TraceLinking Implementations with Their Verified Designs

FINN HACKETT, University of British Columbia, Canada
IVAN BESCHASTNIKH, University of British Columbia, Canada

An important correctness gap exists between formally verifiable distributed system designs and their implementations. Recently proposed work bridges this gap by automatically extracting, or compiling, an implementation from the formally-verified design. The runtime behavior of this compiled implementation, however, may deviate from its design. For example, the compiler may contain bugs, the design may make incorrect assumptions about the deployment environment, or the implementation might be misconfigured.

In this paper we develop TraceLink, a methodology to detect such deviations through *trace validation*. TraceLink maps traces, that capture an execution's behavior, to the corresponding formal design. Unlike previous work on trace validation, our approach is completely automated.

We implement TraceLink for PGo, a compiler from Modular PlusCal to both TLA$^+$ and Go. We present a formal semantics for interpreting execution traces as TLA$^+$, along with a templatization strategy to minimize the size of the TLA$^+$ tracing specification. We also present a novel trace path validation strategy, called *sidestep*, which detects bugs faster and with little additional overhead.

We evaluated TraceLink on several distributed systems, including an MPCal implementation of a Raft key-value store. Our evaluation demonstrates that TraceLink is able to find 9 previously undetected and diverse bugs in PGo's TCB, including a bug in the PGo compiler itself. We also show the effectiveness of the templatization approach and the sidestep path validation strategy.

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

Additional Key Words and Phrases: Distributed Systems, TLA+, Modular PlusCal

ACM Reference Format:
Finn Hackett and Ivan Beschastnikh. 2025. TraceLinking Implementations with Their Verified Designs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 350 (October 2025), 33 pages. https://doi.org/10.1145/3763128

1 INTRODUCTION

Distributed systems are challenging to design and to build. Recent work has proposed a variety of approaches to help establish the correctness of distributed system designs and corresponding implementations. One line of work extracts, or compiles, a system implementation directly from a formally-checked distributed system design [Hackett et al. 2023a; Hawblitzel et al. 2017; Hristov and Bieniusa 2024; Wilcox et al. 2015]. With these tools, manual inconsistencies in the translation from model to implementation are unlikely, because human error is eliminated from the majority of the translation process.

However, these code extraction and compilation tools are themselves unverified. So, even though the design is checked, the output implementation may deviate from the design [Chen et al. 2020; Fonseca et al. 2017; Yang et al. 2011]. Moreover, the extracted implementation cannot be used as is. It needs additional configuration code. And, it may need to be directly modified to include important (but potentially buggy) optimizations that are missing from the formal model. Finally, the system logic is just one aspect that the model must describe. All models of distributed systems

Authors' addresses: Finn Hackett, University of British Columbia, Vancouver, Canada, fhackett@cs.ubc.ca; Ivan Beschastnikh, University of British Columbia, Vancouver, Canada, bestchai@cs.ubc.ca.

ACM 2475-1421/2025/10-ART350
https://doi.org/10.1145/3763128

must also capture the environment. But, there is currently no way to determine if the model's version of the environment accurately reflects the environment in which the implementation is deployed.

We present *TraceLink*, an approach to validating automatically-compiled implementations. TraceLink automatically maps a trace of a distributed system implementation to a formal model. TraceLink is an example of *trace validation* [Cirstea et al. 2024; Howard et al. 2025], a bug-finding technique in the broad space of runtime verification [Bartocci et al. 2018; Falcone et al. 2013]. In the context of TLA$^+$, trace validation has so far been done manually [Cirstea et al. 2024].

Trace validation depends on capturing a trace of implementation state and events [Reger and Havelund 2016]. These steps are then ingested into the TLA$^+$ tooling [Kuppe et al. 2019; Yu et al. 1999] and corresponded to parts of the formal model with an event-by-event relation that matches the two. This comparison can pinpoint specific differences between the abstract implementation traces implied by the system's formal model, and concrete execution traces recorded at runtime. Prior work on trace validation for TLA$^+$ [Cirstea et al. 2024; Howard et al. 2025], however, requires significant manual work: the implementation must be instrumented with sufficient logging statements, and each logged event type must be manually related to events in the model. By contrast, TraceLink is a push-button approach.

We implement TraceLink for PGo [Hackett et al. 2023a], a source-to-source compiler that compiles formal models in Modular PlusCal (MPCal) into TLA$^+$ models that can be formally checked, and simultaneously into runnable GoLang implementations. TraceLink achieves push-button trace validation thanks to two key insights: (1) At the implementation level, TraceLink hooks into PGo-generated systems to include consistent and general-purpose instrumentation of events and state changes. (2) At the specification level, TraceLink introspects the source model and a set of collected traces, generating all the necessary setup code a developer would need to begin validating their system's traces with the TLA$^+$ model checker [Kuppe et al. 2019] against any property supported by TLA$^+$.

TraceLink also implements causal tracing [Fidge 1988; Mattern 1989], and can consider multiple causally-allowed interpretations of a single implementation trace during validation. TraceLink has a small trusted computing base and, as we will show in Section 9, it uncovered a variety of bugs across all of PGo's trusted components, including the PGo compiler.

In summary, we make the following contributions:

(1) We present a formal trace semantics to automate the mapping of implementation-level traces to a formal MPCal model, and we present a templatization scheme that makes it possible to translate long implementation traces into a manageable amount of TLA$^+$.

(2) We describe the design of TraceLink for the PGo toolchain. This includes auto-instrumentation of Go code, and extended trace validation logic that takes advantage of vector clocks. We also introduce a novel *sidestep* property, which is an interpretation of vector clocks that significantly shortens the length of the counterexamples produced when a bug is detected.

(3) We evaluate TraceLink on existing systems built using PGo: `raftkvs`, `locksvc`, and `dqueue`. We report on 9 bugs that we found across PGo's trusted computing base, split into 6 categories: 1 PGo miscompilation bug, 2 inaccurate network models, 2 runtime instrumentation bugs, 2 incorrect timeout behaviors, 1 misconfigured failure detector, and an out of place symbolic abstraction. We show that our *sidestep* property reduces length of counterexamples by 29%. We also find that our templatization strategy achieves 93× compression of the generated TLA$^+$.

2 BACKGROUND

This section introduces the concepts necessary to understand TraceLink's design. We discuss TLA$^+$, Modular PlusCal (MPCal), and its compilation by PGo. We also describe vector clocks.

2.1 TLA$^+$

TLA$^+$ [Lamport 1994, 2002] is a formal modeling language to represent and help people reason about concurrent and distributed systems. It supports set-theoretic reasoning, as well as properties expressed using temporal logic [Pnueli 1977]. TLA$^+$ has been used to prevent and analyze bugs in large-scale industry systems [Bornholt et al. 2021; Brooker and Desai 2025; Hackett et al. 2023b; Newcombe et al. 2015]. The strength of this type of design-level modeling is that it is flexible: developers can construct arbitrary mathematical abstractions that suit their application domain, with useful models often being much smaller than the target system.

There are multiple tools available for reasoning about TLA$^+$, which can be used alongside one another depending on need [Konnov et al. 2022]. Apalache [Konnov et al. 2019] offers symbolic model checking, while the TLA$^+$ Proof System [Merz and Vanzetto 2012] is a proof assistant. In this work, we use the explicit-state Temporal Logic Checker, TLC [Kuppe et al. 2019; Yu et al. 1999], since we are exploring concrete tracing data.

2.2 PGo and Modular PlusCal

PGo [Hackett et al. 2023a] is a compiler from Modular PlusCal (MPCal)[1] specifications to both TLA$^+$ and the Go programming language. It can be used to build verified, efficient distributed systems, and automates translating an otherwise abstract and non-deterministic specification into an executable implementation.

MPCal, like PlusCal [Lamport 2018], is a modeling language that extends TLA$^+$ with imperative constructs. On its own, PlusCal extends TLA$^+$ with explicit *processes* (as in anything similar to threads, coroutines, OS processes) and *statements* (assignments, loops, conditionals), which make TLA$^+$ more accessible.

MPCal extends PlusCal with definitions that separate a specification's intended algorithm from its modeled environment. These are *mapping macros*, *archetypes* (a generic form of PlusCal's processes), and their *instantiations*.

2.3 Modular PlusCal by Example

Consider a distributed algorithm that is expressed in terms of network communication. In PlusCal, this network communication is simulated in-line with the algorithm, and is indistinguishable from the algorithm's behavior. With MPCal, this communication can be encapsulated using *archetypes*, *instantiations*, and *mapping macros*. While we focus on mapping macros due to their importance to TraceLink's design, for exposition, we describe a short example of MPCal's abstraction semantics end to end.

```
archetype AConsumer(ref net[_], ref proc) {
c: while (TRUE) {
    c1: net[PRODUCER] := self;
    c2: proc := net[self];
   }
}
```

Listing 1. Example of an MPCal archetype definition.

Listing 1 shows part of a producer-consumer model written in MPCal. It uses an `archetype` definition to describe a loop that infinitely sends then receives messages on a network. The send

[1]MPCal is introduced in the same paper as PGo.

operation is labelled c1, and writes the process's own identifier, self, to the network at the known address PRODUCER. The receive operation is labelled c2, and it receives a value from the process's own network connection (address self), which it then writes to the abstract output proc. The surrounding while loop is labelled c, causing c1 and c2 to alternate forever, and, like in PlusCal, each label identifies an atomic action the process can take. MPCal's abstraction mechanism allows Listing 1 to refer to net and proc without assuming what they do.

```
variables network = [id ∈ 0..NUM_NODES-1 ↦ ⟨⟩],
          processor = 0;

fair process (Consumer ∈ 1..NUM_CONSUMERS) ==
    instance AConsumer(ref network[_], ref processor)
    mapping network[_] via ReliableFIFOLink;
```

Listing 2. MPCal global variable definitions and archetype instantiation.

Listing 2 shows how an MPCal archetype can be instantiated using model variables. Mirroring how a process must be launched within a system's implementation, MPCal requires that archetypes be instantiated in a specific context for model checking. The archetype AConsumer is instantiated using the instance directive, parameterized with the global network and processor state variables, which are defined above. Additionally, we constrain the network variable to be viewed via the definition ReliableFIFOLink, which allows us to customize the modeled behavior when the archetype reads from or writes to net.

```
mapping macro ReliableFIFOLink {
    read {
        await Len($variable) > 0;
        with (readMsg = Head($variable)) {
            $variable := Tail($variable);
            yield readMsg;
        };
    }
    write {
        yield Append($variable, $value);
    }
}
```

Listing 3. MPCal mapping macro implementing a reliable FIFO network.

Listing 3 shows the mapping macro with which AConsumer was instantiated, which models a TCP connection. It has two parts, *read* and *write*, which model what happens when the network resource is read from and written to, respectively. These read and write behaviors are written in PlusCal syntax, and may contain arbitrary logic. The MPCal-specific syntax is the yield statement, the $variable identifier, and the $value identifier. $variable refers to an underlying state variable in the model (it may be both read and assigned), and $value refers to the value that was written to the resource (only in the context of a write operation).

In a read context, `yield 42` means that the algorithm should receive the value 42, which may be computed using `$variable`. In a write context, `yield $value + 1` has the effect `$variable := $value + 1`, meaning that the underlying state variable is assigned a value computed using `$value` (one greater, in this case). These 2 operations can be used to model arbitrary input/output primitives, which become the environment of an MPCal algorithm. Note that these definitions rely on `$variable` being initialized to a sensible value, which is done during instantiation.

In Listing 3, the mapping macro operates on an underlying TLA$^+$ sequence that models in-flight messages, whose initial value is $\langle\rangle$. In this context, the read operation requires the underlying sequence to be non-empty on line 3, pops a message from the front of the buffer on lines 4-5, and marks the popped message as the mapping macro's result on line 6. The write operation is one statement on line 10, which appends `$value` to the in-flight messages.

2.4 MPCal Critical Sections

TraceLink's auto-instrumentation depends on PGo's implementation of critical sections, which we detail here.

When compiling MPCal, PGo must ensure that the implementation follows the model's concurrency semantics. A key challenge to this is PlusCal's, and therefore MPCal's, control flow semantics. Every collection of statements is grouped under a label, which marks the group as an atomic operation. In this context, the `await` statement can arbitrarily abort computation based on a boolean condition, and `either` statements cause non-deterministic branching. PGo's implementation of these requirements uses a non-nested form of transactions, which provides concurrency isolation and arbitrary rollback support.

As part of this translation model, PGo passes all effectful runtime behavior through a uniform API. This API models the following events: *read*, *write*, *abort*, *pre-commit*, and *commit*. *Read* and *write* identify all side-effects performed by the compiled MPCal, and allow inspection of all data "entering" and "leaving" the compiled code. These operations are considered black-box, and should implement either state variables, or behavior corresponding to a mapping macro, like the one in Listing 3. *Abort*, *pre-commit*, and *commit* are lifecycle events. We ignore *pre-commit* for the purposes of this paper. *Abort* and *commit* are the 2 ways a critical section may end, respectively: rollback and start over, or finish and continue to the next critical section. We discuss our formalization of these semantics in Section 4.

2.5 Ordering events with Vector Clocks

TraceLink uses vector clocks [Fidge 1988; Mattern 1989] to establish a partial order between log entries. If all events in a system are annotated with vector clocks, we can query whether two events happened one before the other, or if they were concurrent and could have happened in either order.

Vector clocks are in the form of a function from process identifier to a natural number describing a snapshot of the identified process's local clock. Vector clocks are partially ordered by the happens-before relation [Lamport 1978], and they form a lattice over this happens-before relation. A weakness of vector clocks is that the metadata per message grows with the number of nodes in the system, which may cause too much overhead for systems with many nodes. In practice, we evaluated TraceLink on executions involving 26 nodes or fewer, and at that scale these overheads were not a bottleneck.

3 TRACELINK OVERVIEW

Figure 1 overviews TraceLink's design. TraceLink takes as input a formal model of a distributed system, as well as the tracing data from an execution of a system that is supposed to be an implementation of the formal model. The formal model is assumed to be written in MPCal, and the

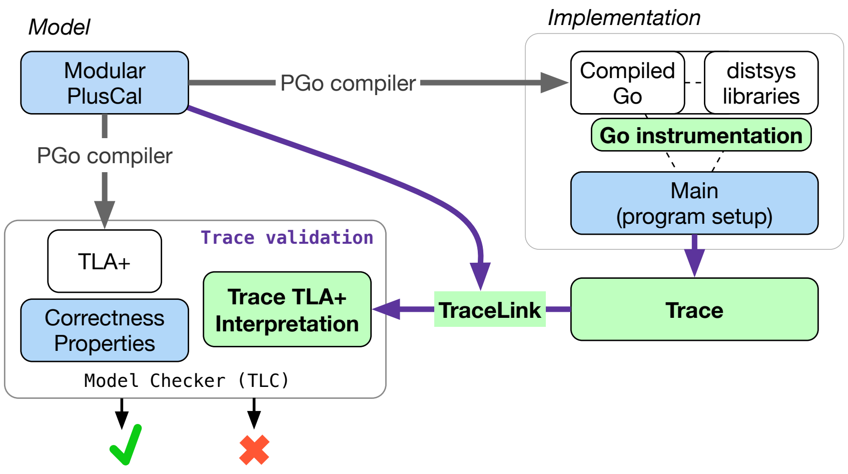

Fig. 1. Design of TraceLink for the PGo toolchain. This diagram extends Figure 1 from the PGo paper [Hackett et al. 2023a]. The blue parts are provided by the user. TraceLink components described in this paper are in green. Note that TraceLink does not modify the PGo compiler.

implementation is assumed to be compiled using PGo. Given these inputs, TraceLink uses TLC to explore the possible interpretations of the given execution trace.

To determine whether the implementation behaves correctly in relation to the MPCal specification, TraceLink imports the TLA$^+$ definitions generated by the PGo compiler. These definitions include the model's next-state relation, which can be used as a refinement property [Abadi and Lamport 1988]. It is also possible reference arbitrary correctness properties from the source specification. For example, the `raftkvs` system comes with 10 properties which could be checked against implementation traces using this method. It is also possible to write new free-form properties and check those.

Given this setup, there are specific types of bugs that can and cannot be found by TraceLink. Some issues stem from the nature of trace validation as a technique, and some issues are specific to TraceLink. We spend this section discussing TraceLink's trusted compute base (TCB), and its effect on TraceLink's bug finding capability.

3.1 Trusted Computing Base (TCB)

TraceLink's TCB is the translation code from implementation traces to TLA$^+$, and the TLC model checker. Notably, the TCB does not include the trace generation process, Go's runtime, PGo's compilation process, any manual modification made to PGo's output Go code, the formal model, or assumptions made by the model about the implementation's environment.

Below, we describe the effect of this TCB on which bugs TraceLink can and cannot detect. We discuss practical examples of these situations in Section 9.2.

3.2 Detectable Bugs

Inconsistent logging. TraceLink asserts that the implementation's logging output is logically self-consistent. While we rely on the TraceLink's logging code to observe the implementation, it is not part of the trusted compute base. Any observably incorrect behavior might be incorrect implementation behavior, or it might be due to an incorrectly implemented logging routine. In either case, it will raise a validation error. Consider, for example, possible changes to the `proc` variable in Listing 1. Assume the system implementation records setting it to value `A`, then a different component reads it and observes value `B`. If there is no 3rd action to explain the change, then TraceLink raises an error. The “instrumentation” bug category in Section 9.2 is a real-world example of this.

Inaccurate environment models. A well-known concern with verified system compilation is that the assumptions under which verification was performed may not reflect the deployment environment. TraceLink evaluates the model's environment assumptions as it explores implementation traces, and it will flag divergences between implementation behavior and the environment model.

For example, consider the `net` variable in Listing 1, whose behavior is defined in Listing 3. When `net` is read from, the model removes and returns the first element of an underlying sequence. Done repeatedly, each element of the underlying sequence should be returned in order, as long as the sequence is non-empty. The implementation, where this environment definition is replaced with hand-written library code, must be recorded producing outputs that match this modeled behavior. The “network assumption”, “timeout model”, and “failure detector” bug categories in Section 9.2 are real-world examples of this.

Miscompilations by PGo. TraceLink asserts that each event in a trace is allowed by the source model. While TraceLink does not log the result of every arithmetic and data structure operation that occurs during system operation, any value that becomes observable to the outside world is recorded. This means that if a PGo miscompilation outputs a wrong value to the world, then validation will fail. The “miscompilation” bug category in Section 9.2 is a real-world example of this.

3.3 Undetectable Bugs

Low-level safety issues. TraceLink trusts that the system emitting logs is performing correct low-level operations, such as memory and resource management. If there are such errors, we expect that the implementation will crash, or that these errors can be detected using established low-level safety assertions [Huang et al. 2024]. These are not meaningfully comparable to an abstract model of the system's intended logical behavior.

“Two wrongs make a right” scenarios. In a case where there are multiple bugs in an implementation, the bugs could interfere with each other to coincidentally produce a correct implementation trace. In this case, trace validation would not detect anything. Re-running the same test scenario multiple times helps. Fuzzing may also improve our ability to find such issues [Gulcan et al. 2024].

Non-modeled factors. It is possible for an implementation to not fulfill its purpose, while only performing operations allowed by its formal specification. For example, a system can be specified to respond to a timeout event by performing a recovery operation. How often that timeout should occur is rarely modeled in MPCal. A misconfigured timeout value during deployment could cause recovery routines to run too often, effectively preventing the system from handling requests. Since recovering from a timeout is a valid behavior, and the intended distribution of timeout events is not modeled, TraceLink cannot distinguish a normal timeout recovery step from excessive repetition of that step.

4 AUTO-INSTRUMENTATION AND TRACE GRAMMAR

Trace validation requires the following: (1) instrumentation to trace behavior of the target implementation; (2) a mapping from implementation trace structure to states in the model. In this Section, we explain the relationship between PGo's runtime library and our tracing format's formal grammar to handle (1). In Section 5, we give a formal semantics that, alongside introspection of the associated MPCal specification, enables automatic translation of trace events into TLA$^+$, achieving (2).

As a PGo-compiled system runs, it tracks all modifications to the system state in terms of the source MPCal's state variables. Reflecting MPCal state variable accesses, it divides MPCal critical sections into a series of sub-steps. These sub-steps need not literally reflect the specification as written – TraceLink would be oblivious to a significant rewrite of PGo's code generation, for example. Only the action as a whole must validate against a corresponding part of the specification. The relevant sub-steps (detailed in Section 2.4) are *read*, *write*, *commit*, and *abort*. Each critical section execution forms a straight-line sequence of read and write steps, ending in either a commit or an abort step. The length of this sequence is bounded because MPCal critical sections cannot contain loops, and conditional branches are flattened into non-branching sequences that can be thought of as "I saw this and then did that". This means that branches are disambiguated based on their effect, so a construct like `IF TRUE THEN x' = 1 ELSE x' = 1` is indistinguishable from `x' = 1` under PGo's execution tracing.

Our mechanism for implementing program values – the state variable values above – is also automatic. All program values in a PGo-generated program are serializable to TLA$^+$, and we rely on this fact. To record precedence between asynchronously logged events, we automate instrumentation of the target program with vector clocks by modifying PGo's runtime library. The process is fully automatic for generated code. When dealing with user Go code, we make instrumentation semi-automatic by transparently tagging vector clock timestamps onto program values entering and leaving any custom code. This requires no changes to user code that operates with pass-through semantics, including sending values across a network or storing them to disk. In rare cases where user code damages or deletes vector clock data, TraceLink's validation will detect the resulting vector clock corruption, and the user code can be edited to avoid the issue. We discuss these scenarios in Section 9.

By relying on PGo's runtime support code, we are able to directly instrument all relevant parts of the implementation using only runtime hooks and introspection of PGo's existing metadata. We do not need to rely on, or modify, the PGo compiler. We first explain our tracing semantics through a small MPCal example, then we give the formal grammar for the general structure.

```
1  readMsg:
2      msg := network[self]; \* recv msg
3      if (msg.type = "A") {
4          goto processA;
5      } else {
6          goto processB;
7      }
```

```
1  "readMsg" ⟵ read(.pc)
2  [type ↦ "B"] ⟵ read(network, 1)
3  write(msg) ⟵ [type ↦ "B"]
4  [type ↦ "B"] ⟵ read(msg)
5  write(.pc) ⟵ "processB"
6  commit()
```

Listing 4. MPCal fragment showing a local variable, an I/O operation, and control flow, alongside an example trace of critical section events that MPCal would generate.

Listing 4 shows an example MPCal fragment and a trace that is generated by its implementation. In this example, a process reads from the network (`network[self]`), receiving a message, and assigning it to a variable `msg`. Then, it reads that variable, inspects its `type` field, and depending on whether the value is `"A"` is not, it branches to the labels `processA` or `processB`.

Unlike MPCal, which is abstract, traced values are concrete; for the trace in Listing 4 we assume `self = 1` and `msg = [type ↦ "B"]`. Line 2 of the trace is the network access (line 2 of MPCal). Line

trace	::=	*record**	
record	::=	(*clock*, *self*, *event**, *end*)	complete MPCal critical section
clock	::=	vector clock	
self	::=	*value*	current process identifier
event	::=	*value* ← `read`(*var*, *idx**)	read operation
	|	`write`(*var*, *idx**, *old-value*?) ← *value*	write operation
var	::=	MPCal state variable name	
idx	::=	*value*	index into a state variable
value	::=	literal TLA$^+$ value	
old-value	::=	*value*	value that was overwritten
end	::=	`commit()`	record changes state as described
	|	`abort()`	record has no effect

Fig. 2. Grammar of PGo-generated traces.

3 of the trace is the write to `msg` (also line 2 of MPCal). Line 4 of the trace is line 3 of the MPCal, and line 5 of the trace corresponds to line 4 of the MPCal.

Note the special `.pc` variable in lines 1 and 5 of the trace. This stands for program counter – the same one from PlusCal's TLA$^+$ translation – and tracks which MPCal label (critical section) should be executed. It is read once at the start of a critical section, and then written once at the end.

4.1 Tracing Format for PGo

Figure 2 gives a formal grammar of PGo's generated execution trace format. Names ending with stars, such as *event**, may appear 0 or more times using the appropriate separator. Names marked with question marks, such as *old-value*?, mark an optional item. Names in italics reference other grammar productions, while monospace names are part of the literal syntax. Underlined statements, such as vector clock, semantically describe a value whose structure has been omitted.

Note that, while the format described here is PGo-specific, we theorize it could be generalized to other trace validation scenarios. Many parts of the structure are generic already, with the specialized parts relating to PGo's specific model for side-effects. Extending or replacing the read/write events might allow for handling a greater variety of implementation traces.

A *trace* is a collection of 0 or more records, each of which reflects a critical section executed by the implementation. The *record* structure is a generalization of the concepts shown earlier. We elaborate on the meaning and usage of each component below.

The *clock* is a vector clock which implies a partial order between records in the trace. Multiple machines, OS processes, or threads may independently log different parts of a PGo implementation's trace with no synchronization, so we use vector clocks to track and record a causal order between events. This implies that record sequences should be evaluated according to their causal order, not their lexical order. The *self* value is the unique TLA$^+$ identifier of the current process, and doubles as its index into the vector clock. We assume that the implementation records vector clocks starting with the empty clock, and incrementing with no gaps.

The sequence of events in a record is illustrated in Listing 4, which we use as an informal example of event semantics.

A `read` event identifies the variable *var* that was read, indices *idx** used by the read operation, and the *value* that was received. This reflects the TLA$^+$ assertion `var[idx]* = value`, and can represent any type of variable access allowed in MPCal. In the most common case of reading a local state variable, the event looks like *value* ⟵ `read`(*var*), with no indices.

Indices are used to model mapping macro semantics. Line 2 of Listing 4 refers to a mapping macro, such as the one in Listing 3, which can be parameterized using index syntax. In this case, the parameter is the network address to access (`self`), which was recorded as `1` on line 2 of the example trace.

Conversely, `write` events represent output events, recording the value that was written rather than read[2]. The assignment to `msg` on line 2 of the MPCal listing appears on line 3 of the trace as a plain assignment. If the listing contained a network send operation of the form `network[idx] := val`, it would be recorded as `write(idx)` ⟵ `val`, for the same reasons that a read operation would be.

The *end* marker identifies whether the critical section was successful. Successful (committed) critical sections' side-effects must be reflected in the rest of the trace, whereas unsuccessful (aborted) critical sections must have no visible side-effects. While they may not have side-effects, aborted critical sections still record observations the implementation made, and can help detect instances where an implementation process saw something invalid before aborting.

5 INTERPRETING EXECUTION TRACES AS TLA$^+$

To validate traces, we need to translate them into TLA$^+$. In this section, we present a complete formal semantics of the translation from any grammatically valid implementation trace to TLA$^+$. These semantics have the same behavior as the TraceLink implementation, but they omit a significant code size optimization that we discuss in Section 6. We assume the source MPCal's TLA$^+$ translation is imported, and its definitions, invariants, and so forth are contextually available to our translation (though we re-use none of the next-state definition). We also assume, and therefore do not discuss, a standard approach to naming fresh variables.

Note that the semantics we present here, including those in the appendices, are new for MPCal. There is no existing formalism, since the original PGo paper [Hackett et al. 2023a] describes MPCal through examples. Our work is therefore a step toward a formally described MPCal, though our semantics cover only the parts of MPCal needed by TraceLink. The only other formalization in the PlusCal space we are aware of is a TLA$^+$ specification of the PlusCal translator, which is part of the TLA$^+$ tools [Kuppe et al. 2019].

To contextualize our translation rules, Listing 5 shows the auto-generated TLA$^+$ definitions that the later rules build on.

- As a global logical timestamp, we define (line 1) and initialize (line 4) a global vector clock variable `__clock`. It is a function from a subset of process identifiers to natural numbers, which grows from the empty function, and is indexed by the `__clock_at` definition below. This variable reflects trace validation's logical time, and uniquely identifies every event along every permutation of the underlying trace.
- To index into vector clocks, we define the operator on line 7. The `__clock_at` operator implements indexing into a vector clock value, which produces either the number at that index, or the default value 0[3].

[2]Write operations can optionally store an *old-value* entry, which can help strengthen trace validation semantics. When accessing a local variable like `msg`, a write operation with an old value captures the value the variable used to have, as well as the incoming value. It is essentially a combined read-then-write record. This is not necessary for trace validation to work, and is not well-defined in combination with mapping macros. Regardless, more complete trace information can help catch bugs earlier.

[3]We choose to reflect the standard implementation of vector clocks here, which is to start with an empty function and grow its domain to eventually include all known process IDs.

```
1  VARIABLE __clock
2
3  Init ==
4    ∧ __clock = ⟨⟩
5    ∧ \* Init from original spec
6
7  __clock_at(__clk, __idx) ==
8    IF __idx ∈ DOMAIN __clk
9    THEN __clk[__idx]
10   ELSE 0
```

```
14 __clock_check(self, __next_clock) ==
15   LET __idx ==
16         __clock_at(__clock, self) + 1
17       __updated_clock ==
18         (self :> __idx) @@ __clock
19   IN  ∧ __updated_clock[self] =
20         __next_clock[self]
21       ∧ ∀ __i ∈ DOMAIN __next_clock :
22          __next_clock[__i] <=
23            __clock_at(
24                __updated_clock, __i)
25       ∧ __clock' = __updated_clock
```

Listing 5. Preamble for all TraceLink-generated TLA$^+$.

- We assume that all the same state variables from the TLA$^+$ specification being validated are defined, and, as identified on line 5, initialized to the same starting values as in the model against which we are validating[4].
- To define the partial order in which trace validation considers the trace, we defined `__clock_check` on line 14. This operator describes the conditions under which a traced event recorded by process `self` and marked with clock `__next_clock` may be evaluated. The updated global clock (`__updated_clock`) may only advance at `self` by 1, as defined on line 16 and enforced on line 19. This ensures we do not skip any traced events, local or otherwise. The other requirement, defined on line 21, requires that *the recorded clock may never reference events not yet explored by the global clock*. Note that this does not ensure happens-before between the global and recorded clocks, or recorded clocks along the explored path. This definition only ensures that the global vector clock updates according to happens-before.

5.1 Worked TLA$^+$ Translation Example

To introduce how our semantics are used, we show a TLA$^+$ representation of the trace in Listing 4. Appendix B.1 works through a translation of the associated `AServer_network_read` definition. All formal semantics we reference are explained in Section 5.

Listing 6 is the final result of the translation. Next, we explain how this translation is derived using our formal semantics. In what follows, we assume $self = 0$ and the smallest vector clock `(0 :> 1)`, mapping *self* to 1.

First, since the record ends with `commit()`, we apply Definition 2a, which generates lines 1, 2, and 8. From here, each individual event is sequenced according to rule 3, which is applied 4 times in total, to reduce the 5 events pairwise into one overall continuation-passing sequence. Note that, while we use `LAMBDA` and `__state` extensively in the definitions, the intended interpretation is that all immediately applied lambdas are treated as syntactic expansion, and, to avoid scoping errors,

[4]This limitation is not fundamental, since later definitions build a next-state relation that can be applied to any initial state, but we leave this to future work. For instance, TraceLink currently cannot validate log-rotated traces from a long-running system (that is, traces whose beginning has been truncated to save space), or traces that are missing events.

```
1 ∧ __clock_check(0, 0 :> 1)
2 ∧ LET __state0 == __state_get
3    IN  ∧ __state.pc[0] = "readMsg"
4        ∧ AServer_network_read(__state, 1, [type ↦ "B"], LAMBDA __state1 :
5             LET __state2 == [__state1 EXCEPT !.msg[0] = [type ↦ "B"]]
6             IN  ∧ __state2.msg[0] = [type ↦ "B"]
7                 ∧ LET __state3 == [__state2 EXCEPT !.pc[0] = "processB"]
8                    IN  __state_set(__state3))
```

Listing 6. Translation of the events in Listing 4, assuming a record with $self = 0$ and the smallest vector clock (0 :> 1).

__state and __next_state become sequentially numbered __state0, __state1, and so forth. The starting state is always __state0.

Now, we describe the sequence of 5 definitions which correspond individual trace events to the generated TLA$^+$.

- Definition 4a generates line 3, which asserts the initial value of .pc.
- Definition 6a generates line 4 by inserting a call to the ReliableFIFOLink mapping macro read routine, assuming context from Listing 9. The read operation is allowed to arbitrarily affect the state, so we pass a continuation lambda that receives the possibly-changed state __state1, replacing __state0.
- Definition 4b generates the let-binding starting on line 5, changing __state1 into __state2 by replacing process 0's copy of the local msg variable.
- Definition 4a generates the assertion on line 6, reading the same msg variable back from __state2, and checking that its value is the same one written just before.
- Definition 4b generates the let binding on line 7, which replaces the current __state2 with __state3, altering the program counter to point to a future step to execute, "processB". The __state3 binding is the last one in the sequence, so it is the one passed to __state_set on line 8. This commits __state3 as final for this critical section, completing the top-level application of Definition 2a.

5.2 Next-state Relation

$$[\![record^*]\!] \overset{\text{def}}{=} \boxed{\begin{array}{l}\texttt{Next ==}\\ \qquad \lor [\![record]\!]^*\end{array}} \tag{1}$$

For each record in the trace, Definition 1 shows that TraceLink translates maps that record directly to choices in a standard TLA$^+$ next-state definition (where each choice is marked by $\lor$). This disjunction removes any order between records, and without other constraints, implies every permutation of events. These permutations are constrained to causal order as part of record translation, using the __clock_check definition.

Note that this definition implies TraceLink would emit, for example, a 10k choice next-state for a 10k record trace. We explain how we compress this factor down to a size proportional to the original TLA$^+$ specification in Section 6.

$$[\![(clock, self, event^*, \texttt{commit()})]\!] \overset{\text{def}}{=} \boxed{\begin{array}{lll} \land & \multicolumn{2}{l}{\texttt{\_\_clock\_check}(self, clock)} \\ \land & \texttt{LET} & \texttt{\_\_state == \_\_state\_get} \\ & \texttt{IN} & [\![event^*]\!]_{\texttt{\_\_state\_set(\_)}} \end{array}} \tag{2a}$$

$$[\![(clock, self, event^*, \texttt{abort()})]\!] \overset{\text{def}}{=} \boxed{\begin{array}{lll} \land & \multicolumn{2}{l}{\texttt{\_\_clock\_check}(self, clock)} \\ \land & \texttt{LET} & \texttt{\_\_state == \_\_state\_get} \\ & \texttt{IN} & [\![event^*]\!]_{\texttt{LAMBDA \_ :}} \\ & & \qquad\qquad \texttt{\_\_state\_set(\_\_state\_get)} \end{array}} \tag{2b}$$

$$[\![event_1, event_2, events^*]\!]_{rest(\_)} \overset{\text{def}}{=} \boxed{[\![event_1]\!]_{\texttt{LAMBDA \_\_state : } [\![event_2,\ events^*]\!]_{rest(\_)}}} \tag{3}$$

5.3 Translating trace records

Definitions 2a and 2b show how each trace record is translated into a TLA$^+$ next-state relation. Most of the semantics are left to the individual events (see below), with specific initial and final steps defined here. These outer definitions show the context in which individual events are translated, and enforce causal order between trace records.

Both records that commit and those that abort start with the same definitions; the corresponding Definitions 2a and 2b both start with the same requirement. Given the record's *self* and *clock* entries, we begin by checking whether causal order is satisfied relative to the current global clock, using `__clock_check`. The record will only be validated when the global vector clock has a suitable value, relative to the locally recorded *clock*.

In the rest of these definitions, we use a concept called by-value state, which relies on two definitions generated by TraceLink: `__state_set` and `__state_get`. These function as getters and setters for *all TLA$^+$ state variables at once.* For example, for state variables `x` and `y`, `__state_get` would be defined as the expression `[x ↦ x, y ↦ y]`, which yields a function[5] that maps state variable names to their values. `__state_set` is its inverse, binding the members of the function value to primed state variables, as in `__state_set(__state) == ∧ x' = __state.x ∧ y' = __state.y`. We pass this state around as a value, which means we can speculate without binding the next-state (using `x'`) until we are ready to commit.

Looking back to Definitions 2a and 2b, if the vector clock is valid, then we build a context in which to write the translated events. We bind our by-value state to `__state` using a let binding, allowing subsequent translation rules to refer to the current state by that name.

The subscript syntax written $[\![\ldots]\!]_{\text{here}}$, which reads `__state_set(_)` in Definition 2a, is the output of the event's translation, modeled as a continuation. We use continuation passing throughout the remaining definitions, each time passing in a function that accepts a syntactic next-state value as a parameter. In Definition 2a, the continuation is just the `__state_set` definition, meaning to commit the final next-state value from evaluating the events. In Definition 2b, the continuation ignores the returned next-state, and instead sets the next-state to the current state, nullifying any state changes implied by the logged events.

Sequences of events within a traced record are considered to happen one after the other, with each event operating on the next-state of the previous one. Definition 3 shows how we combine sequences of multiple events given the continuation *rest(_)*. We first translate $event_1$, then we capture its result state via a continuation `LAMBDA`, binding it to `__state`. By rebinding the current state, we

[5]"Function" here is more commonly called "dictionary" in other languages.

$$[\![value \leftarrow \texttt{read}(var, idx^*)]\!]^{\text{local}}_{rest(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{l} \land\ \texttt{\_\_state}.[\![var]\!]^{\text{local}}[self][idx]^* = value \\ \land\ rest(\texttt{\_\_state}) \end{array}} \tag{4a}$$

$$[\![\texttt{write}(var, idx^*) \leftarrow value]\!]^{\text{local}}_{rest(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{ll} \texttt{LET} & \texttt{\_\_next\_state} == \\ & \quad [\texttt{\_\_state EXCEPT} \\ & \quad\ !.[\![var]\!]^{\text{local}}[self][idx]^* = value\,] \\ \texttt{IN} & rest(\texttt{\_\_next\_state}) \end{array}} \tag{4b}$$

$$[\![\texttt{write}(var, idx^*, old\text{-}value) \leftarrow value]\!]^{\text{local}}_{rest(\_)} \overset{\text{def}}{=} \boxed{\left[\!\!\left[\begin{array}{l} \texttt{read}(var, idx^*) \leftarrow old\text{-}value, \\ \texttt{write}(var, idx^*) \leftarrow value \end{array} \right]\!\!\right]^{\text{local}}_{rest(\_)}} \tag{4c}$$

pass the state produced be evaluating $event_1$ to the translation of $event_2$, $events*$. By inductively applying Definition 3, we can reduce any number of logged events into one continuation-passing sequence. Note that the overall continuation *rest(_)* is kept at the end of the chain, to receive the final next-state value.

Note also that passing a continuation along in this way enables each translation to individually invoke TLA$^+$'s non-determinism semantics using the $\lor$ operator. This is necessary to support non-deterministic mapping macro translations.

5.4 Read and Write Events

Individual read and write events have 3 different interpretations, depending on how the relevant state variable is defined in the MPCal model. The variant being considered by a given definition is marked with a superscript "local", "global", or "mapping". A local variable is local to a given MPCal process, and is syntactically defined within it. Each process has its own copy of the value, and reads and writes between processes do not interfere with each other. A global variable is, as the name suggests, global to the entire specification. It is syntactically defined at the top-level scope of an MPCal specification, and must be passed by reference to an archetype instantiation. Reads and writes to a global variable are visible across processes. A mapping variable is controlled by a mapping macro, and its semantics, including behavior under concurrency, are arbitrarily defined by the listed read and write operations. Which global variable(s) are affected by reads and writes to a mapping variable is defined by how the process is instantiated – that is, which global model variable the mapping macro is bound to, and whether the binding is marked with the `[_]` syntax ("indexed").

All MPCal state variables that are visible in a PGo-compiled implementation are uniquely identified by the pair *ArchetypeName.VariableName*. Unfortunately, this convention does not match the TLA$^+$ definitions generated by PGo. Appendix A – found in the supplemental material – details precisely how we relate the two. In this section, we refer to this variable lookup process using the notation $[\![var]\!]^{\text{local}}$, $[\![var]\!]^{\text{global}}$, or $[\![var]\!]^{\text{mapping}}$, labelled with the kind of variable to which the translation may be applied.

5.4.1 Local Operations. Definitions 4a-4c describe how local variable change events are represented in TLA$^+$.

$$[\![\mathit{value} \leftarrow \texttt{read}(\mathit{var}, \mathit{idx}^*)]\!]^{\text{global}}_{\mathit{rest}(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{l} \land\ \texttt{\_\_state}.[\![\mathit{var}]\!]^{\text{global}} [\mathit{idx}]^* = \mathit{value} \\ \land\ \mathit{rest}(\texttt{\_\_state}) \end{array}} \tag{5a}$$

$$[\![\texttt{write}(\mathit{var}, \mathit{idx}^*) \leftarrow \mathit{value}]\!]^{\text{global}}_{\mathit{rest}(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{ll} \texttt{LET} & \texttt{\_\_next\_state} == \\ & \quad [\texttt{\_\_state EXCEPT} \\ & \quad\ !.[\![\mathit{var}]\!]^{\text{global}} [\mathit{idx}]^* = \mathit{value}] \\ \texttt{IN} & \mathit{rest}(\texttt{\_\_next\_state}) \end{array}} \tag{5b}$$

$$[\![\texttt{write}(\mathit{var}, \mathit{idx}^*, \mathit{old\text{-}value}) \leftarrow \mathit{value}]\!]^{\text{local}}_{\mathit{rest}(\_)} \overset{\text{def}}{=} \boxed{\left[\!\!\left[\begin{array}{l} \texttt{read}(\mathit{var}, \mathit{idx}^*) \leftarrow \mathit{old\text{-}value}, \\ \texttt{write}(\mathit{var}, \mathit{idx}^*) \leftarrow \mathit{value} \end{array} \right]\!\!\right]^{\text{global}}_{\mathit{rest}(\_)}} \tag{5c}$$

$$[\![\mathit{value} \leftarrow \texttt{read}(\mathit{var}, \mathit{idx}^*)]\!]^{\text{mapping}}_{\mathit{rest}(\_)} \overset{\text{def}}{=} \boxed{\mathit{var}\texttt{\_read}(\texttt{\_\_state}, \mathit{idx}^*, \mathit{value}, \mathit{rest})} \tag{6a}$$

$$[\![\texttt{write}(\mathit{var}, \mathit{idx}^*) \leftarrow \mathit{value}]\!]^{\text{mapping}}_{\mathit{rest}(\_)} \overset{\text{def}}{=} \boxed{\mathit{var}\texttt{\_write}(\texttt{\_\_state}, \mathit{idx}^*, \mathit{value}, \mathit{rest})} \tag{6b}$$

Definition 4a shows that reading acts as an assertion, and matches PlusCal local variable semantics. Using the by-value state `__state`, it checks that the state variable of the corresponding name $[\![var]\!]^{\text{local}}$, indexed by *self*, matches the traced *value*. The `__state` is passed on to *rest* unmodified. The optional indices *idx* translate exactly to TLA$^+$ function application syntax, indexing into the state variable as many times as there are *idx*.

Local writes overwrite a state variable, rather than check it. Definition 4b replaces `__state` with `__next_state`, using TLA$^+$ `EXCEPT` syntax to replace the value at $[\![var]\!]^{\text{local}}$. The meaning is similar to immutable dictionary update syntax in other languages, and it uses the same indexing by *self* and idx^* as in the read operation. The modified `__next_state` value is then passed to the continuation *rest*, which will implicitly rebind it to the name `__state` if needed using Definition 3.

Local writes with *old-value* are a combined read-write operation, fusing both. Definition 4c invokes both Definitions 4a and 4b, combined into one via Definition 3. The read is placed before the write, so it checks the initial state, not the state after the write.

5.4.2 Global Operations. Global read-write operations are almost identical to local ones, but without the implicit reference to *self*, so the reads and writes apply to one global variable, not process-specific copies of it. As a result, Definitions 5a-5c are almost identical to Definitions 4a-4c, except for the absence of *self* indexing, and a different variable name lookup rule $[\![var]\!]^{\text{global}}$.

5.4.3 Mapping Macro Operations. Operations on variables controlled by mapping macros are translated into operator calls, one per variable, per archetype instantiation in the source MPCal. That operator contains a compilation of the mapping macro's implementation, which, like all the definitions here, operates on a `__state` value and produces results by continuation-passing.

Definitions 6a and 6b show how control passes to those operators[6]. Both read and write operations coincidentally have the same interface. They both operate on the `__state` value, they can both accept indices, and they both take the *rest(_)* continuation and call it with a potentially updated

[6]Notice that mapping macro reads and writes do not have an *old-value* form. We include that form out of convenience for local and global writes, since it is useful and easy to implement, but it is not essential. For mapped variables, it is easier to ignore the *old-value* if it is present, rather than to enforce a meaningful "previous value" on arbitrarily overridden MPCal behavior.

state value. The *value* parameter is interpreted as the expected value of a read operation, and the value to be written for a write operation. Note that with mapping macros, read operations can have side effects. Due to space limitations, we discuss the translation from mapping macro definitions to these helper operators in Appendix B.

6 TRACE TEMPLATIZATION

The translation process described in Section 5 is complete and can be used on small implementation traces. But, it scales poorly because it outputs a TLA+ specification size proportional to the trace. Anecdotally, when applied to a 30k element implementation trace, this naive approach produced over 500k lines of TLA+ and overloaded TLC's parser.

To scale to realistic traces of complex systems, we propose an alternative representation that yields TLA+ specifications of a size proportional to the source MPCal specification. Our key insight is that, if all the TLA+ values from the trace are replaced by placeholders in the naive translation, a majority of trace record translations become duplicates.

Consider the example translation from Listing 6. If the values for self, vector clock, and expected value are abstracted away, it is a generic single control flow path through the MPCal critical section `readMsg` defined in Listing 4. No trace-specific data is left, only the footprint of how the implementation chose to implement one of the specification's control flow paths. Technically, this means that the size complexity of our generated TLA+ is related to the set of control flow options explored by the implementation, but since the implementation is itself compiled from the MPCal specification using PGo, we expect both quantities to be consistent with one another.

```
∧ __clock_check(__record.self, __record.clock)
∧ LET __state0 == __state_get
    IN  ∧ __state.pc[__record.self] = __record.elems[1].value
        ∧ AServer_network_read(__state, __record.elems[2].idxs[1],
                                    __record.elems[2].value, LAMBDA __state1 :
            LET __state2 == [__state1 EXCEPT !.msg[__record.self] =
                                 __record.elems[3].value]
            IN  ∧ __state2.msg[__record.self] = __record.elems[4].value
                ∧ LET __state3 == [__state2 EXCEPT !.pc[__record.self] =
                                      __record.elems[5].value]
                    IN  __state_set(__state3))
```

Listing 7. Placeholder form of Listing 6, in terms of `__record`

To implement this placeholder form as valid TLA+, we move the specific traced values into the binary format that TLC uses for serializing model checker state. We then refactor our translation to be in terms of an abstract `__record` that refers to this serialized data.

Listing 7 adapts the naive translation in Listing 6 to this format. Alongside the stored vector clock and self values, the `__record` includes a sequence `elems`, which corresponds directly to the sequence of events in the original trace record. At each index, `elems` holds the expected value of the read or write, as well as any used *idx* values[7].

[7]This description omits some details. For example, in practice, we check the length of `__record.elems` as a precondition, systematically cross-check an additional `__record.elems[_].name` field, and tag elements as read or write operations. The `name` field and read/write tags help avoid type errors when wrongly evaluating a `__record` value that has a partially compatible structure, but contains data intended for a different definition.

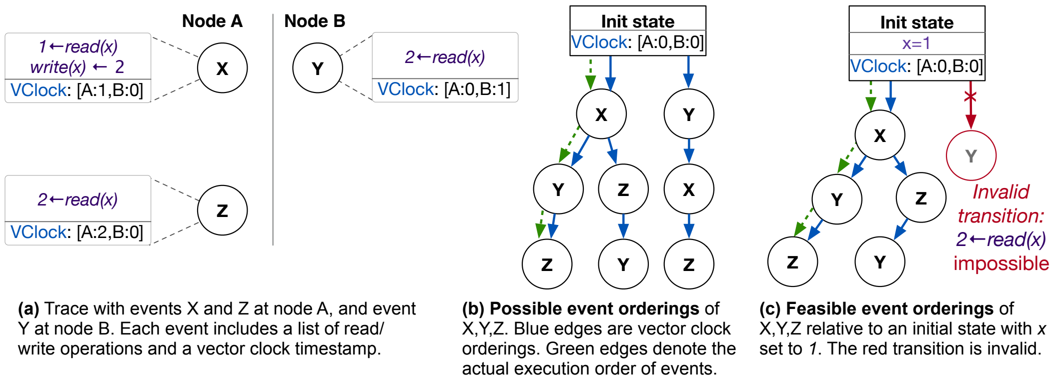
Node A
Node B
1←read(x)
write(x) ← 2
VClock: [A:1,B:0]
X
Y
2←read(x)
VClock: [A:0,B:1]
2←read(x)
VClock: [A:2,B:0]
Z
(a) Trace with events X and Z at node A, and event Y at node B. Each event includes a list of read/ write operations and a vector clock timestamp.
Init state
VClock: [A:0,B:0]
(b) Possible event orderings of X,Y,Z. Blue edges are vector clock orderings. Green edges denote the actual execution order of events.
Init state
x=1
VClock: [A:0,B:0]
Invalid transition: 2←read(x) impossible
(c) Feasible event orderings of X,Y,Z relative to an initial state with x set to 1. The red transition is invalid.

Fig. 3. (a) A trace with three events, (b) possible event orderings of these events that ignore state, and (c) feasible event orderings that account for state.

Retaining Readability. The indirection in the templatized format makes debugging more difficult. The user can no longer inspect the generated TLA$^+$ and see which values were asserted, and TLC counter-examples will not give visibility into `__record`.

We work around this by adding two key additional definitions, which TLC will include in every step of all its counter-examples traces.

First, we add a synthetic `__action` variable which is always set to the `__record` value chosen for a given model checking step. By cross-examining this additional variable alongside the placeholder TLA$^+$ definition, it is possible to recover the true context of each step in a reported counter-example.

Second, we record a function that maps every known *self* value to the next step that process is allowed to take. This means that, if TLC reports a deadlock – meaning it could not identify any path forward in its interpretation of a trace – then we have a list of the options it considered and rejected.

7 TRACE PATH VALIDATION STRATEGIES

So far, we have described TraceLink's TLA$^+$ generation and trace collection features. Now, we show how TraceLink uses TLC to analyze that data, with attention to how TLC explores paths through the implementation trace's state space.

Each event in a trace includes a vector clock timestamp and a set of operations on state. As discussed in Section 4, the vector clock timestamps partially order the events. Figure 3(a) illustrates a trace with three events and Figure 3(b) shows the corresponding total and partial orderings of these events. The total order is for illustration, since the trace does not imply it. In any validation strategy, TraceLink may pick the true total order, or it may pick a different path that also satisfies the vector clocks.

The vector clock ordering provides us with two possible trace path validation strategies:

(1) One-path: Select one path among all the paths in the partial order implied by the vector clock timestamps.
(2) All-paths: Validate every path in the partial order implied by the vector clock timestamps, covering all possible interpretations of the trace. This could be done using DFS or BFS traversal.

These two strategies are natural and have been explored in the literature. For the all-paths strategy, several optimizations are possible, such as those that perform partial order reduction [Flanagan and Godefroid 2005].

Figure 3(c) illustrates how the orderings from Figure 3(b) may interact with state. In particular, note that the transition from the initial state with event Y is impossible. If a system recorded this, it would be a bug. The one-path strategy may get lucky in finding this bug if the selected path includes Y as the first event. All-paths will find this bug, but it is an expensive strategy.

We design a third strategy, called one-path-sidestep, that is able to detect the bug in Figure 3(c):

(3) One-path-sidestep: Select and validate one path among all the paths in the partial order implied by the vector clock timestamps. Also validate that any possible branch away from the chosen path ("side" transition) must be accepted.

The key idea behind one-path-sidestep is to extend the one-path strategy with a *SideStepProgress* check (Listing 8). This check requires that for every process ID, if the vector clock data of its next step would allow progress, then that next step must be valid in the current state. This statement is a strengthening of the next-state relation generated by TraceLink. For the example in Figure 3(c), one-path-sidestep would find the bug regardless of which vector clock ordering is selected.

```
__SideStepProgress ==
    ∀ self ∈ __self_values :
        __clock_check(self, __records[self].clock) => __TryNext(self)
```

Listing 8. TraceLink's partial progress invariant

While one-path-sidestep may find bugs more consistently, its main use is in stripping irrelevant suffixes from TraceLink's reported counter-examples. For instance, consider adding a node C to Figure 3(a), which reports a sequence of valid and unrelated actions P, Q, R, S. TLC is likely to report a counter-example where Y was found to be invalid after exploring all of P, Q, R, S first, despite these events being irrelevant. This arbitrary and irrelevant suffix hides that a much shorter path (in Figure 3(c), just the initial state) also shows Y is invalid. One-path-sidestep will always report this shorter path, which makes understanding TraceLink's counter-examples significantly easier.

Next, we describe the TraceLink implementation and present our evaluation.

8 TRACELINK IMPLEMENTATION

We developed TraceLink in a mix of Scala, Go, and TLA$^+$.

TraceLink. For simplicity, we developed TraceLink as an extension to the PGo compiler. We added new commands to the PGo executable and used its existing parsing routines and data structures. These 1,370 LOC of Scala changes are purely additive, live in separate folders, and do not modify PGo's compiler features.

Our changes handle reading log files, inferring configuration from MPCal, and generating the TLA$^+$ definitions and data files for trace validation. Our prototype does not contain any handwritten TLA$^+$ files.

Go runtime changes. To instrument PGo-generated systems, we modified PGo's Go runtime library with 1,035 LOC. This includes logging (472 LOC), instrumentation to record read and write steps (49 LOC), concurrency testing to derive more diverse traces (28 LOC), and instrumenting the Go-land TLA$^+$ value type with vector clock annotations (486 LOC).

System	MPCal LOC	TLA$^+$ LOC	Go LOC	Events per trace	Concurrency
dqueue	76	99	175	4-22	2
locksvc	80	159	281	10-53	6
raftkvs	990	7,993	3,163	2,772-100,644	6-26

Table 1. Description of systems and their traces that we used to evaluate TraceLink.

9 EVALUATION

We evaluate TraceLink, with a focus on answering the following five research questions:

- **RQ1**: What sorts of bugs does TraceLink find?
- **RQ2**: How does the choice of trace validation policy impact bug finding?
- **RQ3**: What is TraceLink's overhead, considering model checking performance and the size of the generated TLA$^+$?
- **RQ4**: What is the impact of TraceLink's instrumentation on the Go implementation's runtime?
- **RQ5**: What is the manual effort involved in using TraceLink?

To answer these questions, we used TraceLink to validate 3 systems built with PGo. For each system, we collected a number of execution traces, and then we used TraceLink to validate those traces against the source MPCal models, for each type of validation available.

9.1 Subject Systems and Trace Collection

We evaluated TraceLink on three systems prototyped using PGo [Hackett et al. 2023a]. Table 1 lists information about these systems, including the LOC for the MPCal model; and the LOC of TLA$^+$ and Go produced by PGo[8]. dqueue is a simple producer-consumer model. locksvc is a centralized lock service. raftkvs is a key-value store based on Raft [Ongaro 2014]. We validated each of these systems without any preliminary modification, excluding internal changes to PGo's runtime library described in Section 8, and any fixes to the bugs we found.

For each configuration of these systems, we collected 10 execution traces per configuration. We did this by re-running the relevant parts of the systems' integration test suites with tracing enabled. To help us gather more diverse traces, we instrumented PGo's runtime library to add random sleeps in between I/O operations (randomized stress testing, where each sleep duration is chosen from an exponential distribution with an average of 100 microseconds), to approximate the schedule exploration technique in [Burckhardt et al. 2010]. We used a single configuration for dqueue and locksvc, and we ran raftkvs with node counts from n=1 to n=5, repeating system execution until we had 10 distinct traces. For more diverse traces, we also ran a variation of raftkvs n=3 with simulated node failure, which was also part of its test suite. Table 1 lists the range of events for the 10 traces for each system. The table also lists the range of concurrent processes for these traces; this corresponds to the number of entries in the traced vector clocks.

We parallelized our validation runs across 3 identical bare-metal servers, each with 64GB of RAM and a 32-core Intel Xeon Silver 4309Y CPU. We validated every trace under each trace validation policy and collected data on what happened[9]. We performed validation using TLC nightly revision bacd9d2.

[8]The MPCal/TLA$^+$ models we use are part of the PGo project: https://github.com/DistCompiler/pgo. Each one is in a subfolder of the same name in the systems/ directory.

[9]We omitted all-paths validation for rafkvs, since we had to terminate a single all-paths validation of a 3 node execution when it had not finished within 12 hours.

Name	Category	Description
BUG1	network assumption	A common MPCal modeling pattern incorrectly implied a strict order between send operations across unrelated TCP connections.
BUG2	network assumption	The `raftkvs` specification incorrectly assumed the implementation could precisely count all in-flight messages sent to a server.
BUG3	miscompilation	PGo's runtime library incorrectly implemented the `@@` operator in a way that passed all implementation tests and evaluation to date.
BUG4	instrumentation	TraceLink's vector clocks instrumentation incorrectly followed state rollbacks for shared variables, resulting in corrupted traces.
BUG5	instrumentation	The buffer check from BUG2 was incorrectly instrumented in the implementation, leading to missing happens-before constraints.
BUG6	timeout model	The `raftkvs` client was specified to always observe timeouts, whereas sometimes the implementation recorded *not* timing out.
BUG7	timeout model	The `raftkvs` servers were approximated to either always or never detect leader timeouts. The implementation mixes both behaviors.
BUG8	failure detector	Failure detectors in `raftkvs` were modeled as perfectly accurate, which did not reflect the implementation.
BUG9	symbolic abstraction	The `raftkvs` client was modeled to choose from a set of 3 symbolic requests, which did not reflect arbitrary implementation behavior.

Table 2. All the bugs detected with TraceLink during evaluation.

When reproducing a bug, we used two techniques depending on the bug type. For bugs that required MPCal fixes, we used correct implementation traces, and validated those traces against different versions of the corresponding MPCal specification, which exhibited each bug individually. For implementation bugs, we recreated the bug in question, and gathered 10 execution traces from the buggy system, which we then validated against a correct MPCal specification.

For validation, we used the MPCal model's next-state relation as a correctness property via TLA$^+$ refinement [Abadi and Lamport 1988]. We chose not to use the existing correctness properties from the source models, because checking them adds overhead, and nominally the original specification has been found to satisfy those properties. Evaluating whether additional issues may be found by enabling those correctness properties, given that TraceLink's concrete traces naturally go beyond those found during bounded model checking, is left as future work.

9.2 RQ1: Bugs found with TraceLink

To answer RQ1, we list the 9 bugs we found during our evaluation of TraceLink in Table 2. We categorize these bugs into 6 categories: network assumption, miscompilation, instrumentation, timeout model, symbolic abstraction, and failure detector. If a bug does not name a specific system, that bug could apply to any system using PGo. This does not mean all systems were affected, however. BUG3 was a compiler bug, but only `raftkvs` used the affected feature.

Note that, while we found many issues (and one compiler bug) where our MPCal specifications were inaccurate compared to the implementation, fixing those inaccuracies did not change the outcome of model checking the specification. The inaccuracies we found were not hiding specification bugs, and on re-verification, the specifications stayed correct relative to their stated properties. Specification inaccuracies can cause real problems [Fonseca et al. 2017], however, and the fact that we found these bugs demonstrates that TraceLink is a means to detect and prevent these issues.

9.2.1 Network Assumptions (BUG1, BUG2). We found 2 incorrect assumptions about network behavior in the MPCal specifications we evaluated, both of which had been missed by regular model checking and conventional testing.

BUG1 is general, in that it affects any system built with PGo that uses TCP. Due to an ordering constraint implied by MPCal's strict critical section semantics, possible network behaviors were left unexplored during model checking. TraceLink exposed this fact by cross-checking the assumption with a real network interface.

BUG2 is specific to `raftkvs`, and was due to an inaccuracy in how the model represented checking whether a server had pending messages remaining. The MPCal referred to all in-flight messages, while the system could only check its inbound buffer. Since the check was a comparison against 0, the intent was that some mismatch was acceptable as long as the overall condition worked. TraceLink proved that the number of pending messages could be under-reported as 0 in practice, meaning that the implementation could diverge from model checked behavior.

9.2.2 Miscompilations (BUG3). TraceLink was able to find a previously-undiscovered bug in the PGo compiler itself. PGo had been consistently miscompiling the `@@` operator with its arguments swapped, due to a misunderstanding of the operator's TLA$^+$ meaning. The bug passed a suite of end-to-end implementation tests because the miscompiled operator *worked correctly in context.* The incorrect behavior was in the MPCal itself, where the operator was used incorrectly. This was missed during model checking because the intended semantics were considered too self-evident to express as a property, and the implementation (being miscompiled to swap the arguments into correct order) actually worked.

Since TraceLink exhaustively checks specification-implementation consistency, it can be used to find bugs like this one, that both conventional model checking and manual testing can miss.

9.2.3 Instrumentation Issues (BUG4, BUG5). These bugs demonstrate that TraceLink is able to detect errors in its own instrumentation. In each of BUG4 and BUG5, corrupted or under-specified vector clock data was logged during implementation tracing. Because the vector clocks and implementation behavior were not consistent, TraceLink raised a validation error. Rather than give unsound validation results, TraceLink helped us to debug its own instrumentation code.

9.2.4 Timeout Model Issues (BUG6, BUG7). TraceLink was able to detect inaccurate, overly restricted timeout modeling in the `raftkvs` specification. It is easy to forget a simplifying modeling assumption like timeouts always fire or never fire, but this assumption is inaccurate in a real implementation. BUG6 and BUG7 are two different instances of this issue, which had been set, forgotten, and not documented in the original MPCal.

This shows that TraceLink can help keep assumptions about system behavior documented, since it becomes possible to detect when MPCal and implementation behaviors drift apart.

9.2.5 Failure Detector Behavior (BUG8). As well as the undocumented timeout behavior above, TraceLink was also able to detect the `raftkvs` model's well-documented perfect failure detector assumption, which is explained in its README. We replaced the failure detector with a realistic model, capable of reporting both false negatives and false positives, and the modified specification became able to consistently validate implementation traces.

9.2.6 Symbolic Abstractions (BUG9). Sometimes, an MPCal specification uses an obviously inaccurate definition to abstract a part of the system's environment. For `raftkvs`, this was the set of possible client requests, which was defined to contain 3 representative examples. This could not be validated against implementation traces, because the true set of valid requests is much more diverse.

	One-path			One-path-sidestep		
Configuration	**% Triggered**	**Runtime**	**Depth**	**% Triggered**	**Runtime**	**Depth**
raftkvs, n = 1	33.33%	14.4s	2666	50.00%	12.7s	1825
raftkvs, n = 2	78.33%	71.8s	10372	78.33%	61.3s	7587
raftkvs, n = 3	65.00%	148.7s	15103	70.00%	119.6s	10445
raftkvs, n = 3 +fail	86.67%	101.0s	12801	86.67%	88.8s	8456
raftkvs, n = 4	76.67%	387.9s	29053	78.33%	355.1s	21581
raftkvs, n = 5	80.00%	596.0s	39845	81.67%	730.0s	28824

Table 3. % of instances where a bug is caught, runtime, and counter-example depth when validating traces with bugs from Table 2 (all of which impact `raftkvs`), without and with the sidestep strategy.

Our fix was to generalize the request set definition to represent the correct format in general, while keeping the finite approximation as a model-checking only definition. We still need the model checking-only definition because it is impossible for TLC to enumerate the new infinite set when exploring possible request patterns. Keeping this type of approximation as a model checking detail is a best practice in TLA$^+$.

Overall, these bugs cover a broad set of issues in specification-implementation correspondence that were previously difficult or impossible to detect. We found TraceLink helpful in validating several pieces of PGo's trusted code, and we are now more confident in the systems we validated working as specified.

9.3 RQ2: Bug Finding Performance

To answer RQ2, we measured the effect of TraceLink's validation policies on the accuracy, runtime, and counter-example depth when finding the 9 bugs from Table 2, all of which impacted `raftkvs`.

Table 3 shows a summary of our results for the one-path and one-path-sidestep policies. The % of instances where a bug is caught means how often, out of all trace validation runs of that type, the underlying bugs caused a validation failure. Runtime means how long trace validation took to *fail*, when it did. Counter-example depth is how long, in TLA$^+$ state space terms (and elements from the recorded trace), the counter-example was on validation failure.

The most significant difference when enabling sidestep is a decrease in counter-example depth by, on average, 29%. On the other hand, the accuracy in detecting bugs changes relatively little. We explain this in terms of Figure 3. While in small examples it is possible for single-path validation to choose X, Y, Z and "get away with it" (as happens more often for 1-node `raftkvs`), for larger traces with more complex behaviors, if some other part of the system depended on choosing Y first, choosing X first will eventually cause a contradiction, and the bug will usually be detected. But, that contradiction may come hundreds or thousands of events later, leading to a difficult to understand counter-example. This is where we found sidestep to be especially useful: it is sometimes more efficient due to the shorter counter-example, but more importantly, its counter-examples pinpoint the exact state where choosing Y should have been possible, but was not (e.g., the initial state in Figure 3).

Comparing bug finding runtimes, notice that sidestep is often more efficient than single-path, but not always. For 5-node `raftkvs`, sidestep validation terminates over 2 minutes later, even when finding counter-examples that are 11,021 states shorter. This is because of the overhead of checking the sidestep condition, which requires iterating over all processes while asserting a variation of the next-state condition. It is therefore not surprising that, for the `raftkvs` instance with the largest

Configuration	Compressed Size	Naive Size	Compression
dqueue	9.6 KB	8.0 KB	0.8×
locksvc	13.4 KB	25.1 KB	1.9×
raftkvs, n=1	96.4 KB	2.0 MB	20×
raftkvs, n=2	169 KB	17.3MB	103×
raftkvs, n=3	176 KB	18.7 MB	106×
raftkvs, n=3 +fail	187 KB	17 MB	90×
raftkvs, n=4	190 KB	34 MB	181×
raftkvs, n=5	210 KB	49 MB	238×

Table 4. TLA$^+$ file sizes, with and without templatization, including the compression ratio.

number of MPCal processes (26 in total), the time penalty is greater than the savings from a shorter counter-example.

9.4 RQ3: TraceLink Performance

We consider two elements of TraceLink's Performance: its runtime relative to executing TLC on a single path, and templatization effectiveness.

Runtime relative to TLC. We measured the execution time of TLC checking one path of depth k in the MPCal model *without tracing*, and compare that with TraceLink's runtime for a path of the same length k. In essence, this tells us the overhead of TraceLink compared to plain model checking.

We found that, for traces with path length $23 \leq k \leq 47$, the overhead is not measurably different. For larger instances, like `rafkvs` $n = 3$, with a path length of $k = 20,485$, TraceLink had an average overhead of 34.8×, from 9.4s to 5m 28s. This is due to the inherent extra computation that TraceLink performs, including resolving vector clock ambiguity and asserting the entire `raftkvs` MPCal model on every step. In our future work, we plan to further improve TraceLink's runtime efficiency.

Templatization effectiveness. We now consider the effectiveness of templatization of the translated TLA$^+$, described in Section 6. Table 4 compares the output of two versions of TraceLink – with templatization (Compressed) and without (Naive). The numbers are calculated separately per system and distinct configuration we evaluate, and averaged over all collected traces from that group.

For traces with more than a few 10s of elements, the compression factor for the TLA$^+$ specifically is 10-100×. This keeps the TLA$^+$ output at a size that TLC can read. Overall, we find the gains from the technique conclusively outweigh the losses at any non-trivial scale.

9.5 RQ4: Instrumentation Performance

To answer RQ4, we considered the performance overhead of TraceLink's Go instrumentation. For each system and workload that generated our traces, we measured the overhead between the baseline non-instrumented execution, tracing without the stress testing, and tracing with randomized stress testing. We ran each workload 5 times. Across workloads, the executions we measured ran between 1 second and 47 seconds each.

Our results show a median overhead of 23% due to only tracing, and a median overhead of 37% when both tracing and stress-testing. We give medians here because measurement noise distorted the means, with some extreme outliers ranging from 578% to -33% (a speed-up relative to baseline). We believe the noise is caused by the inherent non-determinism underlying the distributed systems we tested.

9.6 RQ5: Manual Effort

To answer RQ5, we qualitatively identified the manual effort required to set up trace validation for a PGo prototype using TraceLink.

Compared to verifying a normal TLA$^+$ file, TraceLink requires one additional command-line invocation: running TraceLink. TraceLink accepts unmodified MPCal files, which contain all the necessary context to generate a working trace validation environment.

For comparison, the current state of the art in TLA$^+$ trace validation is work by Cirstea et al. [Cirstea et al. 2024]. We asked the authors of this work to validate the `locksvc` system using their manual methodology. They reported that it took them 2 hours: 1 hour to adapt the TLA$^+$ definitions, and 1 hour to adapt their additional scripts. Their validation was also able to detect BUG1. They speculate that, if they were asked to validate a large specification such as `raftkvs`, it would take significantly longer.

The automation advantage of TraceLink is especially important when considering software evolution. With changes to the specification, manual instrumentation and manual TLA$^+$ definitions would need to be updated. TraceLink, by contrast, could automate rebuilding the TLA$^+$ needed for validation, as long as the original specification was written in MPCal. We believe that the tighter loop between the model and the compiled implementation enabled by TraceLink will yield a much faster software engineering process.

10 RELATED WORK

Runtime Verification and Trace Validation. TraceLink can be thought of as an offline version of Runtime Verification (RV) for distributed systems [Falcone et al. 2021; Francalanza et al. 2018]. This line of work goes back to work by Sen and Garg [Sen and Garg 2003], which considers checking of partially ordered traces. There are three elements that distinguish TraceLink from prior work in RV. (1) We rely on PGo to guarantee that our instrumentation is general-purpose: we capture all possibly relevant events and state changes so that the trace can be validated against an arbitrary TLA$^+$ specification. (2) We use a formal model, normally unavailable in RV, to infer bindings between implementation traces and model semantics, automating our TLA$^+$ trace interpretation. (3) We rely on TLC [Kuppe et al. 2019] for efficient and general-purpose validation. Overall, our goal with TraceLink is to detect not just implementation issues, but also to find bugs in the model and the PGo compiler.

Cirstea et al. [Cirstea et al. 2024] is the most recent general-purpose work that we are aware of on trace validation for TLA$^+$. There are also industrial case studies with the CCF [Howard et al. 2025], etcd[10], and Zookeeper [Niu et al. 2022]. These all use manual or semi-automated approaches. TraceLink, by contrast, is fully automated assuming usage of PGo (Section 9.6).

Choreographic PlusCal [Foo et al. 2023] goes in a different direction, automatically projecting choreography-like TLA$^+$ specifications into runtime monitors, which assert system correctness at runtime. This technique only works on TLA$^+$ specifications that can be split into locally checkable assertions, however. TraceLink reconstructs a global view of the system being validated, and is therefore compatible with all TLA$^+$ assertions.

Capturing System Traces. Trace validation requires capturing traces of a distributed system execution. This has been previously explored by several tools [Beschastnikh et al. 2020; Mace et al. 2018; Sambasivan et al. 2014; Sigelman et al. 2010]. TraceLink's goal is different from these tools, and it achieves completeness of capturing externally-visible state transitions by making use of the PGo compiler.

[10]See pull request on Github https://github.com/etcd-io/raft/pull/113.

Improving Implementation Coverage. TraceLink requires diverse input traces. Many approaches exist for generating such traces from implementations [Dragoi et al. 2024; Kingsbury 2016; Microsoft 2025; Ozkan et al. 2018; Padhye et al. 2019]. Diverse implementation scenarios can also be derived from formal specifications using model-based testing or fuzzing [Dorminey 2020; Gulcan et al. 2024; Wang et al. 2023].

This prior work is complementary with TraceLink, as TraceLink would likely find more diverse bugs with traces produced by these approaches.

Checking distributed system implementations. There is a rich body of related work on building correct distributed systems. Some of his work operates directly on implementations [Deligiannis et al. 2021; Grant et al. 2018; Guo et al. 2011; Leesatapornwongsa et al. 2014; Lukman et al. 2019; Nelson et al. 2019; Ozkan et al. 2018, 2019; Simsa et al. 2010; Yang et al. 2009]. These tools are practical, but provide only limited guarantees due to scalability challenges. That said, many of these tools can be a rich source of diverse traces for TraceLink.

11 FUTURE WORK AND CONCLUSION

TraceLink and PGo form a closed loop, from an input formal model to an implementation, and from a trace of the implementation back to the formal model. In our future work, we aim to explore how this loop can be used to help infer a formal model for a given implementation with minimal guidance from the user.

In conclusion, we presented an approach to automatically validate a runtime trace of a distributed system against its formal model. We realized this approach in TraceLink, which we implemented for the PGo compiler and its MPCal/TLA$^+$ modeling languages. Unlike prior work, TraceLink is entirely automated.

We evaluated TraceLink on three models and two tracing path validation strategies, and we described the diverse set of 9 bugs that TraceLink uncovered. The key to finding these bugs was TraceLink's ability to identify implementation states not represented by the specification, as well as its ability to validate realistic behaviors that go deeper into model state space than normal exhaustive model checking could achieve. We also showed that TraceLink's TLA$^+$ generation method lets it scale to traces with up to 100,000 elements, and that the *sidestep* validation mode significantly reduces the depth at which TraceLink can find bugs.

12 DATA-AVAILABILITY STATEMENT

TraceLink is developed as an extension of PGo, which is already open-source[11]. The most recent version of PGo contains the entire TraceLink prototype. By extension, PGo contains all the systems we validated as demo systems, and the most recent version fixes all bugs found in this paper.

Our evaluation data and scripts are packaged separately [Hackett and Beschastnikh 2025]. This includes 400GB of captured logs and validation counter-examples, as well as the scripts and documentation to reproduce our experiments.

ACKNOWLEDGMENTS

This research was supported by the NSERC Discovery Grants (RGPIN-2020-05203, RGPIN-2025-06826). We thank Horatiu Cirstea and Stephan Merz for their advice, help during evaluation, and hospitality during initial stages of the project. We thank MongoDB, whose PhD Fellowship funded part of this work. We thank Antithesis for reproducing part of our experiments on their bug-finding platform, increasing our confidence in our results.

[11]https://github.com/DistCompiler/pgo

A VARIABLE LOOKUP RULES

We now explain how we use the MPCal instance declarations to relate the *ArchetypeName* . *VariableName* variable naming convention in the traces to TLA$^+$ definitions generated by PGo.

Note that a core assumption in this translation process is that each MPCal archetype is instantiated once per specification. This is universally true for MPCal specifications that currently exist – the same archetype having two inconsistent instantiations would imply that two conflicting interpretations of an archetype's environment co-exist at the same time, which we find unlikely. This assumption is the only way in which TraceLink does not generalize to all of MPCal; specifications that re-instantiate multiple configurations of the same archetype are rejected. A more complex configuration could be used to distinguish which MPCal `self` value belongs to which instantiation variant, but we leave this consideration as out of scope.

```
1  archetype AServer(ref network[_], ref glob, ref foo, bar)
2      variables msg;
3  {
4      \* ...
5  }
6  variables net = [0 ↦ ⟨⟩], global;
7
8  process (Server ∈ {0}) == instance AServer(ref net[_], ref global, 42, 11)
9      mapping net[_] via ReliableFIFOLink;
```

Listing 9. Example MPCal instance declaration

To define the variable name lookup semantics supported by TraceLink, we show all possible cases in Listing 9. It defines the `AServer` archetype, where the contents of Listing 4 might be found. From within `AServer`, 5 state variables are accessible, and a 6th is always implicit. Scoped from within `AServer`, they are: `.pc`, `AServer.network`, `AServer.glob`, `AServer.foo`, `AServer.bar`, and `AServer.msg`. They are respectively categorized as local (special case), mapping, global, local, local, and local. Note that we chose different names between parameters and global variables to avoid ambiguity; it is valid for `AServer.network` to refer to a global variable named `network`.

The first variable, `.pc`, is always assumed to exist in MPCal: the program counter itself. It is the only local variable that is not tied to any specific archetype, since it is inherent to all of them. Like in PlusCal, the corresponding TLA$^+$ state variable is always called `pc`. As a result, TraceLink uses the special case $[\![\texttt{.pc}]\!]^{\text{local}} = \texttt{pc}$.

For `AServer.network`, the `mapping` clause on line 9 declares that it is backed by the mapping macro `ReliableFIFOLink` (defined in Listing 3), and `ref net[_]` on line 8 gives it the underlying TLA$^+$ state variable `net`. The `[_]` suffix means that it has one *idx* (the repeated metavariable in Definition 6a and 6b) – if it had no `[_]`, it would have no *idx*, and if it had several `[_]`, it would have that number. Usually, mapped variables have either 0 or 1 *idx*. Given this information, we can infer that $[\![\texttt{AServer.network}]\!]^{\text{mapping}} = \texttt{net}$, and that traced reads and writes will be performed by `AServer_network_read(__state, __idx0, __value, __rest(_))` and `AServer_network_write(__state, __idx0, __value, __rest(_))`, respectively. Note that the dot in the variable name is converted to an underscore in order to produce a valid TLA$^+$ identifier.

`AServer.global` satisfies $[\![\texttt{AServer.glob}]\!]^{\text{global}} = \texttt{global}$, since it is passed as `ref global` on line 8. This means that traced reads and writes will be performed according to Definitions 5a-5c.

$$[\![stmt_1 \ ; \ stmt_2 \ ; \ stmts^*]\!]^{\text{stmt}}_{rest(\_)} \overset{\text{def}}{=} \boxed{[\![stmt_1]\!]^{\text{stmt}}_{\texttt{LAMBDA \_\_state : } [\![stmt_2 \ ; \ stmts^*]\!]^{\text{stmt}}_{rest(\_)}}} \tag{7}$$

`AServer.foo` is an unusual edge case of MPCal, where a constant value `42` is passed on line 8 to the parameter `ref foo` defined on line 1. In this case, the name of `ref foo` is used to define an implicit local variable of the same name, initialized to `42`. This case satisfies $[\![\texttt{AServer.foo}]\!]^{\text{local}} = \texttt{foo}$, and like the other two below, reads and writes will be performed according to Definitions 4a-4c.

`AServer.bar` is a by-value archetype parameter, which is initially bound to the value `11` on line 8. It acts as a local variable definition, satisfying $[\![\texttt{AServer.bar}]\!]^{\text{local}} = \texttt{bar}$.

`AServer.msg` is encapsulated within the `AServer` archetype on line 2. As a regular local variable, it satisfies $[\![\texttt{AServer.msg}]\!]^{\text{local}} = \texttt{msg}$.

Looking at all these definitions, notice that the TLA$^+$ translation removes the archetype name prefix. This can cause name collisions. We resolve these name collisions by imitating the PlusCal generator, since it generates the TLA$^+$ that TraceLink should match against. The tie-breaker is to append an incrementing number for each repetition of a state variable name, from earliest to latest in the input file. Assuming a file has 3 different archetypes, each with a local variable called `msg`, then their TLA$^+$ translations will be renamed according to the sequence `msg`, `msg0`, and `msg1`.

B MAPPING MACRO TRANSLATION

To provide implementations for the operators called in Definitions 6a and 6b, we translate the mapping macros references in the MPCal model into TLA$^+$. This is not the same translation used in PGo, since it has a different purpose. In PGo, the mapping macros are woven into the algorithm for model checking at the PlusCal level. On the other hand, TraceLink requires a version of those macros that is separated from the rest of the model, and which can be used in the continuation-passing context we have defined so far.

Many of our translations are the same standard rules common to PlusCal[], so we highlight cases unique to our translation, or unique to handling MPCal constructs. As in PlusCal and MPCal, by default, TLA$^+$ expressions are passed through unchanged during translation. To distinguish between MPCal statements and expression context, we mark rules with a superscript of "stmt" or "expr". Usage of *var*, *idx**, and *value* refer contextually to parameters of the expansion implied by Definitions 6a and 6b. That is, the translation is rooted at a synthetic TLA$^+$ operator definition of the form $var\texttt{\_read(\_\_state, \_\_idx}^*\texttt{, \_\_value, \_\_rest(\_))} == [\![stmts^*]\!]^{\text{stmt}}_{\texttt{\_\_rest(\_)}}$, and the identical corresponding `_write` form. The statements being translated are those in the related mapping macro's `read { stmts^* }` and `write { stmts^* }` blocks, respectively.

We combine the definitions of the read and write forms, since they are identical except for in a few special cases. For definitions specific to read and write contexts, their label is suffixed with either "/read" or "/write", as in "expr/read", "stmt/write", and so forth.

Note also that mapping macros, unlike general MPCal, do not allow `goto`, `while` loops, or labels. This means that MPCal mapping macros may only contain straight-line code and branches, so we do not consider those more complex features in this work.

Definition 7 shows that sequences of MPCal statements are composed in the same way as sequential trace events (Definition 3) – by continuation passing.

Definitions 8a-8d show the translations of all PlusCal's control flow statements that are supported in an MPCal mapping macro context. Notice that branching control flow is handled by duplicating the continuation. To keep the translation simple, TraceLink mirrors PGo's own behavior of duplicating control flow branches when compiling control flow. This can lead to code size blow-up, but

$$[\![\texttt{if}\ (e_c)\ \{stmts_1^*\}\ \texttt{else}\ \{stmts_2^*\}]\!]^{\text{stmt}}_{rest(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{ll}\texttt{IF} & [\![e_c]\!]^{\text{expr}} \\ \texttt{THEN} & [\![stmts_1^*]\!]^{\text{stmt}}_{rest(\_)} \\ \texttt{ELSE} & [\![stmts_2^*]\!]^{\text{stmt}}_{rest(\_)}\end{array}} \tag{8a}$$

$$[\![\texttt{either}\ \{stmts_1^*\}\ \texttt{or}\ \{stmts_2^*\}^*]\!]^{\text{stmt}}_{rest(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{l}\lor\ [\![stmts_1^*]\!]^{\text{stmt}}_{rest(\_)} \\ \lor\ [\![stmts_2^*]\!]^{\text{stmt}}_{rest(\_)} \\ \lor\ ^*\end{array}} \tag{8b}$$

$$[\![\texttt{with}\ (name = e)\ \{stmts^*\}]\!]^{\text{stmt}}_{rest(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{ll}\texttt{LET} & name == [\![e]\!]^{\text{expr}} \\ \texttt{IN} & [\![stmts^*]\!]^{\text{stmt}}_{rest(\_)}\end{array}} \tag{8c}$$

$$[\![\texttt{with}\ (name \in e)\ \{stmts^*\}]\!]^{\text{stmt}}_{rest(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{l}\exists name \in [\![e]\!]^{\text{expr}} : \\ \quad [\![stmts^*]\!]^{\text{stmt}}_{rest(\_)}\end{array}} \tag{8d}$$

$$[\![\texttt{\$variable}[e_i]^* := e_v]\!]^{\text{stmt}}_{rest(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{ll}\texttt{LET} & \texttt{\_\_next\_state} == \\ & \quad [\texttt{\_\_state EXCEPT} \\ & \quad !.[\![var]\!]^{\text{mapping}}[\texttt{\_\_idx}]^*[[\![e_i]\!]^{\text{expr}}]^* = [\![e_v]\!]^{\text{expr}}] \\ \texttt{IN} & rest(\texttt{\_\_next\_state})\end{array}} \tag{9a}$$

$$[\![\texttt{\$variable}]\!]^{\text{expr}} \overset{\text{def}}{=} \boxed{\texttt{\_\_state}.[\![var]\!]^{\text{mapping}}[\texttt{\_\_idx}]^*} \tag{9b}$$

fortunately mapping macros and individual critical sections in MPCal are not usually long, so it is not common to see this issue in practice.

Given that this translation emits TLA$^+$ directly, as opposed to PGo's architecture which operates on a PlusCal representation (to which what we write below is inapplicable), we believe it is possible to avoid code duplication with a small modification to the branch handling described above. Rather than syntactically expanding the *rest(_)* continuation into each branch, we can emit a local operator `__rest(__state) ==` $[\![stmts^*]\!]^{\text{stmt}}_{rest(\_)}$, which contains one syntactic copy of the continuation. Then, we can reference that local operator by name as the continuation for each branch. Due to the reduced TLA$^+$ legibility and potentially large number of indirections, however, we do not use this approach in our implementation. We leave improving this approach to the point of practicality as future work.

Definitions 9a and 9b describe the read and write semantics for the special mapping macro expression `$variable`. This expression refers to the mapping macro's underlying state variable, including any indices `__idx` that may have been applied to it outside the mapping macro.

Definition 9a shows the semantics of a PlusCal write statement referencing the variable, which are very similar to 5b. The key differences are the addition of `[__idx]`* and the translation of e_i and e_v. Unlike an implementation trace, where the indices and value are fully evaluated TLA$^+$ expressions ready to insert, an MPCal specification's sub-expressions may require additional translation. Additionally, the `__idx` arguments are unique to mapping macros: they are the originally applied indices, in this case coming from idx^* in the trace, to which the additional mapping macro-internal indices e_i are appended. We write them at `__idx` in order to distinguish that they are parameters to our compiled mapping macro, and not metavariables like *idx*.

$$[\![\texttt{\$value}]\!]^{\text{expr/write}} \overset{\text{def}}{=} \boxed{\texttt{\_\_value}} \tag{10}$$

$$[\![\texttt{yield}\ e]\!]^{\text{stmt/read}}_{\mathit{rest}(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{l} \land\ \texttt{\_\_value} = [\![e]\!]^{\text{expr}} \\ \land\ \mathit{rest}(\texttt{\_\_state}) \end{array}} \tag{11a}$$

$$[\![\texttt{yield}\ e]\!]^{\text{stmt/write}}_{\mathit{rest}(\_)} \overset{\text{def}}{=} \boxed{\begin{array}{ll} \texttt{LET} & \texttt{\_\_next\_state ==} \\ & \quad \texttt{[\_\_state EXCEPT} \\ & \quad\ \texttt{!.}\,[\![\mathit{var}]\!]^{\text{mapping}}\,\texttt{[\_\_idx]}^{*} = [\![e]\!]^{\text{expr}}\texttt{]} \\ \texttt{IN} & \mathit{rest}(\texttt{\_\_next\_state}) \end{array}} \tag{11b}$$

Definition 9b describes the translation of `$variable` in expression position. In this situation, the result of the expression is the value of the state variable, indexed from `__state`. As explained above, `__idx`* is a sequence of 0 or more pre-applied indices from outside the mapping macro, used here to index further into the state variable's value.

When in a write context, mapping macros have access to `$value`, the value that is being written. Definition 10 shows that, in TraceLink's compiled version of mapping macros, this value is always the macro operator's parameter `__value`, which originally comes from the tracing data.

The `$value` expression is not available in a mapping macro's read context, so we do not define it.

In a read context, Definition 11a shows that the `yield` statement translates to an assertion that the expected value, `__value`, is equal to the translation of the expression e. Any side-effects the read operation has must be achieved by separately assigning to `$variable`, so the `yield` itself just marks the value that the algorithm should receive from reading a given mapped variable, which should always be equal to `__value`.

In a write context, Definition 11b shows that the `yield` statement acts like a simplified form of the assignment rule given by Definition 9a. The write interpretation of `yield` is to set the underlying state variable, indexed by 0 or more indices `__idx`, to the value of e. Unlike an assignment to `$variable`, no additional indices e_i may be specified.

B.1 Mapping Macro Translation Example

```
AServer_network_read(__state0, __idx0, __value, __rest(_)) ==
    ∧ Len(__state0.net[__idx0]) > 0
    ∧ LET readMsg == Head(__state0.net[__idx0])
            __state1 == [__state0 EXCEPT !.net[__idx0] =
                             Tail(__state0.net[__idx0])]
       IN  ∧ __value = readMsg
           ∧ __rest(__state1)
```

Listing 10. Translation of the read section of Listing 3

Returning to the `ReliableFIFOLink` mapping macro invoked on line 4, Listing 10 shows how the mapping macro's read operation (given in MPCal by Listing 3, lines 2-8) is translated into TLA$^+$ using our definitions.

The top-level definition is named after the MPCal variable being read or written – this means the same mapping macro can be translated multiple times, applied to different TLA$^+$ state variables. As defined in Listing 9, this instantiation has one index, `__idx0`. As with trace translations, to satisfy TLA$^+$ naming constraints, `__state` and `__next_state` in the definitions are interpreted as sequentially numbered names, starting with the state parameter `__state0`. `__value` is the expected read value given by expanding Definition 6a, and `__rest(_)` is the continuation, which should be called with the final `__state`$_n$.

All pairs of sequenced statements are reduced using Definition 7, which composes the individual statement semantics using continuation-passing. As before, immediately applied `LAMBDA` expressions are expanded syntactically.

Line 2 comes from the initial `await` statement, which is compiled using the standard PlusCal rules, except for the effect of Definition 9b, which replaces `$variable` with a reference to the `net` global variable in `__state0`, at the correct index `__idx0`. Line 3 is an application of Definition 8c to translate the `with` statement that follows. Definition 9b is applied once again to the bound expression to expand `$variable`. Lines 4 and 5 are an instance of Definition 9a, which implements a write to `$variable` (which still refers to `net` at `__idx0`), creating an updated `__state1`. Definition 9b is used again to expand `$variable` in the assigned value. Note that we combine the bindings on lines 3 and 4 for presentation – an exact interpretation of the two rules would produce nested `LET` expressions, which is equivalent. Line 6 is an application of Definition 11a to the mapping macro's `yield` statement, comparing `readMsg` to the `__value` that was seen by the trace. Line 7 commits `__state1` as the final state within the mapping macro, passing it to `__rest`. In this example, `__rest` would resolve to the `LAMBDA` whose body starts on line 5 of Listing 6.

REFERENCES

Martin Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*. 165–175.

Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. *Introduction to Runtime Verification.*

Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2020. Visualizing Distributed System Executions. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 9 (Mar 2020), 38 pages.

James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

Marc Brooker and Ankush Desai. 2025. Systems Correctness Practices at AWS: Leveraging Formal and Semi-formal Methods. *Queue* 22, 6 (2025), 79–96.

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. 167–178.

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (Feb. 2020), 36 pages.

Horatiu Cirstea, Markus A. Kuppe, Benjamin Loillier, and Stephan Merz. 2024. Validating Traces of Distributed Programs Against TLA$^+$ Specifications. In *22nd Intl. Conf. Software Engineering and Formal Methods (SEFM 2024) (LNCS, Vol. 15280)*, Alexandre Madeira and Alexander Knapp (Eds.). Springer, Aveiro, Portugal, 126–143.

Pantazis Deligiannis, Narayanan Ganapathy, Akash Lal, and Shaz Qadeer. 2021. Building Reliable Cloud Services Using Coyote Actors. In *Proceedings of the ACM Symposium on Cloud computing (SoCC)*.

Star Dorminey. 2020. Kayfabe – model-based program testing. https://conf.tlapl.us/2020/

Cezara Dragoi, Srinidhi Nagendra, and Mandayam Srivas. 2024. A Domain Specific Language for Testing Distributed Protocol Implementations. In *12th Intl. Conf. Networked Systems (NETYS 2024) (LNCS, Vol. 14783)*, Armando Castañeda, Constantin Enea, and Nirupam Gupta (Eds.). Springer, Rabat, Morocco, 100–117.

Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems, Summer School Marktoberdorf.*

Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. 2021. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.* 23, 2 (April 2021), 255–284.

Colin J. Fidge. 1988. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *ACSC*. University of Queensland, Australia, 55–66.

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Symposium on Principles of Programming Languages (POPL)*.

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

Darius Foo, Andreea Costea, and Wei-Ngan Chin. 2023. Protocol Conformance with Choreographic PlusCal. In *Theoretical Aspects of Software Engineering*, Cristina David and Meng Sun (Eds.). Springer Nature Switzerland, Cham, 126–145.

Adrian Francalanza, Jorge A. Pérez, and César Sánchez. 2018. *Runtime Verification for Decentralised and Distributed Systems.*

Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. 2018. Inferring and Asserting Distributed System Invariants. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

Ege Berkay Gulcan, Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Srinidhi Nagendra. 2024. Model-guided Fuzzing of Distributed Systems. *CoRR* abs/2410.02307 (2024).

Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

Finn Hackett and Ivan Beschastnikh. 2025. *TraceLinking Implementations with their Verified Designs (Evaluation)* . https://doi.org/10.5281/zenodo.16926533

Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. 2023a. Compiling Distributed System Models with PGo. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. 2023b. Understanding Inconsistency in Azure Cosmos DB with TLA+. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 1–12.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2017. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM* 60, 7 (2017), 83–92.

Heidi Howard, Markus A. Kuppe, Edward Ashton, Amaury Chamayou, and Natacha Crooks. 2025. Smart Casual Verification of the Confidential Consortium Framework. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Marian Hristov and Annette Bieniusa. 2024. Erla+: Translating TLA+ Models into Executable Actor-Based Implementations. In *International Workshop on Erlang*.

Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. 2024. Comprehensive Memory Safety Validation: An Alternative Approach to Memory Safety. *IEEE Security & Privacy* 22, 4 (2024), 40–49.

Kyle Kingsbury. 2016. Jepsen. https://jepsen.io/

Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ Model Checking Made Symbolic. *Proceedings of the ACM on Programming Languages (OOPSLA)* 3, Article 123 (2019), 30 pages.

Igor Konnov, Markus Kuppe, and Stephan Merz. 2022. Specification and Verification with the TLA+ Trifecta: TLC, Apalache, and TLAPS. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*.

Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. 2019. The TLA+ Toolbox. *Electronic Proceedings in Theoretical Computer Science* 310 (2019), 50–62.

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)* 21, 7 (1978), 558–565.

Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 872–923.

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Leslie Lamport. 2018. A PlusCal's User Manual. Online material.

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Trans. Comput. Syst.* 35, 4, Article 11 (Dec. 2018), 28 pages.

Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. 215–226.

Stephan Merz and Hernán Vanzetto. 2012. Automatic Verification of TLA + Proof Obligations with SMT Solvers. In *Logic for Programming, Artificial Intelligence, and Reasoning*. 289–303.

Microsoft. 2025. Azure Chaos Studio. https://azure.microsoft.com/en-ca/services/chaos-studio/ Accessed: 2025-03-25.

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Communications of the ACM (CACM)* 58, 4 (2015), 66–73.

Zhi Niu, Luming Dong, Yong Zhu, and Li Chen. 2022. Verifying Zookeeper based on Model-Based runtime Trace-Checking using TLA+. In *Proceedings of the 7th International Conference on Cyber Security and Information Engineering (ICCSIE '22)*. 13–18.

Diego Ongaro. 2014. *Consensus: Bridging theory and practice.* Stanford University.

Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 160 (Oct. 2018), 28 pages.

Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. 2019. Trace aware random testing for distributed systems. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 180 (Oct. 2019), 29 pages.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *28th ACM SIGSOFT Intl. Symp. Software Testing and Analysis (ISSTA 2019)*, Dongmei Zhang and Anders Møller (Eds.). ACM, Beijing, China, 329–340.

Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the Annual Symposium on Foundations of Computer Science*.

Giles Reger and Klaus Havelund. 2016. What Is a Trace? A Runtime Verification Perspective. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*.

Raja Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory Ganger. 2014. So, You Want to Trace Your Distributed System? Key Design Insights from Years of Practical Experience.

Alper Sen and Vijay K. Garg. 2003. Partial Order Trace Analyzer (POTA) for Distributed Programs. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003), 22–43. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc.

Jiri Simsa, Randy Bryant, and Garth Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems. In *Proceedings of the 5th International Conference on Systems Software Verification (SSV).*

Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model Checking Guided Testing for Distributed Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23).* 127–143.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. *ACM SIGPLAN Notices* 50, 6 (Jun 2015), 357–368.

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI).*

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. *SIGPLAN Not.* 46, 6 (June 2011), 283–294.

Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods.* 54–66.

Received 2025-03-26; accepted 2025-08-12