CAV'07, LNCS 4590, pp 240 - 253, 2007

Parametric and Sliced Causality

Feng Chen and Grigore Roşu

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

Abstract. Happen-before causal partial orders have been widely used in concurrent program verification and testing. This paper presents a parametric approach to happen-before causal partial orders. Existing variants of happen-before relations can be obtained as instances of the parametric framework. A novel causal partial order, called *sliced causality*, is then defined also as an instance of the parametric framework, which loosens the obvious but strict happen-before relation by considering static and dynamic dependence information about the program. Sliced causality has been implemented in a runtime predictive analysis tool for Java, named jPredictor, and the evaluation results show that sliced causality can significantly improve the capability of concurrent verification and testing.

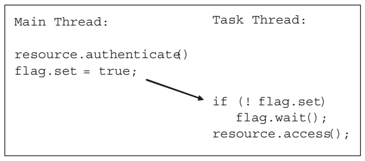
1 Introduction

Concurrent systems are notoriously difficult to verify, test and debug due to their inherent nondeterminism. The *happen-before* causality, first introduced in [14], provides an effective way to analyze the potential dynamic behaviors of concurrent systems and has been widely used in concurrent program verification and testing [21, 16, 18, 19, 9]. Approaches based on happen-before causality extract causal partial orders by analyzing process or thread communication at runtime; the extracted causal partial order can be regarded as an abstract model of the runtime behaviors of the program and thus can be checked against the desired property. This way, one analyzes *a class of executions* that are characterized by the same causal partial order. Therefore, for verification, the state space to explore can be reduced, while for testing, one can predict potential errors without re-execute the program. Happen-before based approaches are sound (no false alarms) and can handle general purpose properties, e.g., temporal ones.

Several variants of happen-before causalities have been introduced for applications in different domains [14, 16, 19]. Although these notions are similar in principle, there is however no adequate unifying framework for all of them. A proof of soundness has to be re-done for every variant of happen-before causality, which typically involves sophisticate details of the specific domain. This may slow future developments, in particular defining new, or domain-specific causalities. The first contribution of this paper is a *parametric framework* for causal partial orders, which is purposely designed to facilitate defining and proving correctness of happen-before causalities. The proof of correctness of a happen-before relation is reduced to proving a simple, easy to check closed local property of causal dependence. Existing variants of happen-before relations can be obtained and proved sound as instances of our parametric framework.

The second contribution of this paper consists of using the above framework to define a novel causal partial order relation, called *sliced causality*, which aims at improving coverage of analysis without giving up soundness or genericity of properties to check: it works with any *monitorable* (safety) properties, including regular patterns, temporal assertions, data-races, atomicity, etc. Previous approaches based on happen-before (such as [16, 18, 19]) extract causal partial orders from analyzing *exclusively* the dynamic thread communication in executions. Since they consider *all* interactions among threads, e.g., all reads/writes of shared variables, the obtained causal partial orders are rather *restrictive*, or *rigid*, in the sense of allowing a reduced number of linearizations and thus of errors that can be detected. In general, the larger the causality (as a binary relation) the fewer linearizations it has, i.e., the more restrictive it is.

Let us consider a simple and common safety property for a shared resource, that any access should be authenticated. Figure 1 shows a buggy multi-threaded program using the shared resource. The main thread authenticates and then the task thread uses the authenticated resource. They communicate via the `flag` variable. Synchronization is unnecessary, since only the main thread modifies `flag`. However, the developer makes a (rather common [7]) mistake by using `if` instead of `while` in the task thread. Suppose now that we observed a successful run of the program, as shown by the arrow. The traditional happen-before will not be able to find the bug because of the causality induced by write/read on `flag`. But since `resource.access()` is *not* controlled by `if`, our technique will be able to correctly predict the violation from the successful execution. When the bug is fixed by replacing `if` with `while`, `resource.access()` will be controlled by `while` because it is a non-terminating loop; then no violation will be reported by our technique.



```
Main Thread:                    Task Thread:

resource.authenticate()
flag.set = true;
                                if (! flag.set)
                                   flag.wait();
                                resource.access();
```

Fig. 1. Multi-threaded execution

In summary, based on an apriori static analysis, sliced causality drastically cuts the usual happen-before causality by removing unnecessary dependencies; this way, a significantly larger number of consistent runs can be inferred and thus analyzed. Experiments show that, on average, the sliced causality relation has about 50% direct inter-thread causal dependencies compared to the more conventional happen-before partial order (Section 5). Since the number of linearizations of a partial order tends to be exponential with the size of the *complement* of the partial order, any linear reduction in size of the sliced causality compared to traditional happen-before relations is expected to *increase exponentially the coverage* of the corresponding analysis, still avoiding any false alarms. Indeed, the use of sliced causality allowed us to detect concurrency errors that would be very little likely detected using the conventional happen-before causalities.

This paper is organized as follows. Section 2 introduces existing happen-before causalities. Section 3 defines our parametric framework. Section 4 proposes sliced causality. Section 5 discusses the evaluation of sliced causality and Section 6 concludes. Most proofs are omitted because of limited space; they can be found in [4].

2 Happen-Before Causalities

The first happen-before relation was introduced almost 3 decades ago by Lamport [14], to formally model and reason about concurrent behaviors of distributed systems. Since then, a plethora of variants of happen-before causal partial order relations have been introduced in various frameworks and for various purposes. The basic idea underlying happen-before relations is to observe the events generated by the execution of a distributed system and, based on their order, their type and a straightforward causal flow of information in the system (e.g., the receive event of a message follows its corresponding send event), to define a partial order relation, the happen-before causality. Two events related by the happen-before relation are causally linked in that order.

When using a particular happen-before relation for (concurrent) program analysis, the crucial property of the happen-before relation is that, for an observed execution trace τ, other *sound permutations* of τ, also called *linearizations* or *linear extensions* or *consistent runs* or even *topological sortings* in the literature, are also possible computations of the concurrent system. Consequently, if any of these linearizations violates or satisfies a property φ, then the system can indeed violate or satisfy the property, regardless of whether the particular observed execution that generated the happen-before relation violated or satisfied the property, respectively. For example, [6] defines formulae *Definitely*(φ) and *Possibly*(φ), which hold iff φ holds in all and, respectively, in some possible linearizations of the happen-before causality.

The soundness/correctness of a happen-before causality can be stated as follows: given a happen-before causal partial order extracted from a run of the concurrent system under consideration, all its linearizations are *feasible*, that is, they correspond to other possible execution of the concurrent system. To prove it, one needs to formally define the actual computational model and what a concurrent computation is; these definitions tend to be rather intricate and domain-specific. For that reason, proofs need to be redone in different settings facing different "details", even though they follow conceptually the same idea. In the next section we present a simple and intuitive property on traces, called *feasibility*, which ensures the desired property of the happen-before causality and which appears to be easy to check in concrete situations.

To show how the various happen-before causalities fall as special cases of our parametric approach, we recall two important happen-before partial orders, one in the context of distributed systems where communication takes place exclusively via message passing, and another in the context of multithreaded systems, where communication takes place via shared memory. In the next section we show that their correctness [2, 19] follow as corollaries of our main theorem. In Section 4 we define another happen-before causality, called *sliced causality*, which non-trivially uses static analysis information about the multithreaded program. The correctness of sliced causality will also follow as a corollary of our main theorem in the next section.

In the original setting of [14], a distributed system is formalized as a collection of processes communicating only by means of asynchronous message passing. A process is a sequence of events. An event can be a *send* of a message to another process, a *receive* of a message from another process, or an *internal* (local) event.

Definition 1. *Let τ be an execution trace of a distributed system consisting of a sequence of events as above. Let E be the set of all events appearing in τ and let the* ***happen-before*** *partial order "$\rightarrow$" on E be defined as follows:*

1. *if e_1 appears before e_2 in some process, then $e_1 \rightarrow e_2$;*
2. *if e_1 is the send and e_2 is the receive of the same message, then $e_1 \rightarrow e_2$;*
3. *$e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ implies $e_1 \rightarrow e_3$.*

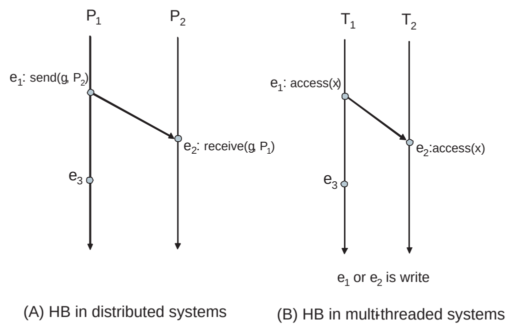


Fig. 2. Happen-before partial-order relations

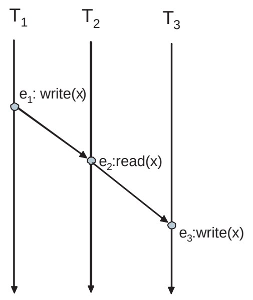


Fig. 3. Happen-before causality in multi-threaded systems

A space-time diagram to illustrate the above definition is shown in Figure 2 (A), in which e_1 is a send message and e_2 is the corresponding receive message; $e_1 \rightarrow e_2$ and $e_1 \rightarrow e_3$, but e_2 and e_3 are not related. It is easy to prove that $(E, \rightarrow)$ is a partial order. The soundness of this happen-before relation, i.e., all the permutations of τ consistent with $\rightarrow$ are possible computations of the distributed system, was proved in [2] using a specific formalization of the global state of a distributed system. This property will follow as an immediate corollary of our main theorem in the next section.

Happen-before causalities have been devised in the context of multithreaded systems for various purposes. For example, [15, 16] propose datarace detection techniques based on intuitive multi-threaded variants of happen-before causality, [19] proposes a happen-before relation that drops read/read dependencies, and [20] even drops the write/write conflicts but relates each write with all its subsequent reads atomically.

Finding appropriate happen-before causalities for multithreaded systems is a non-trivial task. The obvious approach would be to map the inter-thread communication in a multi-threaded system into send/receive events in some corresponding distributed system. For example, starting a new thread generates two events, namely a send event from the parent thread and a corresponding receive event from the new thread; releasing a lock is a send event and acquiring a lock is a receive event. A write on a shared variable is a send event while a read is a receive event. However, such a simplistic mapping suffers from several problems related to the semantics of shared variable accesses. First, every write on a shared variable can be followed by multiple reads whose order should not matter; in other words, some "send" events now can have multiple corresponding "receive" events. Second, consider the example in Figure 3. Since e_3 may write a different value into x other than e_1, the value read at where e_2 occurs may change if we observe e_3 after e_1 but before e_2 appears. Therefore, $e_1e_3e_2$ may not be a trace of some feasible execution since e_2 will not occur any more. Hence, a causal order between e_2 and e_3 should be enforced, which cannot be captured by the original definition in [14].

The various causalities for multithreaded systems address these problems (among others). However, each of them still needs to be proved correct: any sound permutation of events results in a feasible execution of the multithreaded system. If one does not prove such a property for one's desired happen-before causality, then one's analysis techniques can lead to false alarms. We next recall one of the simplest happen-before relations for multi-threaded systems [19]:

Definition 2. *Let τ be an execution of a multithreaded system, let E be the set of all events in τ, and let the* ***happen-before*** *partial order "$\rightsquigarrow$" on E be defined as follows:*

1. *if e_1 appears before e_2 in some thread, then $e_1 \rightsquigarrow e_2$;*
2. *if e_1 and e_2 are two accesses on the same shared variable such that e_1 appears before e_2 in the execution and at least one of them is a write, then $e_1 \rightsquigarrow e_2$;*
3. *$e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_3$ implies that $e_1 \rightsquigarrow e_3$.*

In Figure 2 (B), $e_1 \rightsquigarrow e_2$ and $e_1 \rightsquigarrow e_3$, but e_2 and e_3 are not comparable under $\rightsquigarrow$.

3 Parametric Framework for Causality

We here define a parametric framework that axiomatizes the notion of causality over events and *feasibility* of traces in a system-independent manner. We show that proving the feasibility of the linearizations of a causal partial order extracted from an execution can be reduced to checking a simpler "closure" local property on feasible traces.

Let Events be the set of all events. A trace τ is a finite ordered set of (distinct) events $\{e_1 < e_2 < \cdots < e_n\}$, usually identified with the Events^* word $e_1 \cdots e_n$. Let $\xi_\tau = \{e_1, e_2, \ldots, e_n\}$ be called the alphabet of τ and $<_\tau$ be the total order on ξ_τ induced by τ. Let Traces denote the set of all such traces. Given a set X, let $\mathcal{PO}(X)$ denote the set of partial orders defined on subsets of X, that is, the set of pairs $(\xi, <)$ where $\xi \subseteq X$ and $< \subseteq \xi \times \xi$ is a partial order.

Definition 3. *A* causality operator *is a partial function $C : \mathit{Traces} \nrightarrow \mathcal{PO}(\mathit{Events})$ s.t.:*

1. *If $C(\tau) = (\xi, <)$, then $\xi = \xi_\tau$ and $< \subseteq <_\tau$; and*
2. *For any $\tau = \tau_1 e_1 e_2 \tau_2$ such that $C(\tau) = (\xi, <)$ with $e_1 \not< e_2$, $C(\tau_1 e_2 e_1 \tau_2)$ is also defined and equal to $C(\tau)$.*

Let $Dom(C)$ denote the domain of C.

Note that condition (ii) in the definition above is closely related to that of trace equivalence introduced by Mazurkiewicz in [11]. However, the theory of Mazurkiewicz traces starts with a given dependency relation on events and considers equivalence of traces according to that fixed dependency, while in our framework, we prefer to associate a separate dependency relation to each trace, assuming that only at runtime we can get enough information about the causality for a given trace; for example, acquiring lock l and writing shared variable x can be or not dependent events, depending on the particular execution of the program. This allows us to be more precise while still using part of the generic Mazurkiewicz trace theory to simplify our correctness proofs.

Any concurrent system can produce only a particular subset of *feasible* traces, which are in the following relationship with the corresponding causality operator:

Definition 4. *Given a causality operator C, a set $\mathcal{F}$ of traces is C-*feasible *iff C is defined on $\mathcal{F}$ and for any $\tau \in \mathcal{F}$ with $C(\tau) = (\xi, <)$, $\mathcal{F}$ contains all the linearizations of $C(\tau)$ (i.e., all traces τ' such that $\xi_{\tau'} = \xi$ and$< \subseteq <_{\tau'}$).*

Theorem 1. *$Dom(C)$ is C-feasible, More precisely, if $C(\tau) = (\xi, <)$ then for any τ', $C(\tau') = C(\tau)$ if and only if $\xi_{\tau'} = \xi$ and $< \subseteq <_{\tau'}$.*

Corollary 1. *$\mathcal{F}$ is C-feasible iff C is defined on $\mathcal{F}$ and for any $\tau = \tau_1 e_1 e_2 \tau_2$ such that $C(\tau) = (\xi, <)$, $e_1 \not< e_2$ implies $\tau_1 e_2 e_1 \tau_2 \in \mathcal{F}$.*

The two variants of happen-before relations discussed in Section 2 can be captured as instances of our parametric framework. For the happen-before relation defined in Definition 1, let Events_{hb} be the set of all the send, receive and internal events.

Corollary 2. *For an observed trace τ, any permutation of τ consistent with $\rightarrow$ is a possible computation of the distributed system.*

Proof: Let $\mathcal{C}_{hb}$ be the partial function $\mathit{Traces}_{hb} \twoheadrightarrow \mathcal{PO}(\mathit{Events}_{hb})$ with $\mathcal{C}_{hb}(\tau) = (\xi_\tau, \rightarrow)$ for any $\tau \in \mathit{Traces}_{hb}$. Let $\mathcal{F}_{hb}$ be the set of computation traces of the distributed system as defined in [2]. The result follows from Corollary 1, noticing that C_{hb} is a causality operator and $\mathcal{F}_{hb}$ is $\mathcal{C}_{hb}$-feasible. □

For the happen-before relation in Definition 2, let Events_{mhb} be the set of all the write and read events on shared variables as well as all internal events.

Corollary 3. *For an observed trace τ of a multi-threaded system, any permutation of τ consistent with $\leadsto$ is a possible execution of the multi-threaded system.*

Proof: Let $\mathcal{C}_{mhb}$ be the partial function $\mathit{Traces}_{mhb} \twoheadrightarrow \mathcal{PO}(\mathit{Events}_{mhb})$ with $\mathcal{C}_{mhb}(\tau) = (E_\tau, \leadsto)$ for any $\tau \in \mathit{Traces}_{mhb}$. Let $\mathcal{F}_{mhb}$ be the set of traces that are generated by all feasible executions of the multi-threaded system (see, e.g., [19] for a formalization of multi-threaded systems). The result follows from Corollary 1, noticing that C_{mhb} is a causality operator and $\mathcal{F}_{mhb}$ is $\mathcal{C}_{mhb}$-feasible. □

4 Sliced Causality

Without additional information about the structure of the program that generated the event trace τ, the least restrictive causal partial order that an observer can extract from τ is the one which is total on the events generated by each thread and in which each write event of a shared variable precedes all the corresponding subsequent read events. This is investigated and discussed in detail in [20]. In what follows we show that one can construct a much more general causal partial order, called *sliced causality*, by making use of dependence information obtained statically and dynamically. Briefly, instead of computing the causal partial order on all the observed events like in the traditional happen-before based approaches, our approach first slices τ according to the desired property and then computes the causal partial order on the achieved slice; the slice contains all the events relevant to the property, as well as all the events upon which the relevant events depend. This way, irrelevant causality on events is trimmed without breaking the soundness of the approach, allowing more permutations of relevant events to be analyzed and resulting in better coverage of the analysis.

We employ dependencies among events to assure the correct slicing. The dependence discussed here somehow relates to *program slicing* [12], but we focus on finer grained units here, namely events, instead of statements. Our analysis keeps track of actual memory locations in every event, available at runtime, avoiding inter-procedural analysis. Also, we need *not* maintain the entire dependence relation, since we only need to compute the causal partial order among events that are relevant to the property to check. This leads to an effective vector clock (*VC*) algorithm ([3]).

Intuitively, event e' *depends upon* event e in τ, written $e \sqsubset e'$, iff a change of e may change or eliminate e'. This tells the observer that *e should occur before e' in any consistent permutation of τ*. There are two kinds of dependence: (1) *control dependence*, written $e \sqsubset_{ctrl} e'$, when a change of the state of e may eliminate e'; and (2) *data-flow dependence*, written $e \sqsubset_{data} e'$, when a change of the state of e may lead to a change in the state of e' . While the control dependence only relates events generated by the same thread, the data-flow dependence may relate events generated by different threads: e may write some shared variable in a thread t, which is then read in another thread t'.

4.1 Events and Traces

Events represent atomic steps observed in the execution of the program. In this paper, we focus on multi-threaded programs and consider the following types of events (other types can be easily added): write/read of variables, beginning/ending of function invocations, acquiring/releasing locks, and starts and exits of threads. A statement in the program may produce multiple events. Events need to store enough information about the program state in order for the observer to perform its analysis.

Definition 5. *An* ***event*** *is a mapping of* ***attributes*** *into corresponding* ***values****. A* ***trace*** *is a sequences of events. We let τ, τ', etc., denote traces. From now on in the paper, we assume an arbitrary but fixed trace τ and let ξ denote ξ_τ (recall $\xi_\tau = \{e \mid e \in \tau\}$) for simplicity; events in ξ are called* ***concrete events****.*

For example, one event can be $e_1 : (counter = 8, thread = t_1,\ stmt = L_{11},\ type = write,\ target = a,\ state = 1)$, which is a write on location a with value 1, produced at statement L_{11} by thread t_1. One can easily include more information into an event by adding new attribute-value pairs. We use *key*(e) to refer to the value of attribute *key* of event e. The attribute *state* contains the value associated to the event; specifically, for the write/read on a variable, *state*(e) is the value written to/read from the variable; for ending of a function call, *state*(e) is the return value if there is one; for the lock operation, *state*(e) is the lock object; for other events, *state*(e) is undefined. To distinguish among different occurrences of events with the same attribute values, we add a designated attribute to every event, *counter*, collecting the number of previous events with the same attribute-value pairs (other than the *counter*). This way, all events appearing in a trace can be assumed different.

4.2 Control Dependence on Events

Informally, if a change of *state*(e) may affect the occurrence of e', then we say that e' has a *control dependence* on e, and write $e \sqsubset_{ctrl} e'$. For example, in Figure 4, the write on x at S_1 and the write on y at S_2 have a control dependence on the read on i at C_1, while the write on z at S_3 does not have such control dependence. Control dependence occurs inside of a thread, so we first define the total order within one thread:

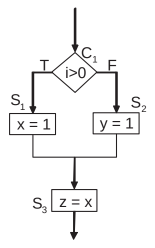


Fig. 4. Control dependence

Definition 6. *Let $<$ denote the union of the total orders on events of each thread, i.e., $e < e'$ iff thread(e) = thread(e') and $e <_\tau e'$.*

The control dependence among events in sliced causality is parametric in a control dependence relation among statements. In particular, one can use off-the-shelf algorithms for classic [8] or for weak [17] control dependence. We chose to use the termination-sensitive control dependence (TSCD) introduced in [5] in our implementation of jPREDICTOR[3]. Nevertheless, all we need to define sliced causality is a function returning the *control scope* of any statement C, say *scope* (C), which is the set of statements whose reachability depends upon the choice made at C, that is, the statements that control depend on C, for some appropriate or preferred notion of control dependence. Our approach also regards the lock acquire statement as a control statement that controls all the following statements, since the thread has to wait for the lock to continue its execution.

We assume that any control statement generates either a *read* event (the lock acquire is regarded as a read on the lock) or no event (the condition is a constant) when checking its condition. For the control statement with a complex condition, e.g., involving function calls and side effects, we can always transform the program to simplify its condition to a simple check of a boolean variable: one can compute the original condition before the control statement, store its result in a fresh boolean variable, and then modify the control statement to check only that variable in its condition.

Definition 7. *$e \sqsubset_{ctrl} e'$ iff $e < e'$, $stmt(e') \in scope(stmt(e))$, and e is "largest" with this property, i.e., there is no e'' such that $e < e'' < e'$ and $stmt(e') \in scope(stmt(e''))$.*

Intuitively, an event e is control dependent on the *latest* event issued by some statement upon which $stmt(e)$ depends. For example, in Figure 4, a write of x at S_1 is control dependent on the most recent read of i at C_1 and not on previous reads of i at C_1.

The *soundness* of analysis based on sliced causality is contingent to the *correctness* (no false negatives) of the employed control dependence: the analysis produces no false alarms when the *scope* function returns for each statement *at least* all the statements that control-depend on it. An extreme solution is to include all the statements in the program in each scope, in which case sliced causality becomes precisely the classic happen-before relation. As already pointed out in Section 1 and empirically shown in Section 5, such a choice significantly reduces the coverage of analysis. A better solution, still over-conservative, is to use weak dependence when calculating the control scopes. If termination information of loops is available, termination-sensitive control dependence can be utilized to provide correct and more precise results. One can also try to use the classic control dependence instead, but one should be aware that false bugs may be reported (e.g., when synchronization is implemented based on "infinite" loops).

4.3 Data Dependence on Events

If a change of $state(e)$ may affect $state(e')$ then we say e' has a *data dependence* on e and write $e \sqsubset_{data} e'$. Formally,

Definition 8. *For events e and e', $e \sqsubset_{data} e'$ iff $e <_\tau e'$ and one of the following holds:*

1. *$e < e'$, $type(e) = read$ and $stmt(e')$ uses $target(e)$ to compute $state(e')$;*
2. *$type(e) = write$, $type(e') = read$, $target(e) = target(e')$, and there is no other e'' with $e <_\tau e'' <_\tau e'$, $type(e'') = write$, and $target(e'') = target(e')$;*
3. *$e < e'$, $type(e') = read$, $stmt(e') \notin scope\,(stmt(e))$, and there exists a statement S in $scope\,(stmt(e))$ such that S can change the value of $target(e')$.*

The first case in this definition encodes the common data dependence. For example, for an assignment $x := E$, the write of x has data dependence on the reads generated by the evaluation of E. The second case in Definition 8 captures the interference dependence [13] in multithreaded programs, saying that a read depends on *the most recent* write of the same memory location. For instance, in Figure 4, if the observed execution is $C_1S_1S_3$ then the read of x at S_3 is data dependent on the most recent write of x at S_1. We treat lock release as a write on the lock and lock acquire as a read. The third case in Definition 8 is more intricate and relates to the relevant dependence in [10]. Assuming another execution of Figure 4, say $C_1S_2S_3$, no data dependence defined in cases *1* and *2* can be found in this run. However, the change of the value of the read of i at C_1 can potentially change the value of the read of x at S_3: if the value of i changes then C_1 may choose to execute the branch of S_1, resulting in a new write of x that may change the value of the read of x at S_3. Therefore, we say that the read of x at S_3 is data dependent on the read of i at C_1, as defined in case *3*. Note that although this dependence is caused by a control statement, it can *not* be caught by the control dependence; for example, the read of x at S_3 is *not* control dependent on the read of i at C_1 since $S_3 \notin scope(C_1)$. Aliasing information is needed to correctly compute dependence defined in case *3*, which one can obtain using any available techniques.

An important observation of Definition 8 is that there are no write-write, read-read, read-write data dependencies. Specifically, case *2* only considers the write-read data dependence, enforcing the read to depend upon only the latest write of the same variable. In other words, a write and the following reads of the same variable form an *atomic* block of events. This captures in a more general setting the work in [20].

4.4 Slicing Causality Using Relevance

When checking a trace τ against a property φ, not all the events in τ are relevant to φ; for example, to check dataraces on a variable x, accesses to other variables or function calls are irrelevant. Moreover, the *state* attributes of some relevant events may not be relevant; for example, the particular values written to or read from x for datarace (on x) detection. We next assume a generic *filtering function* that can be instantiated, usually automatically, to concrete filters depending upon the property φ under consideration:

Definition 9. *Let α: $\mathsf{Events} \nrightarrow \mathsf{Events}$ be a partial function, called a* ***filtering function****. The image of α, that is $\alpha(\mathsf{Events})$, is written more compactly Events_α; its elements are called* ***abstract relevant events****, or simply just* ***relevant events****. All thread start and exit events are relevant: $\alpha(e)$ defined whenever $type(e) = start$ or $type(e) = exit$.*

Let us assume an arbitrary but fixed property φ in what follows. Intuitively, $\alpha(e)$ is defined if and only if e is relevant to φ; if $\alpha(e)$ is defined, then $key(\alpha(e)) = key(e)$ for any attribute $key \neq state$, while $state(\alpha(e))$ is either undefined or equal to $state(e)$.

Definition 10. *Let $\alpha(\tau)$, written more compactly as τ_α, be the trace of relevant events achieved by applying α on events in τ. Let ξ_α denote ξ_{τ_α} for simplicity.*

This relevance-based abstraction plays a crucial role in increasing the predictive power of our analysis approach: in contrast to the concrete event set ξ, the corresponding abstract event set ξ_α allows more permutations of abstract events; instead of calculating permutations of ξ and then abstracting them into permutations of ξ_α like in traditional happen-before based approaches, we will calculate valid permutations of a *slice* of $\xi \cup \xi_\alpha$

that contains only events (directly or indirectly) relevant to φ. This slice is defined using the dependence on concrete and abstract events.

Definition 11. *All dependence relations are extended to abstract relevant events: If $e < / \sqsubset_{ctrl} / \sqsubset_{data}$ e' then also $\alpha(e) < / \sqsubset_{ctrl} / \sqsubset_{data}$ e', $e < / \sqsubset_{ctrl} / \sqsubset_{data}$ $\alpha(e')$, and $\alpha(e) < / \sqsubset_{ctrl} / \sqsubset_{data}$ $\alpha(e')$, whenever $\alpha(e)$ and/or $\alpha(e')$ is defined; $\sqsubset_{data}$ is extended only when $state(\alpha(e'))$ is defined.*

We next define a novel dependence relation, called *relevance dependence*, which is concerned with *potential* occurrences of relevant events. Consider Figure 4 again. Suppose that relevant events include writes of y and z. For the execution $C_1S_1S_3$, only one relevant event is observed, namely the write of z at S_3 (e'), which is not control dependent on the read of i generated at C_1 (e). Consider now another execution $C_1S_2S_3$; in addition to e', a new relevant event will be generated, namely the write of y at S_2, caused by the different choice made at C_1. Hence, a change of $state(e)$ *may affect the number of generated relevant events*. Formally, we define *relevance dependence* as follows:

Definition 12. *For $e \in \xi, e' \in \xi_\alpha$, we write $e \sqsubset_{rlvn} e'$ iff $e < e'$, $stmt(e') \notin scope(stmt(e))$, and there is a statement $S \in scope(stmt(e))$ that may generate a relevant event.*

Intuitively, if $e \sqsubset_{rlvn} e'$ then e' is not control dependent on e, but when $state(e)$ changes, some new relevant events may occur before e'. This may invalidate some permutations of ξ_α since valid permutations should preserve the *exact* number of relevant events.

Definition 13. *Let $\sqsubset$ be the relation $(\sqsubset_{data} \cup \sqsubset_{ctrl} \cup \sqsubset_{rlvn})^+$. If e and e' are concrete or relevant events such that $e \sqsubset e'$, then we say that e' **depends upon** e.*

Definition 14. *Let $\overline{\xi_\alpha} \subseteq \xi \cup \xi_\alpha$ be the* relevant slice *of events, extending ξ_α with events $e \in \xi$ such that $e \sqsubset e'$ for some $e' \in \xi_\alpha$. Let $\overline{\tau_\alpha}$ be the **abstract trace** of τ, i.e., the permutation of $\overline{\xi_\alpha}$ consistent with $<_\tau$.*

Intuitively, $\overline{\xi_\alpha}$ contains all the events that are directly or indirectly relevant to the property α. Our goal here is to define an appropriate notion of causal partial order on $\overline{\xi_\alpha}$ and then to show that any permutation consistent with it is sound. Recall that we fixed a trace τ; in what follows, τ' is used to refer to any arbitrary trace.

Definition 15. *Let $\prec^\tau \subseteq \overline{\xi_\alpha} \times \overline{\xi_\alpha}$ be the relation $(< \cup \sqsubset)^+$, which we call the **sliced causality** (or **sliced causal partial order**) of τ.*

From here on, by "causal partial order" we mean the sliced one. We next show that sliced causality is an instance of the parametric framework in Section 3.

Definition 16. *Let C_α: $\mathsf{Traces} \nrightarrow \mathcal{PO}(\mathsf{Events})$ be the partial function defined as $C_\alpha(\tau') = (\xi_{\tau'}, \prec^{\tau'})$ for each $\tau' \in \mathsf{Traces}$. Let $\mathcal{F}_\alpha \subseteq \mathsf{Traces}$ be the set of all possible abstract traces: for each $\tau_\mathcal{F} \in \mathcal{F}_\alpha$, there is some execution generating τ' such that $\overline{\tau'_\alpha} = \tau_\mathcal{F}$.*

Theorem 2. *C_α restricted to $\mathcal{F}_\alpha$ is total and a causality operator. That is, for any abstract trace $\tau_\mathcal{F} \in \mathcal{F}_\alpha$, each liniearization of $\prec^{\tau_\mathcal{F}}$ corresponds to some possible execution of the multi-threaded system.*

Proof (Sketch). The first condition of the causality operator definition can be easily verified. The more intricate part is to show that the trace obtained by permuting two consecutive independent events in an abstract trace is in $\mathcal{F}_\alpha$, that is, it corresponds to a possible execution. This is achieved by definning partial executions and feasible prefix traces, i.e., (abstract) traces corresponding to partial executions. The technical details of this proof can be found in the companion technical report.

We can therefore analyze the permutations of relevant events consistent with sliced causality to detect potential violations *without* re-executing the program.

5 Evaluation

We implemented a vector clock algorithm for computing sliced causality as part of jPredictor, a prototype tool for concurrency error detection of Java programs. To measure the effectiveness of sliced causality in contrast with more conventional happen-before causalities, we also implemented the algorithm in [19] for extracting happen-before causality from Java programs. Interested readers are referred to [3] for more details about the algorithm and the implementation of jPredictor. jPredictor has been evaluated on several concurrent programs. Two measurements were used during the evaluation to compare the sliced causality with the conventional happen-before causality, namely the size of the partial order and the prediction capability to detect data races. The results of our evaluation are shown in Table 1.

Benchmarks				Causality Size		Races	
Program	LOC	S. V.	Threads	H.B.	S.C.	S.C.	H.B.
Banking	150	10	11	18	2	1	1
Http-Server	170	2	7	22	2	2	1
Elevator	530	20	4	240	2	0	0
sor	600	42	4	21	8	0	0
tsp	1.1k	15	3	5	2	1	0
Daisy	1.5K	312	3	41	23	1	0
Raytracer	1.8k	4	4	7	3	1	1
SystemLogHandler	320	3	3	2	1	1	0
WebappLoader	3k	10	3	9	5	3	0

Table 1. Evaluation of sliced causality

The first part of Table 1 shows the benchmarks that we used, along with their size (LOC abbreviates "lines of code"), number of shared variables (S.V.), and number of threads created during their executions. Banking and Http-Server are two simple examples showing relatively classical concurrent bug patterns discussed in detail in [7]. Elevator, sor, and tsp come from [22]. Daisy is a small highly concurrent file system proposed to challenge and evaluate software verification tools. Raytracer is a program from the Java Grande benchmark suite, and SystemLogHandler and WebappLoader are two components of Tomcat 5.0.28. The property under verification is the datarace of the shared variable. The test cases used in experiments were manually generated using *fixed* inputs. More bugs could be found if more effective test generation techniques were employed, but that was not our objective here.

The second part of the table shows the coverage improvement of the analysis when using sliced causality versus the happen-before causality. A more precise measure of coverage would be the number of all the sound linearizations of the causal partial order; unfortunately, counting sound linearizations is a #P-complete problem [1], so it may be no easier than the problem of generating them. Note that a fully constrained partial order, that is, a total order, admits only one linearization, while a fully unconstrained partial order, that is, a set, admits an exponential number of linearizations. Extrapolating, even though it should not be taken as an absolute measure in all situations, we can say that the fewer causal dependencies a partial order has, the larger the number of sound permutations; moreover, we can also say that the number of linearizations is exponential in the number of unordered elements in the partial order. This simplistic and admittedly informal reasoning leads us to an important insight: any reduction in the number of causal dependencies may have a significant impact on coverage; in particular, a linear reduction of the number of causal dependencies can lead to an *exponential* increase in the coverage of the analysis. Since the total orders on the events of each thread are enforced by both sliced causality and happen-before, we only measure the causal dependencies due to direct inter-thread communication. Therefore, the following dependencies are counted, their number declared the *size of the causality*, and then used as a measurement metric: $e_1 \sqsubset e_2$, $\mathit{thread}(e_1) \neq \mathit{thread}(e_2)$, and there is no e_3 such that $e_1 \sqsubset e_3$ and $e_3 \sqsubset e_2$. Our experiments illustrate that sliced causality is significantly smaller than the conventional happen-before, indicating a magnificent increase in the number of covered potential executions.

The third part of Table 1 directly compares the effectiveness of race detection using sliced causality versus happen-before. The first column in this part is the number of races detected by sliced causality, while the second column gives the number of races detected by the standard, unsliced happen-before causality using the *same* execution traces. As expected, sliced causality is more effective in detecting dataraces, since it covers more potential runs. Even though, in theory, the standard happen-before technique may also be able to detect, through many executions of the system, the errors detected from one run using sliced causality, we were not able to find any of the races in some programs, e.g., in tsp and Tomcat, without enabling the sliced causality. Moreover, in these experiments, sliced causality did *not* produce any false alarms and, except for Tomcat, it found *all* the previously known dataraces. For Tomcat, two bugs were revealed from the detected dataraces, both of which had been submitted to and confirmed by the developers of Tomcat. More details can be found in [3].

6 Conclusion

We presented a parametric approach to happen-before causal partial orders, which facilitates defining and proving correctness of happen-before relations. Existing variants of happen-before causalities can be obtained as instances of this parametric framework. A novel causal partial order relation, called sliced causality, was also defined and shown correct within our parametric framework. Sliced causality employs static and dynamic analysis to filter out unnecessary dependencies on events in order to improve the coverage of analysis without losing soundness. Evaluation shows that sliced causality can significantly increase the coverage of concurrent program analysis and testing.

References

1. G. Brightwell and P. Winkler. Counting linear extensions is #p-complete. In *Annual ACM symposium on Theory of computing (STOC)*, 1991.
2. K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
3. F. Chen and G. Roşu. Predicting concurrency errors at runtime using sliced causality. Technical Report UIUCDCS-R-2005-2660, Department of Computer Science at UIUC, 2005.
4. F. Chen and G. Roşu. Parametric and sliced causality. Technical Report UIUCDCS-R-2007-2807, Department of Computer Science at UIUC, 2007.
5. F. Chen and G. Roşu. Parametric and termination-sensitive control dependence - extended abstract. In *International Static Analysis Symposium (SAS)*, 2006.
6. R. Cooper and K. Marzullo. Consistent detection of global predicates. In *ACM/ONR workshop on Parallel and distributed debugging (PADD)*, 1991.
7. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.
8. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
9. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGPLAN symposium on Principles of programming languages (POPL)*, 2005.
10. T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 1999.
11. H. J. Hoogeboom and G. Rozenberg. Dependence graphs. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, pages 43–67. World Scientific, 1995.
12. S. Horwitz and T. W. Reps. The use of program dependence graphs in software engineering. In *International Conference on Software Engineering (ICSE)*, 1992.
13. J. Krinke. Static slicing of threaded programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 1998.
14. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of ACM*, 21(7):558–565, 1978.
15. R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 1991.
16. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.
17. A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.
18. A. Sen and V. K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *Proceedings of the Seventh International Conference on Principles of Distributed Systems (OPODIS)*, 2003.
19. K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 2003.
20. K. Sen, G. Roşu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 2005.
21. S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient detection of global properties in distributed systems using partial-order methods. In *International Conference on Computer Aided Verification (CAV)*, 2000.
22. C. von Praun and T. R. Gross. Object race detection. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.