for example, committing a buffered *write* into memory now depends on the fact that the previously buffered *writes* were already committed, while acquiring a lock depends on the fact that the write buffer is empty. Nevertheless, we believe these issues are not insurmountable, but rather promising avenues for further research.

References

[1] Anonymous. details omitted due to double-blind reviewing.

[2] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12:91–122, May 1994. ISSN 0734-2071. doi: 10.1145/176575.176576. URL `http://dx.doi.org/10.1145/176575.176576`.

[3] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul 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. ISBN 1-59593-414-6. doi: 10.1145/1147403.1147416. URL http://dx.doi.org/10.1145/1147403.1147416.

[4] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 278–289, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250697. URL http://dx.doi.org/10.1145/1250662.1250697.

[5] Feng Chen and Grigore Rosu. Parametric and sliced causality. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 240–253, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_27. URL http://dx.doi.org/10.1007/978-3-540-73368-3_27.

[6] 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. ISBN 0-89791-341-8. doi: 10.1145/76263.76329. URL http://dx.doi.org/10.1145/76263.76329.

[7] Perry A. Emrath and David 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. ISBN 0-89791-296-9. doi: 10.1145/68210.69224. URL http://dx.doi.org/10.1145/68210.69224.

[8] Azadeh Farzan and Parthasarathy Madhusudan. Causal atomicity. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 315–328. Springer, 2006. ISBN 3-540-37406-X. doi: 10.1007/11817963_30. URL http://dx.doi.org/10.1007/11817963_30.

[9] Cormac Flanagan and Stephen 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. ISBN 1-58113-729-X. doi: 10.1145/964001.964023. URL http://dx.doi.org/10.1145/964001.964023.

[10] Malay K. Ganai and Aarti Gupta. Efficient modeling of concurrent systems in BMC. In *Proceedings of the 15th international workshop on Model Checking Software*, SPIN '08, pages 114–133, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85113-4. doi: 10.1007/978-3-540-85114-1_10. URL http://dx.doi.org/10.1007/978-3-540-85114-1_10.

[11] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 3540607617.

[12] 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. ISSN 1045-9219. doi: 10.1109/71.238303. URL http://dx.doi.org/10.1109/71.238303.

[13] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12: 463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL http://dx.doi.org/10.1145/78969.78972.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675439. URL http://dx.doi.org/10.1109/TC.1979.1675439.

[15] 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. ISBN 0-387-17906-2.

[16] Mayur Naik, Alex Aiken, and John 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. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134018. URL http://dx.doi.org/10.1145/1133981.1134018.

[17] Robert H. B. Netzer and Barton P. Miller. Detecting data races in parallel program executions. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 109–129. MIT Press, 1990. ISBN 0-262-57080-7.

[18] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, 2003. ISSN 0362-1340. doi: 10.1145/966049.781528. URL http://dx.doi.org/10.1145/966049.781528.

[19] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_27. URL http://dx.doi.org/10.1007/978-3-642-03359-9_27.

[20] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem A. Sakallah. Generating data race witnesses by an SMT-based analysis. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi,

editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2011. ISBN 978-3-642-20397-8. doi: 10.1007/978-3-642-20398-5_23. URL http://dx.doi.org/10.1007/978-3-642-20398-5_23.

[21] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997. ISSN 0734-2071. doi: 10.1145/265924.265927. URL http://dx.doi.org/10.1145/265924.265927.

[22] Edith Schonberg. On-the-fly detection of access anomalies. *SIGPLAN Not. – Best of PLDI 1979-1999*, 39:313–327, April 2004. ISSN 0362-1340. doi: 10.1145/989393.989426. URL http://dx.doi.org/10.1145/989393.989426.

[23] Koushik Sen, Grigore Rosu, and Gul Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *FMOODS*, volume 3535 of *LNCS*, pages 211–226. Springer, 2005. ISBN 3-540-26181-8. doi: 10.1007/11494881_14. URL http://dx.doi.org/10.1007/11494881_14.

[24] Mandana Vaziri, Frank Tip, and Julian 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. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111067. URL http://dx.doi.org/10.1145/1111037.1111067.

[25] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 137–146, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: 10.1145/1122971.1122993. URL http://dx.doi.org/10.1145/1122971.1122993.