Protocol Testing in a Multi-Process Discrete-Event System with a Central Coordinator

Peter Donovan
peterdonovan@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Abstract

We propose a set of algorithms and techniques motivated by properties of a distributed coordination protocol, the Lingua Franca distributed runtime, which make existing techniques unsuitable. We demonstrate that if we use a simplistic coverage metric, pairwise ordering coverage within only the runtime's central coordinator (the RTI), we can obtain a signal that seems likely to be meaningful in the sense that it does saturate, even though we are unable to prove that 100 percent coverage has been obtained. We argue that for this system, and for any similar system that also uses a central coordinator and which enforces orderings between most pairs of events, the coverage metric used is meaningful. We show that efficient algorithms can be used to get high throughput of test runs and to efficiently work with large amounts of testing data, and we make a qualitative argument that the data obtained have been helpful for validating manually-maintained properties of the implementation.

CCS Concepts: • **Software and its engineering → Empirical software validation**.

Keywords: Concurrency, software testing, coordination protocols.

ACM Reference Format:
Peter Donovan. 2025. Protocol Testing in a Multi-Process Discrete-Event System with a Central Coordinator. In *Proceedings of Placeholder for Conference Name (N/A).* ACM, Berkeley, California, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

There exists an important class of systems which are "mostly synchronized" in the sense that, when run on a given input, nearly all pairs of trace events are ordered with respect to each other in the limit as the system is run for a long time. This tends to be true of discrete-event simulators in which no event r at logical time g_r (with respect to times in the simulated system) can be processed until we have processed all pending events r' at times $g_{r'}$ that occur sufficiently long before g_r. Although there are some exceptions, such as simulations that can be trivially decomposed into non-interacting parts, simulations that have unidirectional communication over an unbounded buffer without backpressure, or simulations that use relatively exotic techniques such as the "time warp" proposed by Jefferson, [2] it is not unusual for discrete-event simulations to have this property. In particular, the Lingua Franca federated runtime, which is the context in which the techniques described here are evaluated, is "mostly synchronized."

```
s ← []
x ← InitialState
while ¬IsTerminating(x) do
    r ← ChooseEnabledEventToExecute(x)
    x ← ProcessEvent(e, x)
    s ← Cons(r, s)
end while
```

Figure 1. An example of the framework in which protocol testing is often performed. This figure is adapted from [10].

The purpose of this work is to describe how to explore the possible sequences of trace events in such a system.

2 Related Work

The most closely related work relates specifically to concurrency testing that explores possible orderings of potentially noncommuting events. There are multiple recent works in this area, where the objective is to find a good behavior for the `ChooseEnabledEventToExecute` routine, shown in Figure 1, which models the nondeterministic component of the operational semantics.

Work that is based on this conceptual framework is able to efficiently explore the partial ordering space by simulating multiple processes within a single (virtual) machine, and sometimes even within a single thread. This approach is attractive because it eliminates much of the syscall and synchronization overhead associated with multi-process programs, while remaining embarrassingly parallel because entirely separate test runs can be executed simultaneously.

For example, Bita is a concurrency testing tool that works for Akka programs.[8] In its model, interesting events are events when a message arrives in an actor's mailbox, because after that, the order in which the mailbox is processed

N/A, December 2023, Berkeley, CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

is deterministic. It works by selecting the desired next message from among the messages that are in-flight with respect to the execution model (that is, enabled to be placed in the mailbox). The work described in this report is similar to that described by Tasharofi et al. in that it focuses on transposing specific pairs of events, but it is simpler in that it does not have any way to check if the schedules it produces are feasible, and it is not specific to Akka.

Morpheus, which targets BEAM, the Erlang virtual machine, also fits into this paradigm.[9] Morpheus uses a similar method using virtual inboxes, and its implementation is tightly integrated with Erlang-specific interprocess communication primitives.

Another common theme is the need to reduce the search space by considering traces that have the same partial ordering of events to be equivalent. For example, vector clocks can be used to track at run-time which new total orderings of events actually lead to new partial orderings.[5] The importance of distinguishing partial orderings rather than, for example, a total ordering derived from a central clock, has influenced later work, including Bita[8]. Because all messages in Lingua Franca's federated execution flow through a central coordinator, the RTI, and because it can be expensive to instrument all accesses to shared memory, the work described in this report does not take advantage of these techniques, instead focusing on events within the RTI and considering them as totally ordered; it is possible, however, that incorporating fine-grained vector clocks would yield better results.

The relatively small part of this work which checks whether various predicates hold over all observed traces has some similarities to run-time monitoring, where predicates of a specific form are defined such that they can be efficiently checked.[6] In this work, however, predicates need to be efficiently checkable offline, using data that can be stored compactly, which is a different constraint from needing to be efficiently checkable at run-time, in deployment.

A high-level observation by Lopez et al. is that protocol-testing tools of this kind should not be confused with race detectors, which focus on low-level races; they present a taxonomy which makes this distinction clear.[3] As a result, although the literature has extensive work on effective race detection for low-level programs that do not use a virtual machine or any one fixed set of APIs, this work is mostly for shared-memory programs that have data races, not protocol-level bugs.[7][4]

3 Definitions

Throughout, "time" refers to values read from a monotonic, strictly increasing clock or counter that is accessible to all processes.

Trace events are observable events during the execution of a (possibly multi-process) program, with timestamps and possibly other attributes associated with them.

$<_s$ refers the observed ordering of trace events within the same process as they occur in the trace s. More precisely, $r_0 <_s r_1$ iff r_1 is not observed to occur earlier than r_0 in the trace s. A consequence of this definition is that if either r_0 or r_1 does not appear in s, then trivially $r_0 <_s r_1$ and $r_1 <_s r_0$; however, when restricted to events that appear in all traces, $<_s$ is total for all s because we use a central clock, consider only events in the central RTI, and make the simplifying assumption that any event may access any part of shared memory.

$\prec$ refers to the relation $\bigcap_s <_s$, where the variable s ranges over all feasible non-error traces. Intuitively, it is the ordering relation enforced by the runtime implementation. It is an abuse of notation because $\prec$ may not be transitive. In particular, we may have that $r_0 \prec r_1$, and $r_1 \prec r_2$, but there exist traces s in which r_1 is never observed, and $r_2 <_s r_0$.

A relation $\prec$ is k-total if, for any tracepoint r_0, $|\{r_1 \mid r_0 \not\prec r_1 \wedge r_1 \not\prec r_0\}| \leq k$. This formalizes the notion of being "mostly synchronized": If the happens-before relation $\prec$ of a program P is k-total, where k is small relative to the length of most non-error traces of P, then P is mostly synchronized. It is also a generalization of totalness in the sense that a relation that is 0-total is total.

Within a given node, there is some set U of trace events which are unordered with respect to each other. Formally, $U = \{(r_0, r_1) \mid r_0 \not\prec r_1 \wedge r_1 \not\prec r_0\}$. Given some set S of non-error traces which are empirically observed to be feasible, $\tilde{U} := \{(r_0, r_1) \mid (\exists s \in S. r_0 <_s r_1) \wedge (\exists s' \in S. r_1 <_{s'} r_0)\}$. We have that $\tilde{U}$, our empirical approximation of U, is a subset of U.

Define pairwise transposition coverage as $|\tilde{U}|/|U|$, a number in the interval $[0, 1]$. The objective of fuzzing the SUT is to aggregate observations until $|\tilde{U}|/|U| \approx 1$. This definition of coverage is motivated by the following facts:

1. In a mostly synchronized system in which most activity is mediated by a central coordinator, the events that occur in the central coordinator can be used as a proxy for all activity in the system.
2. Sometimes, it may be too difficult to attain a partial ordering coverage which is large enough to be distinguishable from zero. Partial ordering coverage, as might be measured using vector clocks, is the same as total ordering coverage when observations are restricted to a single node. Even in a system whose enforced precedence relation $\prec$ is k-total for some small $k > 0$, the number of possible observed (partial) orderings $<_s$ is exponential in the length of a trace. Tasharofi et al. have pointed out that even if there is no notion of 100% coverage, a metric can still be useful for comparing different sets of schedules,[8] but nonetheless it

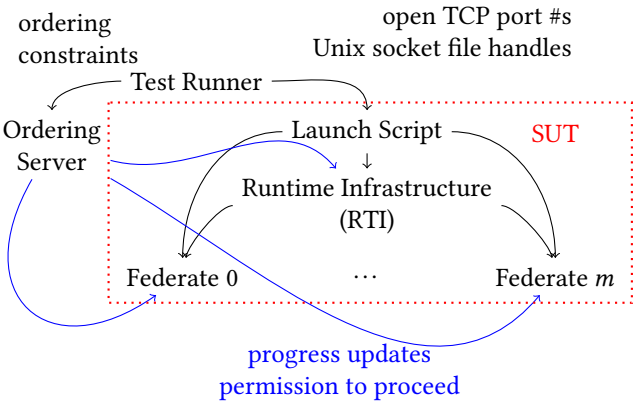

Figure 2. Communications between the test runner, the ordering server, and the SUT. The SUT includes the launch script, the RTI (the central coordinator), and the federates.

can be helpful to be able to identify whether coverage has saturated.

3. It is often the case that bugs arise from the order in which only a small number of events occur; this fact is well-known and is referred to as the small-scope hypothesis.[10] The engineer who focuses on pairwise transposition coverage hopes that the wrong ordering of only *two* events is sufficient to reveal a bug.
4. When one's goal is not to find bugs but to empirically understand a system, or to empirically check that its specification is not out of date, it may be conceptually easiest to focus on relationships between only two events at a time. For example, if one's understanding of the system says that trace event r_0 causes trace event r_1, then we should expect to never observe r_1 to occur before r_0 with respect to a monotonic system clock that is shared by all processes.

4 Methods

4.1 System Architecture

A run of the testing tool progresses through four states: An "initial" state that stores settings, such as the limit on the number of tests to run concurrently and the saving frequency; a "compiled" state which stores paths to the compiled executables; a "collected metadata" state which includes details about the tracepoints, which later are referred to only by numeric IDs; and an "testing" state. Transitions correspond to checkpoints where intermediate results are saved to disk.

The system repeatedly performs an enumerative search over the pairwise orderings that have not yet been observed and attempts to trigger them. In principle, it may be possible to complete it using fewer than $\Theta(n^2)$ test runs if many new pairwise orderings are observed in one run.

It may seem that the obvious way to perform better than $O(n^2)$ in k-total partial orderings $\prec$ is to avoid trying to transpose trace events that are $2k$ apart in the original trace after we have already failed to transpose any two elements that are k apart. However, we have only heuristic reasons to believe that adding a single constraint between two trace events r_0 and r_1 can transpose them if a non-error trace that transposes them exists. This is why there is no formal guarantee that no coverage would be lost by performing that optimization, any more than there is any guarantee – not even a probabilistic one – that the coverage obtained is close to 100 percent.[1]

Indeed, the current implementation of the system cannot determine whether the introduction of an ordering constraint between two trace events, combined with a feasible prefix of a trace, combine with $\prec$ to give an unsatisfiable set of constraints, i.e., cause a deadlock. Consequently deadlocks are handled using timeouts. Sound and well-known deadlock detection algorithms exist because deadlock states are a stable property,[1] but they may limit the practicality of generalizing this tool beyond its original setting due to the manual effort required to instrument the code.[2]

4.2 Streaming-Transpositions Algorithm

The streaming-transpositions algorithm is a very simple algorithm the efficiency of which depends on the SUT being mostly synchronized and on a total ordering of the trace events of interest having been observed. Its purpose is to determine, for each pair of trace events r_0 and r_1, whether a non-error trace has been observed that refutes that $r_0 \prec r_1$.

Suppose we observe one feasible non-error trace, which we call the "original trace," that totally orders all trace events via an ordering relation $<_{og}$. At this point we are already halfway done, having refuted exactly half of the pairwise ordering relationships $r_0 \prec r_1$ that we could in principle ask about. Then, as we collect more traces, for each trace event r_0, we record all trace events that were initially observed to occur after r that have later been observed to occur before r.

When $\prec$ is k-total, this algorithm is space-efficient, using only nk space, where n is the number of tracepoints in the original trace. This is because, for each tracepoint r, we only have to store up to k trace events that have been observed to occur after r after initially being observed to occur before r.

Additionally, the algorithm can be made time-efficient if k is known. This is because, in that case, two tracepoints that can appear in either order with respect to each other cannot be observed to have more than $2k$ tracepoints in between them. Therefore, when we observe a new trace, we

[1] This limitation results from the lack of any tight integration with any specific VM or API, and we are aware that when an instrumented VM or API is available, it is not representative of the state of the art.

[2] Of course, code does not have this problem if it restricts itself to instrumented communication APIs or if it is written in a language that makes interprocess communication syntactically explicit.

can perform an update in nk time by, for each tracepoint r, searching only the following $2k$ tracepoints to try to find tracepoints that originally occurred earlier than r. Even if k is not known precisely, this optimization can be made while retaining the property that we will never incorrectly refute that $r_0 \prec r_1$.

This algorithm is sensitive to individual observations in the sense that it cannot distinguish between an ordering between two tracepoints that is only observed once and an ordering of two tracepoints that is observed many times. If we suspect that the SUT has a bug such that, very rarely, it can exhibit an error trace with an ordering of tracepoints that "should be impossible" without returning a nonzero exit code, then in the limit as we collect many traces there may be a high probability that we incorrectly refute that $r_0 \prec r_1$. This effect might be attenuated by re-starting the streaming algorithm every p test runs with a fresh original trace. By doing this it should be possible to obtain an estimate of the probability of an observation that refutes $r_0 \prec r_1$ on the basis of an estimate of the probability that such an observation occurs at least once in a given sequence of p runs.

4.3 Upper-Bounding the Unordered Pairs of Tracepoints

We wish to overapproximate $\prec$ using rules of the following form:

⟨*Rule*⟩ ::= { event: ⟨*predicate*⟩,
preceding_event: ⟨*binaryrelation*⟩ }

⟨*binaryrelation*⟩ ::= '`TagPlusDelay2FedEquals`'
| '`TagLessThan`'
| '`FederateEquals`'
| '`IsFirst`' ⟨*binaryrelation*⟩
| '`IsFirstForFederate`' ⟨*binaryrelation*⟩
| '`And`' ⟨*binaryrelation*⟩*
| '`Or`' ⟨*binaryrelation*⟩*
| '`Unary`' ⟨*predicate*⟩

⟨*predicate*⟩ ::= '`EventIs`' ⟨*EventKind*⟩
| '`IsFirst`' ⟨*predicate*⟩
| '`And`' ⟨*predicate*⟩
| '`Or`' ⟨*predicate*⟩
| '`Not`' ⟨*predicate*⟩
| '`BoundBinary`' ⟨*Event*⟩, ⟨*binaryrelation*⟩

Some types of binary relations, and details of the definition of `Event` and `EventKind`, are elided for brevity. The idea is that there is some domain-specific set of predicates that identify the events to which the rule applies, and then there is some domain-specific set of binary relations that identify which events must be predecessors of that event. The `Unary` binary relation ignores its first argument. The `BoundBinary` predicate corresponds to a binary relation, the first argument of which is bound to some specific event in a trace; the predicate `IsFirst` identifies pseudo-events that mark the first event that satisfies some predicate. The binary relation `IsFirst` is similar, but its associated predicate is equivalent to its associated binary relation, with the first argument bound to a specific event. Other predicate types, like `And`, behave as you would expect, while others, such as `FederateEquals`, are domain-specific and may depend on metadata provided about the communication topology of the SUT.[3]

4.3.1 Computing the Transitive Closure.

We assume that there is a set of rules which each can accept a pair of trace events r_0 and r_1 and determine whether $r_0 \prec r_1$.

We further assume that there is a set of trace events which are guaranteed to occur in all non-error traces. In a mostly synchronized system similar to the one in question, where distinct traces are distinct runs of the same integration test, the majority of events that are ever observed will be in this set. We also assume that it is reasonable to guess that an event f that is observed in all traces triggered by the testing software is indeed guaranteed to occur in all non-error traces, in which case we write $\text{Always}(r)$.

The following rule expresses the sense in which $\prec$ is transitive:

$$r_0 \prec r_1 \wedge r_1 \prec r_2 \wedge \text{Always}(r_1) \implies r_0 \prec r_2 \qquad (1)$$

A consequence of this is that $\prec$ is transitive when restricted to events that are guaranteed to occur in all non-error traces.

Predicates written using this framework are sufficiently general to express a reasonable notion of causality. In particular, if an event r can be caused by any element of some set of events $\tilde{R}$, then we can pre-process traces by inserting a "pseudo-event" $\tilde{r}$ that corresponds to the first element of $\tilde{R}$ that is observed to occur. Then, assuming that $\text{Always}(\tilde{r})$ and $\text{Always}(r)$, we can write $\tilde{r} \prec r$ to express that r cannot happen unless an event that causes r happens first. This pre-processing is not necessary when $|\tilde{R}| = 1$.

These predicates can also be used to specify when a process needs to be ready to handle an event. When used in this way, $r_0 \prec r_1$ lets us use r_1 as a proxy for a state of a process in which it cannot be expected to successfully handle the event r_1.

The following section presents an argument that in addition to being useful, predicates of this form have the desirable property that their consequences can be efficiently computed and represented in a mostly synchronized system.

4.3.2 Algorithm.

Suppose we already know which trace events are ordered by $\prec$ with respect to each other, and we wish to determine the consequences of rule (1).

Since $\prec \subseteq <_{og}$, where $<_{og}$ is the original non-error trace that we observed, the obvious way to do this is to determine

[3] Currently these predicates are written and implemented only as data structures in `axioms.rs`.

the set of all predecessors of the least trace event r_0 with respect to $<_{og}$, and then the second-least trace event r_1, and so on. This works because for a given trace event r_j which is jth with respect to $<_{og}$, given that our other rules will only mark earlier events $r_{\ell_0}, \ldots, r_{\ell_d}$ with $\ell_i < j$ as being earlier with respect to $\prec$ than r_j, the set of all r' such that $r' \prec r_j$ is simply $\left(\bigcup_{\ell \in \{\ell_0,\ldots,\ell_d\}} \{r_i \mid r_i \prec r_\ell \wedge \text{Always}(r_\ell)\}\right) \cup \{r_{\ell_0}, \ldots, r_{\ell_d}\}$.

The only problem is that the sets $\{r_i \mid r_i \prec r_\ell\}$ might have size that grows linearly with respect to the length of the trace. Therefore, if we store these sets explicitly in memory as we process the trace events in increasing order with respect to $<_{og}$, then we will use quadratic time and space.

When $\prec$ is k-total, we can circumvent this problem by instead storing the sets $\{r_i \mid (r_i \not\prec r_\ell) \wedge (i < \ell)\}$, which will have size at most k because they only include trace events that are unordered with respect to r_ℓ. Then, to compute $\{r_i \mid (r_i \not\prec r_j) \wedge (i < j)\}$, in terms of its predecessors $\{\ell_0, \ldots, \ell_d\}$, we compute

$\bigcap_{\ell \in \{\ell_0,\ldots,\ell_d\}} \left(\{r_i \mid (r_i \not\prec r_\ell) \wedge (i < \ell)\} \cup \{r_{\ell+1}, \ldots, r_{j-1}\}\right)$

because that is the complement of

$\left(\bigcup_{\ell \in \{\ell_0,\ldots,\ell_d\}} \{r_i \mid r_i \prec r_\ell \wedge \text{Always}(r_\ell)\}\right) \cup \{r_{\ell_0}, \ldots, r_{\ell_d}\}$

intersected with the set of trace events preceding r_j with respect to $<_{og}$.

This algorithm will use only nk space, as desired. Sadly, it will only use less than $\Theta(n^2)$ time if the trace events marked as immediate predecessors of an event r by the rules are guaranteed to be located close to r in the trace, which is not true in general.

4.4 The Ordering Server

One naive approach to exploring the possible interleavings is simply to run the software under test many times, under identical conditions, until it deadlocks or crashes with an assertion error. This technique has zero overhead because no attempt is made to manipulate the order in which events occur in the system, and as a result, it can be a useful way to obtain positive examples of feasible traces. However, it does not yield much confidence that traces that are not observed cannot occur.

A possible improvement over this approach is to insert random sleeps in the SUT. We experimented with this idea and rejected it because it incurs excessive syscall overhead. Furthermore, we did not expect it to yield reproducible results because, in a system in which sleeps are on a granularity of milliseconds, it is too expensive to sleep for long enough at enough distinct program points to force the tracepoints to occur in a desired order.

We propose instead to select a small number of pairwise ordering constraints – perhaps only a single constraint – to enforce within any given run.

4.4.1 Assumptions. For a general non-atomic trace event, it would be necessary to record the physical start time and end time of the event using the monotonic system clock[4] in order to determine whether it has been observed to occur strictly earlier or later than other trace events. In this case, two events that occur in overlapping time intervals would be unordered with respect to each other. This poses a problem for the `streaming-transpositions` algorithm, which assumes that $<_{og}$ is a total order.

Fortunately, we can avoid this complication and totally order trace events using a single timestamp per event if the following assumption holds:

Input-tracepoint-effect assumption: All inputs to the event are received before the tracepoint, and all effects of the event are made visible to other threads or processes after the tracepoint.

The causal notions of "inputs" and "effects" which appear in this statement must be interpreted in the context of a formal notion of causality in which we can only refute that event r_1 causally depends on event r_0 by exhibiting a realizable non-error trace in which the effects of r_0 are realized only after the inputs of r_1 are received. Then, if the input-tracepoint-effect assumption holds, then we cannot incorrectly refute that r_1 causally depends on event r_0.

Note that any definitions of "inputs" and "effects" are acceptable as long as they are consistent with our notion of causality. For example, r_0 might have a message over the network as its input and a value stored in memory (where it is visible to other threads) as its effect. The event r_1 might correspond to reading a value from shared memory (as its input) and then sending a message to another process. The causal relationship that we posit might be that r_0 causes r_1 by allowing r_1 to read the update via shared memory. In this case, it would not be reasonable to argue that a change in the state of the TCP socket must be included formally in the effects of r_0, even though r_0 does indeed remove some bytes from the TCP socket, because that "effect" is not part of the mechanism by which we believe r_0 is ordered with respect to r_1.

The input-tracepoint-effect assumption is useful because it simplifies the process of instrumenting the code with tracepoints (such that only one timestamp need be collected), although it is unavoidable that care is required in placing tracepoints and in ensuring that memory barriers at tracepoints are respected.

4.5 Data Compression and Storage

4.5.1 Algorithm for Compactly Storing Entire Traces. The ability to store entire traces is required in order to enable exploratory data analysis, which engineers can use to manually lift specifications from an implementation. Furthermore, it is likely to be practical without running out of disk space because of the throughput limitations imposed by the syscall

[4] Although leap-seconds occur on some system clocks, Unix does expose a monotonic clock.

overhead associated setting up and using TCP sockets for interprocess communication.

The naive approach is to obtain an original trace that likely contains most of the important trace events that will every be observed, and to number the trace events in increasing order with respect to $<_{og}$, the ordering relation associated with that particular concrete trace. Then subsequent traces can simply be represented as sequences s of integers that correspond to these trace events.

If we know with certainty that $\prec$ is k-total, then we can make the following optimization: the jth integer, s_j is not the absolute number of the jth trace event observed; instead, the s_j is the difference between j and the number of the jth trace event observed. Then $|s_j| \leq 2k$, which allows s_j to be represented using fewer bits.

Using fewer bits may not save a very large number of space, but storing s_j as a difference from j can in principle enable even more compression when combined with hash-consing. Suppose the sequence s is represented not as a single array, but as a binary tree whose leaves are subsequences of s. Then, if two distinct traces are identical for the first half of all trace events, they can share a common left subtree. Furthermore, the benefit of saving s_j as a difference in this context is that sharing is possible in principle between different subsequences of s. We can even let entirely distinct tests, producing distinct families of traces, sharing the same repository of hash-consed subtrees, and optimistically hope that there will be a large amount of sharing because, of most traces are similar to the first one observed, many subtrees will be similar to each other, with long sequences of zeros.

In a highly idealized scenario in which we manipulate the timing of one tracepoint at a time and the tracepoints that we do not explicitly manipulate happen in the same order, we could even represent all observed traces in space proportional to $kn \log_2 n$, where $\prec$ is k-total and the traces have length n, and where $\log n$ is the length of the spine of the new tree resulting from the immutable update.

Sadly, the assumption that tracepoints that are not manipulated usually happen in the same order is quite strong. A strong evaluation of the amount of sharing that occurs in practice is left for future work.

4.6 Incremental and Parallel Data Representation

If the amount of data written to disk is proportional to the length of time that the testing software has been running, all time spent will in the limit be spent on writing results to disk unless the save interval increases at least linearly with the amount of data aggregated. Therefore, in order to save results reasonably frequently, it is necessary for every part of the persistent data representation to be fully incremental in the form of deltas.

This means that only traces observed since the last write to disk are written to disk again. The streaming-transpositions data structure is also adapted to be incremental by storing old observations and new observations in separate hash sets. After a save, the hash set of new observations is merged into the set of old observations using the same merge operation that is used to parallelize the streaming-transpositions algorithm, and empty sets of new observations are then initialized.

The sequence of deltas is obtained from a linked list of small files, where each file contains the path to its parent in the list as well as the paths to deltas corresponding to each test. Once the linked list of small files has been recovered, parallelism comes from deserializing all deltas for all tests over all time in parallel. Some deltas are much larger than others because some tests produce much longer traces than others and because much more data is initially accumulated in the streaming-transpositions algorithm near the beginning than near the end. However, the underlying scheduler used by `rayon`, the parallelism library, load-balances automatically via work-stealing.

4.7 Fault Tolerance

The need to attain high throughput is directly in tension with the need to mitigate spurious test failures due to overutilization of operating system resources, such as TCP sockets.

The problem of distinguishing spurious test failures of this kind from meaningful test failures is left to future work. So, if we accept the presence of spurious test failures, the problem reduces to attaining the highest throughput of non-failures that is possible without creating unrecoverable errors in the test framework itself. This is not trivial to accomplish because by design, the software under test is chaotic (because low-probability traces in its execution are being explored), and also tightly coupled with the ordering server through Unix socket communication.

Incremental data storage ensures that it is practical to save intermediate results frequently. Therefore, it is always possible to re-start the system when it crashes, perhaps after manually deleting the files saved most recently before the crash. However, this is seldom necessary because exceptions raised in the testing software are caught at thread boundaries and ignored. Because exceptions that occur in a critical section (while operating on a shared data structure) mark the data structure as invalid, we believe that it is safe to catch these exceptions without the risk of corrupted data. Furthermore, deadlocks that can occur due to bugs in the testing software are resolved by simply cancelling all asynchronous tasks periodically when it is time to save all updates to disk.

5 Evaluation

5.1 Throughput

Testing throughput increased markedly after modifying the runtime to eliminate sleeps. This modification seemed acceptable because the correctness of this version of the Lingua Franca federated runtime is documented to not depend on real-time behavior.

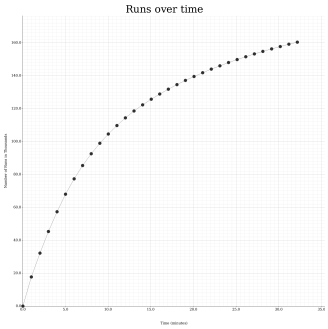

Figure 3. Test throughput over time.

Figure 3 shows the number of test runs completed as a function of time.

Initially the throughput easily exceeds 300 test runs per second, with most CPU activity appearing to consist of syscalls, but throughput decreases dramatically over time.[5]

5.2 Error Rate

The error rate averages less than 3 percent for all tests. Manual inspection of the error streams of the tests shows that many of the errors are communication-related (e.g., "broken pipe"), indicating that they could be caused by excessive operating system load, while many errors do appear to indicate real bugs (e.g., "segmentation fault").[6]

5.3 Coverage

To evaluate the usefulness of the ordering server, we compared it against a baseline that was identical except that the ordering server was disabled (it was changed to enforce zero pairwise orderings instead of one pairwise ordering).

The results of this comparison are shown in Figure 4. It appears clear that the ordering server has some effect because in all but one test, the score is greater than or approximately equal to zero, indicating that greater coverage was obtained by enabling the ordering server. Note that changing which value is in the numerator or denominator of the geometric mean changes only the sign and not the magnitude of its logarithm, so this is a symmetric comparison.

Based on the results of Figure 5, we can see that some but not all tests saturate in coverage within a few thousand runs.

5.4 Usefulness for Characterizing the SUT

The intent of 4.3 was to overcome the difficulty mentioned by Tasharofi et al. of determining "what 100% means" [8]. I did not make sufficient progress to obtain useful bounds on the number of pairs per test that are unordered with respect

[5]More investigation is required. If this is because of bad performance scaling as traces are accumulated, then this is an engineering mistake that should be fixed.

[6]We should find out how many false positives there are and find out how to detect and eliminate false positives. This was not pursued only because of lack of time.

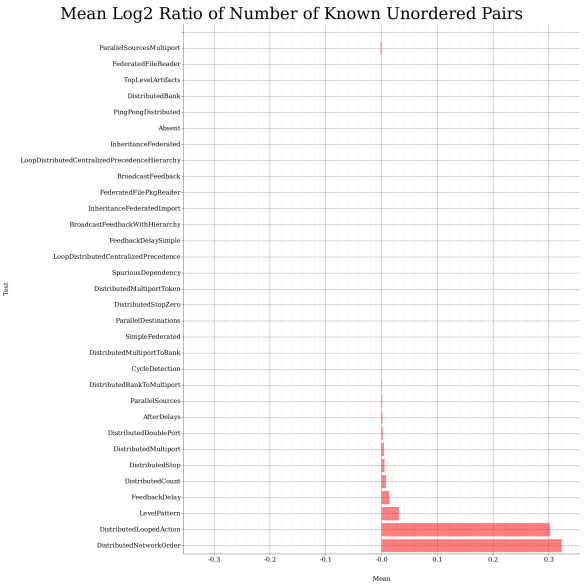

Figure 4. Coverage relative to the baseline, expressed as a logarithm-base-2 of the geometric mean of the ratio of the number of unordered (w.r.t. $\prec$) pairs observed. Tracepoints were excluded from this measurement if, either with the ordering server enabled or with it disabled, they were not observed to be unordered with any other tracepoints.

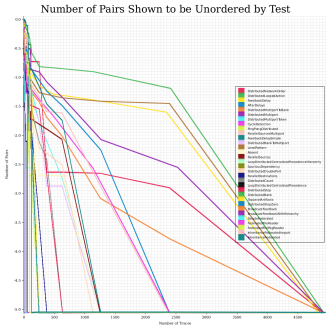

Figure 5. Proportion of pairs that are shown to be unordered after 5000 runs that are not yet shown to be unordered after k runs.

to each other. However, the precedence rules defined did help to validate the testing system, where counterexamples to the rules were used to find two bugs in the testing tool (one involving misinterpretation of sentinel tracepoint IDs, and another involving incorporation of irrelevant information in some hashes).[7] Conversely, the data collected from testing helped to find counterexamples to some of the rules which I, as a maintainer of the SUT, had previously believed to be true. This includes counterexamples that, in a given

[7]The bug involving sentinel tracepoint IDs means that some results that I reported at the poster presentation were not valid; this bug was discovered very late.

test, only manifest a few percent of the time, even under instrumentation, indicating that they might not have been found using a more minimalist testing setup.

6 Conclusion

The work discussed here is largely still in-progress. However, early results indicate that high coverage may be achievable using integration tests that are intended to be complex enough to reveal nontrivial bugs. They also suggest that the data collected using this tool may be useful for validating assumptions about the implementation of a coordination protocol.

References

[1] K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems* 3, 1 (Feb. 1985), 63–75. https://doi.org/10.1145/214451.214456

[2] David Jefferson. 1987. Distributed Simulation and the Time Warp Operating System. (1987).

[3] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. 2018. A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs. http://arxiv.org/abs/1706.07372 arXiv:1706.07372 [cs].

[4] Koushik Sen. [n. d.]. Race Directed Random Testing of Concurrent Programs. ([n. d.]).

[5] Koushik Sen and Gul Agha. 2006. Automated Systematic Testing of Open Distributed Programs. In *Fundamental Approaches to Software Engineering*, Luciano Baresi and Reiko Heckel (Eds.). Vol. 3922. Springer Berlin Heidelberg, Berlin, Heidelberg, 339–356. https://doi.org/10.1007/11693017_25 Series Title: Lecture Notes in Computer Science.

[6] K. Sen, A. Vardhan, G. Agha, and G. Rosu. 2004. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings. 26th International Conference on Software Engineering*. 418–427. https://doi.org/10.1109/ICSE.2004.1317464 ISSN: 0270-5257.

[7] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM, New York New York USA, 62–71. https://doi.org/10.1145/1791194.1791203

[8] Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph Johnson. 2013. Bita: Coverage-guided, automatic testing of actor programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Silicon Valley, CA, USA, 114–124. https://doi.org/10.1109/ASE.2013.6693072

[9] Xinhao Yuan and Junfeng Yang. 2020. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, 1141–1156. https://doi.org/10.1145/3373376.3378484

[10] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. Partial Order Aware Concurrency Sampling. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 317–335. https://doi.org/10.1007/978-3-319-96142-2_20