Technical Report http://hdl.handle.net/2142/17336, September 2010

Maximal Causal Models for Sequentially Consistent Multithreaded Systems

Traian Florin Șerbănuță, Feng Chen, and Grigore Roșu

Department of Computer Science
University of Illinois at Urbana-Champaign
{tserban2,fengchen,grosu}@illinois.edu

Abstract

This paper shows that it is possible to build a *theoretically maximal and sound* causal model for concurrent computations from a given execution trace. For an observed execution, the proposed model comprises all consistent executions which can be derived from it using only knowledge about the execution machine. The existence of such a model is of great theoretical value. First, by comprising *all* feasible executions, it can be used to prove soundness of other causal models: indeed, several models underlying existing techniques are shown to be embedded into the maximal model, so all these models are sound. 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 an initial observed trace. Thus, maximal causal models are not only theoretically relevant, but they are also amenable for developing practical analysis tools.

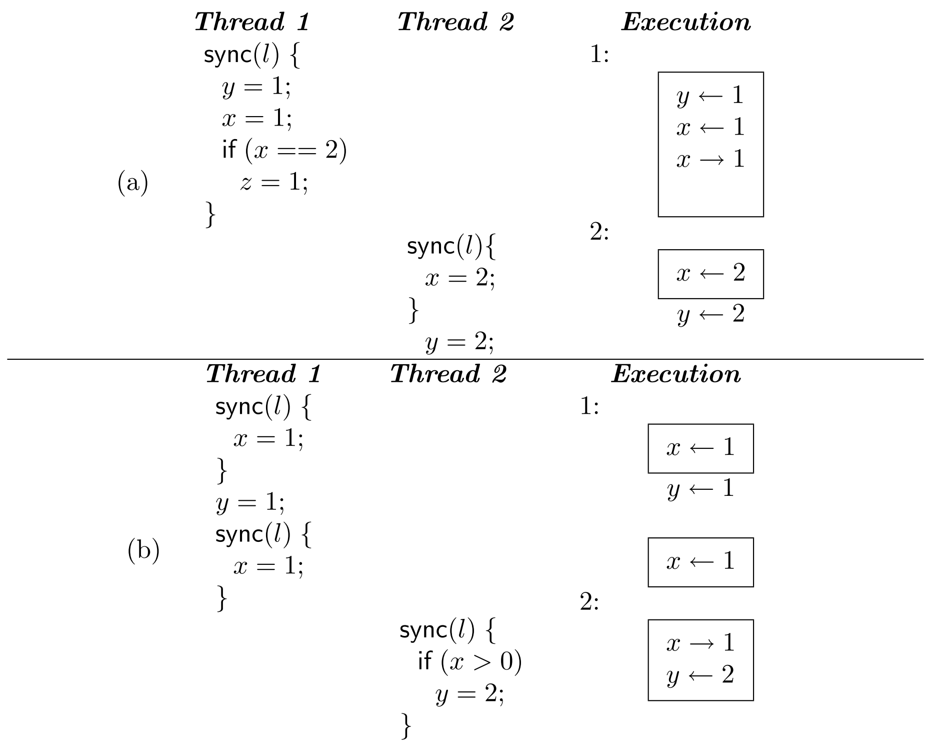
1 Introduction

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 (i.e., at least one of them is a write) to an object are not causally ordered, then a data-race is reported. But is this the best one can do? Of course, not. A series of papers propose more relaxed happen-before causal models where one can also permute blocks protected by the same lock, provided that they access disjoint variables, thus discovering new concurrency bugs not observable with plain happens-before. But is this the best one can do? Of course, not. 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. Others 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 positive reports). 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 build upon some underlying sound causal model, which is possibly being relaxed for efficiency reasons, each paper seems to focus more on how to capture it efficiently rather than on 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 one observed trace. Since what can be inferred from a trace is intrinsically dependent on the chosen causal model, definitions of causal properties differ from technique to technique, with the undesirable result that a causal property (e.g., a datarace) in one model might not be recognized by another model.

Motivating Examples. Each example in Fig. 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 shared location write and read operations are signified 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 Fig. 1(a), a simple happen-before approach ordering all accesses to concurrent objects [18] cannot not 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. Happen-before with lock atomicity [16] 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 happen-before models [19, 22], 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 Fig. 1(a) *has or does not have* a datarace, depending upon the particular causal model employed.



```
(a)
Thread 1            Thread 2            Execution
sync(l) {                               1:
  y = 1;                                   [ y ← 1
  x = 1;                                     x ← 1
  if (x == 2)                                x → 1 ]
    z = 1;
}                                       2:
                    sync(l){               [ x ← 2 ]
                      x = 2;                 y ← 2
                    }
                      y = 2;
```

```
(b)
Thread 1            Thread 2            Execution
sync(l) {                               1:
  x = 1;                                   [ x ← 1 ]
}                                            y ← 1
y = 1;
sync(l) {                                  [ x ← 1 ]
  x = 1;
}                                       2:
                    sync(l) {              [ x → 1
                      if (x > 0)             y ← 2 ]
                        y = 2;
                    }
```

Figure 1: Motivating examples.

However, we were not able to find any existing (sound) causal model able to perceive the race condition in Fig. 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, morover, 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* [12]. 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 assumptions are also errors for other memory models; (3) recent research in computer architecture (e.g., [3]) 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 multithreaded computations, based on consistency and feasibility axioms, which yields sound and maximal (by definition) causal models for observed executions. Next, a series of existing causal models formally or informally used in runtime verification [12, 19, 22] are shown to be subsumed by our model, thus also (re)proving their soundness. Finally, a constructive representation of our proposed models is provided, which could be useful for exploration and other analysis purposes.
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, 7, 10, 15–19, 21]. Our axiomatic approach is closest in spirit to that of Netzer and Miller [15], which proposes an axiomatization of a happens-before causal order between memory accesses and semaphore operations. Instead, we directly axiomatize legal multithreaded systems executions. Some recent model checking approaches (e.g., [8]) make use of similar axiomatizations for sequential consistency. However, their purpose is to reduce the state space to be explored using sequential consistency constraints. Our focus here is rather foundational, attempting to unify existing causal models and causal definitions of execution-dependent properties, by building a maximum model to support them. Another interesting and productive line of research attempts to use information about the actual program code to either statically detect potential bad behaviors [5, 14], or to use information about the program and about the property to be checked to further relax the models of executions [4]. 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 discuses sequential consistency. Section 3 axiomatizes consistent multithreaded systems and defines maximal sound causal models for their executions. Section 4 uses maximal causal 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 our maximal one, thus proving their soundness. Section 6 presents a constructive characterization of the maximal model, and Section 7 concludes. The appendix, included for reviewers' convenience, contains proofs for all results as well as a model checking algorithm based on the results from Section 6.

2 Execution Model and Sequential Consistency

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).

Concurrent Objects, Serial Specification. We adopt the Herlihy and Wing [11] 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.
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.

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, ($\mathit{thread}{=}t_1$, $\mathit{op}{=}\mathit{write}$, $\mathit{target}{=}x$, $\mathit{data}{=}1$) 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., $(t_1, \mathit{write}, x, 1)$. For any event e and attribute *attr*, $\mathit{attr}(e)$ denotes the value corresponding to the attribute *attr* in e, and $e[v/\mathit{attr}]$ 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.

Sequential consistency can be now elegantly defined as follows:

Definition 1 (Attiya and Welch [1]). *Let τ be any trace. Then:*
*(1) τ is **legal** iff $\tau\restriction_o$ satisfies o's serial specification for any object o;*

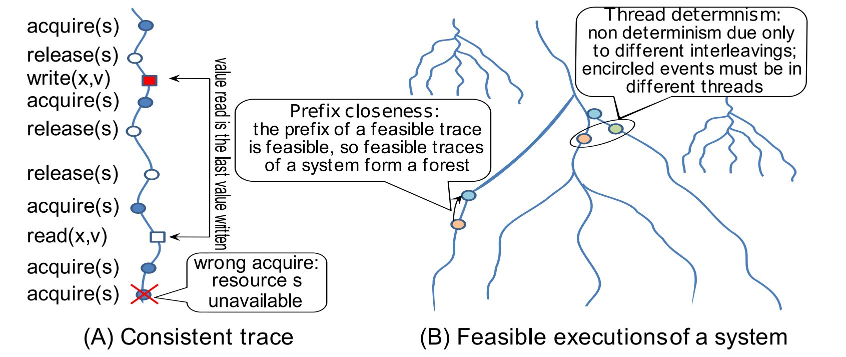

acquire(s)
release(s)
write(x,v)
acquire(s)
release(s)
release(s)
acquire(s)
read(x,v)
acquire(s)
acquire(s)
value read is the last value written
wrong acquire: resource s unavailable
Prefix closeness: the prefix of a feasible trace is feasible, so feasible traces of a system form a forest
Thread determnism: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

Figure 2: Consistent traces and feasible executions

(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 to any observed execution a maximal sound causal model, comprising *all* executions which can potentially be inferred from that execution alone, without additional knowledge of the system generating it.

Fig. 2 highlights the two major concepts underlying our approach, namely *consistent traces* and *feasible executions*. A consistent trace (Def. 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 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 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, *thread 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 *feasible*$(\mathcal{S})$ be their set. Instead of defining *feasible*$(\mathcal{S})$ as we did in [20], 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, $\mathit{feasible}(\mathcal{S})$ must be prefix closed: if $\tau_1\tau_2$ is $\mathcal{S}$-feasible, then τ_1 is $\mathcal{S}$-feasible.
Thread 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. Formaly, 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 $\mathit{op}(e)$ is a request for a value from $\mathit{target}(e)$ (e.g., $\mathit{op}(e) = \mathit{read}$) 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 an operation requesting a value from a concurrent object (e.g., reading the value of a memory location) 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** iff $\mathit{feasible}(\mathcal{S})$ 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 *maximal* 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 $\mathit{feasible}(\tau)$, is the smallest set of traces containing τ which is prefix-closed and satisfies the thread determinism property. A trace in $\mathit{feasible}(\tau)$ is called τ-**feasible**.*

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

The intuition for $\tau \in \mathit{feasible}(\sigma)$ 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 can produce τ. Since $\mathit{feasible}(\sigma)$ was chosen to be the smallest set of traces closed under the axioms above, it follows, also intuitively, that if $\tau \notin \mathit{feasible}(\sigma)$ then there is some program that yields σ but which cannot yield τ. We say "intuitively" since, for simplicity, we here did not give a formal definition of $\mathcal{S}$, programs, and their execution; the interested reader can check [20].

Therefore, observing an execution trace τ, one can regard $\mathit{feasible}(\tau)$ and the *maximal causal model* corresponding to τ, in that it comprises all the traces that can be thought of as causally equivalent to τ. We are not concerned with how to encode or represent this causal model (one can, e.g., regard τ as its representation). In Section 6 we show that, even though it has an existential nature, the traces comprised by this model can be effectively generated and thus analyzed.

4 Formal Definitions for Causal Properties

The benefits of having a maximal causal model are twofold: (1) one can use this model to prove soundness of other causal models by showing that they can be embedded in the maximal one (we discuss this in Section 5); and (2) having a maximal model allows for uniform and consistent along models definitions of causal properties for traces, such as dataraces and atomicity. Indeed, having a maximal model generalizing all possible sound models allows for unique semantical definitions which can be shared among all such models. Let us clarify this idea below, using causal dataraces and atomicity as guiding examples.

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 $\mathit{thread}(e_1) \neq \mathit{thread}(e_2)$, $\mathit{target}(e_1) = \mathit{target}(e_2)$, and $\mathit{write} \in \{\mathit{op}(e_1), \mathit{op}(e_2)\}$. A *datarace* occurs when an execution contains two events having a data-conflict, without proper synchronization between them [18]. 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, 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 [18]; if considering happens before with locksets [16], 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 our feasibility closure is obtained directly from an axiomatization of a consistent system, and is thus maximal by definition, 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 iff there exists a τ-feasible trace σ such that $\sigma\restriction_{\mathit{thread}(e_1)} = \tau_1\restriction_{\mathit{thread}(e_1)}$ and $\sigma\restriction_{\mathit{thread}(e_2)} = \tau_1 e_1 \tau_2\restriction_{\mathit{thread}(e_2)}$.*

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 thread 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.

Using this definition, the race in Fig. 1 is finally captured by the causal datarace definition, having $(1, \mathit{acquire}, l)(1, \mathit{write}, x, 1)(1, \mathit{release}, l)(2, \mathit{acquire}, l)(2, \mathit{read}, x, 1)$ as a witness belonging to the feasibility closure of the observed execution.

Atomicity. Similarly, one can easily define within this model a proper notion of atomicity associated to a consistent trace. 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 [22], or local atomicity [6], 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 strengthens the intuition for the maximality of our model and, moreover, shows that these rather adhoc (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.*

Theorem 1, in combination with Proposition 1, additionally shows 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.

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 [9].

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 [13]: 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:

Theorem 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)*.*

Weak Happens Before. Several more recent trace analysis techniques [19, 22] 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. 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)*, 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. [19] 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 also noticed by Wang and Stoller [22], 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 the maximal model.

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

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 happen-before dependence as if they were particular *write* and *read* operations (on the lock variable) [18]. However, this prevents synchronized blocks from being permuted, and thus imposes coverage limitations. The lock-set approaches, also called hybrid happen-before [16], propose to handle locks separately, associating to each event the set of locks [17] 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:

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

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 maximal model advocated in this paper.
Lock atomicity via write-read atomicity [19]. 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 dependences.

However, this definition needs a careful approach. Consider the example in Fig. 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.

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

Lock atomicity via locksets. Wang and Stoller [22] 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.

Theorem 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.*

6 Characterizing the Feasibility Closure

Section 3 showed how the maximal 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. However, for analysis purposes, it is preferable to have a constructive way of computing the feasibility closure. 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 thread determinism axiom, that is, the one regarding operations which might receive different values from their objects. To achieve a complete constructive characterization of feasibility closures, we have to go beyond prefixes of interleavings, more exactly, one request operation per thread beyond. This is because, as guaranteed by thread 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 different value 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 $op(e)$ requests a value from the concurrent object, 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 sound and complete characterization for the τ-feasible traces:

Theorem 7. *Given a consistent trace τ, a trace τ' is τ-feasible iff it is a consistent extended interleaving prefix of τ.*

7 Conclusion and Future Work

We have developed theoretically sound and maximal causal models for concurrent executions, which can be naturally associated to each observed trace, capturing all the feasible and causally equivalent traces which could be inferred from the observed trace. 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 maximal model can effectively be explored (and model checked), suggesting that it may also have practical potential.

An especially promising line of research is studying the complexity of verifying candidates for a causal property (violation) against the maximal model, and finding efficient algorithms for doing so.

References

[1] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *TOCS*, 12(2):91–122, 1994.

[2] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. A theory of data race detection. In *PADTAD*, pages 69–78, 2006.

[3] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: bulk enforcement of sequential consistency. In *ISCA*, pages 278–289, 2007.

[4] F. Chen and G. Roşu. Parametric and sliced causality. In *CAV*, volume 4590 of *LNCS*, pages 240 – 253, 2007.

[5] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *PADD*, pages 89–99, 1988.

[6] A. Farzan and M. Parthasarathy. Causal atomicity. In *CAV*:315–328, 2006.

[7] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.

[8] M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In *SPIN*, pages 114–133, 2008.

[9] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *LNCS*. Springer, 1996.

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

[11] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Transactions on Computers*, 28(9):690–691, 1979.

[13] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, volume 255 of *LNCS*, pages 279–324, 1987.

[14] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319, 2006.

[15] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. In *LCPC*, pages 109–129. MIT, 1990.

[16] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, pages 167–178, 2003.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP*, pages 27–37, 1997.

[18] E. Schonberg. On-the-fly detection of access anomalies. In *PLDI*, pages 285–297, 1989.

[19] K. Sen, G. Roșu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis. In *FMOODS*, pages 211–226, 2005.

[20] T. F. Șerbănuță, F. Chen, and G. Roșu. Maximal causal models for multithreaded systems. Technical Report `http://fsl.cs.uiuc.edu/pubs/mm.pdf`.

[21] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345, 2006.

[22] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPOPP*, pages 137–146, 2006.

A Proofs of the results

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

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

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 $n = \mathit{thread}(e)$, 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 $(\tau'e)\restriction_n = \tau'\restriction_n e$ is a prefix of $\tau\restriction_n$, it follows that $\tau'\restriction_n = \tau_1\restriction_n$. Using the thread determinism for $\tau_1 e$ and τ', we obtain that $\tau'e$ is τ-feasible (since it is consistent). □ □

Theorem 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', $[\tau'] \subseteq \mathit{feasible}(\tau)$. Hence, $[\tau] \subseteq \mathit{feasible}(\tau)$.*

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 thread 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, $[\tau'] \subseteq \mathit{feasible}(\tau') \subseteq \mathit{feasible}(\tau)$. □
□

Theorem 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 $<^{wr}_{\sigma_2} = <^{wr}_{\sigma_1}$, 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. □ □

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

Proof. Again, we reduce our proof to Theorem 7. First, any linearization of $\prec^{\sigma}_{hb}$ 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 specifiaction for mutexes is also satisfied. Therefore, any linearization of $\prec^{\sigma}_{hb}$ consistent with the lock atomicity of σ, is a consistent interleaving of σ, thus σ-feasible. □ □

Theorem 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 is in the $<^{a}_{\sigma}$ relation. □ □

Theorem 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 Theorem 3, $<^{wr}_{\sigma'} = <^{wr}_{\sigma}$ implies the serial specification of memory locations is obeyed in σ'. Additionally, from the proof of Theorem 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. □ □

Theorem 7. *Given a consistent trace τ, 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 thread 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 $\mathit{thread}(e) = n$, and $\pi_n(\tau_1) = \pi_n(\tau_2)$. Since $\pi_n(\tau_1 e)$ is an extended prefix of $\pi_n(\tau)$, then either $\pi_n(\tau_1 e)$ is a prefix of τ, or $\mathit{op}(e) = \mathit{read}$, and there exists e', such that $\mathit{thread}(e') = n$, $\mathit{op}(e') = \mathit{read}$, $\mathit{target}(e') = \mathit{target}(e)$, and $\pi_n(\tau_1)e'$ is a prefix of $\pi_n(\tau)$.

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

B Model Checking Algorithm.

The algorithm in Fig. 3 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 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 16), and the search for the next advance-able thread is restarted (line 17). If no additional thread can be advanced from this state, the algorithm backtracks, undoing the effects of previous advances (lines 18–26).

A simple amortized analysis of our 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 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.

```
Input:   Trace $\tau_0$ of size $n$ over $\mathsf{k}$ threads.
Maps:    $\mathsf{thread} : \{1, \ldots, \mathsf{k}\} \rightarrow \mathit{Stack}$
         $\sigma : \mathit{Locations} \rightarrow \mathit{Int}$
Initial: $\sigma[x] \leftarrow 0$, for all variables
         $\sigma[l] \leftarrow 0$, for all semaphores
         $\mathsf{thread}[t] \leftarrow \tau_0{\restriction_t}$, for all threads
         $\mathsf{Advanceable} \leftarrow \{1, \ldots, \mathsf{k}\}$
$\tau \leftarrow \epsilon$; $t \leftarrow 0$;
while $t < \mathsf{k}$ do
    $t \leftarrow t + 1$;
    if $t \in \mathsf{Advanceable}$ then
        $e \leftarrow$ top(thread[$t$]);
        if $op(e) \neq acquire \vee \sigma[target(e)] > 0$ then            // advance
            $l \leftarrow target(e)$;
            if $op(e) = read \wedge data(e) \neq \sigma[l]$ then     // extended prefix
                $\mathsf{Advanceable} \leftarrow \mathsf{Advanceable} \setminus \{t\}$ ;
                $data(e) \leftarrow \sigma[l]$;
            else                                                  // update state
                pop(thread[$t$]);
                if $op(e) = write$ then $\sigma[l] \leftarrow data(e)$;
                if $op(e) = acquire$ then $\sigma[l] \leftarrow \sigma[l] - 1$;
                if $op(e) = release$ then $\sigma[l] \leftarrow \sigma[l] + 1$;
            end
            push($\tau$,$e$); check $\tau$ against $\varphi$;
            $t \leftarrow 0$;
        end
    end
    while $t = \mathsf{k} \wedge \tau \neq \epsilon$ do                               // backtrack
        $e \leftarrow$ pop($\tau$); $t \leftarrow thread(e)$; $l \leftarrow target(e)$;
        if $t \notin \mathsf{Advanceable}$ then                          // extended prefix
            $\mathsf{Advanceable} \leftarrow \mathsf{Advanceable} \cup \{t\}$;
        else                                                     // restore state
            push(thread[$t$],$e$);
            if $op(e) = write$ then $\sigma[l] \leftarrow data(\mathsf{latest}(\tau, l))$;
            if $op(e) = acquire$ then $\sigma[l] \leftarrow \sigma[l] + 1$;
            if $op(e) = release$ then $\sigma[l] \leftarrow \sigma[l] - 1$;
        end
    end
end
```

Figure 3: Exploring the feasibility closure