Validating Traces of Distributed Programs Against TLA+ Specifications

Horatiu Cirstea, Markus A Kuppe, Benjamin Loillier, Stephan Merz

▶ **To cite this version:**

Horatiu Cirstea, Markus A Kuppe, Benjamin Loillier, Stephan Merz. Validating Traces of Distributed Programs Against TLA+ Specifications. 2024. hal-04813639

HAL Id: hal-04813639
https://inria.hal.science/hal-04813639v1

Preprint submitted on 2 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

arXiv:2404.16075v2 [cs.PL] 17 Sep 2024

Validating Traces of Distributed Programs Against TLA$^+$ Specifications*

Horatiu Cirstea[1], Markus A. Kuppe[2], Benjamin Loillier[1], and Stephan Merz[1]

[1] University of Lorraine, CNRS, Inria, LORIA, Nancy, France
[2] Microsoft Research

Abstract. TLA$^+$ is a formal language for specifying systems, including distributed algorithms, that is supported by powerful verification tools. In this work we present a framework for relating traces of distributed programs to high-level specifications written in TLA$^+$. The problem is reduced to a constrained model checking problem, realized using the TLC model checker. Our framework consists of an API for instrumenting Java programs in order to record traces of executions, of a collection of TLA$^+$ operators that are used for relating those traces to specifications, and of scripts for running the model checker. Crucially, traces only contain updates to specification variables rather than full values, and developers may choose to trace only certain variables. We have applied our approach to several distributed programs, detecting discrepancies between the specifications and the implementations in all cases. We discuss reasons for these discrepancies, best practices for instrumenting programs, and how to interpret the verdict produced by TLC.

1 Introduction

Distributed systems are at the heart of modern cloud services and they are known to be error-prone, due to phenomena such as message delays or failures of nodes and communication networks. Applying formal methods during the design and development of these systems can help increase the confidence in their correctness and resilience. For example, the TLA$^+$ [18] specification language and verification tools have been successfully used in industry [20,24] for designing distributed algorithms underlying modern cloud systems. TLA$^+$ and similar specification formalisms are most useful for describing and analyzing systems at high levels of abstraction, but they do not provide much support for validating actual implementations of these systems. Although TLA$^+$ supports a notion of refinement, formally proving a chain of refinements from a high-level design of a distributed algorithm to an actual implementation would be a daunting task, complicated by the fact that standard programming languages do not provide explicit control of the grain of atomicity of the running program. In this work, we present a lightweight approach to validating distributed programs against high-level specifications that relies on recording finite traces of program executions

* This work was partly supported by a grant from Oracle Corporation.

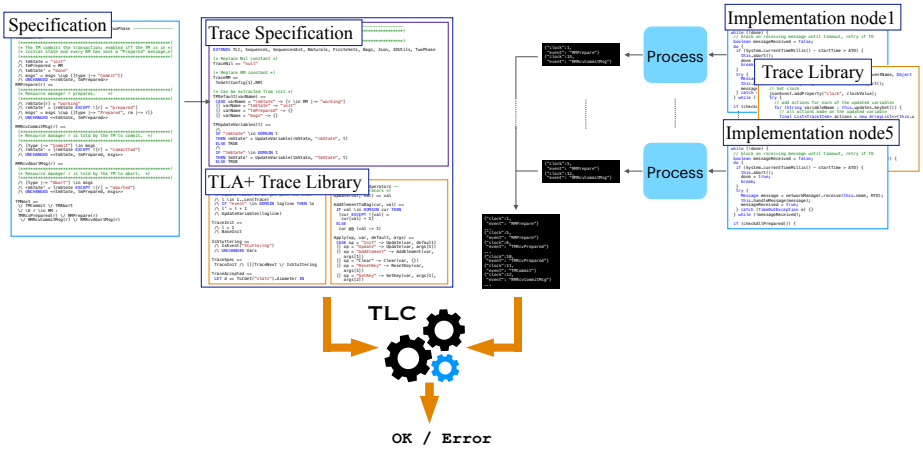

Fig. 1: Overview of trace validation.

and leveraging the TLA$^+$ model checker TLC [29] for comparing those traces to the state machine described in the TLA$^+$ specification. Although this approach does not provide formal correctness guarantees, even when the TLA$^+$ specification has been extensively verified, we have found it very useful for discovering and analyzing discrepancies between the runs of distributed programs and their high-level specifications. We have thus been able to discover serious bugs that had gone undetected by more traditional quality assurance techniques.

Figure 1 summarizes the approach. The starting point consists of a distributed Implementation and of the Specification, written in TLA$^+$, it is supposed to implement. We have designed a Trace Library that facilitates the instrumentation of Java programs in order to record information on how program operations correspond to transitions described in the TLA$^+$ specification, including updates of its variables. Executing the instrumented programs produces traces in JSON format that are aggregated into a single file. We also provide a TLA$^+$ Trace Library that helps write a Trace Specification, extending the original specification. The model checker TLC is then used to check the program trace against the trace specification. Although our trace library was developed for Java, the overall approach can easily be adapted to any other imperative language. We also implemented a few features in TLC that support our approach.

Because the traces record the evolution of the state of the TLA$^+$ specification, our approach is easiest to apply when the specification exists prior to building the implementation, the implementor is familar with it, and uses it as a blueprint when writing the code. However, we have also used the approach in order to "reverse engineer" a TLA$^+$ specification from an existing distributed program and better understand its operation. Trace validation can also help ensure that the specification and implementation remain in sync over time because it is easy to apply it again in case of changes to the specification or the implementation.

The main problem when instrumenting a program is to identify suitable "linearization points" at which the program completes a step that corresponds to an atomic transition of the high-level state machine. Basic guiding principles are to log an event when shared state has been updated, such as when sending or receiving messages, performing operations on locks or on stable data storage. Feedback from trace validation can help with adapting the instrumentation in order to take into account different grain of atomicity between the specification and the implementation, as discussed later. Because data representation generally differs between the TLA$^+$ specification and the actual program, it may be difficult or impractical to compute the value of a specification variable (or its update) corresponding to the data manipulated by the implementation. We therefore allow traces to be incomplete and only record some information about the corresponding abstract state. We reduce the problem of trace validation to one of constrained model checking and show how TLC can reconstruct missing information. This leads to a tradeoff between the precision of information recorded in the trace (and potentially of the verdict of validation) and the amount of search that TLC must perform during model checking.

The paper[3] is organized as follows: Section 2 provides some background on TLA$^+$ and introduces our running example. Our approach to instrumentation is described in Section 3. In Section 4 we formalize the trace validation problem, describe how we realized the approach using TLC, and discuss our experience with it. Section 5 discusses related work, and Section 6 concludes the paper and presents some perspectives for future work.

2 Background

2.1 TLA$^+$ Specifications

TLA$^+$ [18] is a specification language based on Zermelo-Fraenkel set theory and linear-time temporal logic that has found wide use for writing high-level specifications of concurrent and distributed algorithms. It emphasizes the use of mathematical descriptions based on sets and functions for specifying data structures. In TLA$^+$, the state space of a system is represented using variables, and formulas are evaluated over *behaviors*, i.e., sequences of states that assign values to variables. Algorithms are described as state machines whose specifications are written in the canonical form $Init \wedge \Box[Next]_{vars} \wedge L$. In this formula, *Init* is a state predicate describing the possible initial states of the system, *Next* represents the next-state relation, usually written as the disjunction of actions describing the possible state transitions, *vars* is a tuple containing all state variables that appear in the specification, and L is a temporal formula asserting liveness and fairness assumptions. A state predicate is a formula of first-order logic that is evaluated over single states. A transition predicate (or, synonymously, action) is a first-order formula that may contain unprimed and primed occurrences of variables. Such a formula is evaluated over a pair of states, with unprimed variables

[3] This is an extended version of [**?**].

```
CONSTANT RM
VARIABLES rmState, tmState, tmPrepared, msgs
vars ≜ ⟨rmState, tmState, tmPrepared, msgs⟩
Messages ≜ [type : {"Prepared"}, rm : RM] ∪ [type : {"Commit", "Abort"}]
TPInit ≜
   ∧ rmState = [r ∈ RM ↦ "working"] ∧ tmState = "init"
   ∧ tmPrepared = {} ∧ msgs = {}
RMPrepare(r) ≜
   ∧ UNCHANGED ⟨tmState, tmPrepared⟩ ∧ rmState[r] = "working"
   ∧ rmState' = [rmState EXCEPT ![r] = "prepared"]
   ∧ msgs' = msgs ∪ {[type ↦ "Prepared", rm ↦ r]}
RMRcvCommitMsg(r) ≜
   ∧ UNCHANGED ⟨tmState, tmPrepared, msgs⟩ ∧ [type ↦ "Commit"] ∈ msgs
   ∧ rmState' = [rmState EXCEPT ![r] = "committed"]
RMRcvAbortMsg ≜ ...
TMRcvPrepared(r) ≜
   ∧ UNCHANGED ⟨rmState,tmState,msgs⟩ ∧ tmPrepared' = tmPrepared ∪ {r}
   ∧ tmState = "init" ∧ [type ↦ "Prepared", rm ↦ r] ∈ msgs
TMCommit ≜
   ∧ UNCHANGED ⟨rmState, tmPrepared⟩ ∧ tmState = "init" ∧ tmPrepared = RM
   ∧ tmState' = "done" ∧ msgs' = msgs ∪ {[type ↦ "Commit"]}
TMAbort ≜ ...
TPNext ≜
   ∨ TMCommit ∨ TMAbort
   ∨ ∃ r ∈ RM : ∨ RMPrepare(r) ∨ TMRcvPrepared(r)
                ∨ RMRcvCommitMsg(r) ∨ RMRcvAbortMsg(r)
Spec ≜ TPInit ∧ □[TPNext]_vars
Consistent ≜
   ∀ r1,r2 ∈ RM: ¬(rmState[r1] = "aborted" ∧ rmState[r2] = "committed")
```

Fig. 2: TLA$^+$ Specification of Two-Phase Commit.

referring to the values before the transition and primed variables to the values after the transition. The formula $[Next]_{vars}$ holds of a pair of states $\langle s, t\rangle$ if either *Next* holds of $\langle s, t\rangle$ (and therefore the pair represents an actual step of the system) or the tuple *vars* evaluates to the same value in the two states (and the pair represents a stuttering step). Systematically allowing for stuttering steps enables the refinement of a specification S by another specification I written at a lower level of abstraction to be represented as the validity of the implication $I \Rightarrow S$. The complementary property L is used to express fairness assumptions and is at the basis of verifying liveness properties of algorithms. Since in this work we only analyze finite traces of programs, we ignore liveness properties and are interested in finite behaviors, i.e., sequences $s_0 \ldots s_n$ of states such that *Init* holds of s_0 and $[Next]_{vars}$ holds for all pairs $\langle s_i, s_{i+1}\rangle$ for $i \in 0\,..\,n-1$.

As a running example for this paper, Fig. 2 contains an excerpt of the TLA$^+$ specification of the well-known Two-Phase Commit protocol where a transaction manager (TM) helps a set of resource managers (RMs) reach agreement on whether to commit or abort a transaction: the RMs send messages indicating that they are prepared to commit, while the TM listens for such messages and based on the votes of the participants broadcasts a commit or an abort message. This specification is part of a collection of example TLA$^+$ modules.[4] The module first declares a constant parameter `RM` that represents the set of RMs and four variables representing the control states of the RMs (represented as a function with domain `RM`) and of the TM, the set of RMs that have declared their preparedness to carry out the transaction, and the set of messages that have been sent during the protocol. The initial state of the system is described by the predicate `TPInit`: every RM is in state `"working"`, the TM in state `"init"`, and the sets of prepared RMs and of messages are empty. The following operators define actions that describe individual state transitions. For example, `RMPrepare(r)` represents an RM `r` declaring its preparedness to carry out the transaction by moving to control state `"prepared"` and adding a corresponding message to the set of messages `msgs`. The action `TPNext` corresponds to the next-state relation of the state machine, defined as the disjunction of the previously defined actions, and formula `Spec` represents the overall specification. The TLA$^+$ tools, including the model checker TLC and the proof assistant TLAPS [2], can be used to verify properties of the specification, including the invariant `Consistent` that RMs must agree about committing or aborting a transaction.

2.2 Java Implementation

A possible Java implementation of the resource managers is presented in Fig. 3. Only a simplified version of the main method is shown, the auxiliary methods are faithful Java translations of the actions in the TLA$^+$ specification.[5]

An RM is identified by a `name` and uses a `network` (manager) to send and receive messages. Once it completes its task (represented by method `working`), it sends a message to the TM indicating that it is prepared to commit and waits for a reply. The method `handleMessage` causes the transaction to be committed or aborted, according to the decision received from the TM. If no reply is received before the `RECEIVE_TIMEOUT`, the RM resends its prepared message to the TM.

3 Instrumenting Distributed Programs

Our objective in this work is to check traces of program executions against a TLA$^+$ specification of the algorithm the program is expected to implement. In order to obtain such traces (in JSON format), we instrument implementations,

[4] `https://github.com/tlaplus/Examples/tree/master/specifications/transaction_commit`; the complete specification is also given in the appendix.

[5] The full implementation is available at `https://github.com/lbinria/TwoPhase`. The main method of the TM is given in the appendix.

```
public class ResourceManager {
  String name, tmName;
  ResourceManagerState state;
  NetworkManager network;
  public void run() throws IOException {
     working();
     while (true) {
        sendPrepared();
        try {
           Message message = network.receive(name, RECEIVE_TIMEOUT);
           handleMessage(message);
           return;
        } catch (TimeOutException e) {}
} } }
```

Fig. 3: Java implementation of the RM of the Two-Phase Commit protocol.

registering changes made to specification variables and logging them to a file, together with a timestamp. Section 4 describes the structure of TLA$^+$ trace specifications used to process the traces generated in this way.

The main class of the Java library we designed, TLATracer, essentially provides two methods: `notifyChange` for tracking variable updates, and `log` used to produce one log entry in the trace file that reflects all the variable changes recorded with `notifyChange` since the last call to `log` (or since the start of the process if `log` was never called before).[6]

```
void notifyChange(String var, List<String> path,
                  String op, List<Object> args);
long log(String eventName, Object[] args, long clockValue);
```

Updates to variables are tracked using the method `notifyChange`, which allows the programmer to specify an update to several fields (identified using the `path` argument) of the TLA$^+$ variable `var` by applying the operation `op` to the old field value and the arguments `args`. Our library supports operations such as updating the variable by a new value, adding or removing a value to or from a set or bag (multi-set) etc. In its general form, the `log` method records the variable changes as well as the name and optionally the parameters of the corresponding TLA$^+$ action. The time when the log has been performed is used as a timestamp for the corresponding entry. The library provides different types of clocks (in-memory, file-based or server-based) that can be used with the tracer, in which case the argument need not be indicated explicitly, but it should be specified when the

[6] A more detailed presentation of the library is given in the appendix. The library available at `https://github.com/lbinria/trace_validation_tools/` offers more convenience methods to track variable changes and log events.

implementation uses a logical clock. The library currently supports scalar logical clocks but vector clocks can easily be added.

For example, in the `sendPrepared` method, an RM sets its state to `"prepared"` and sends a corresponding message to the TM (lines 2 and 6, respectively):[7]

```
1 void sendPrepared()  {
2   state = ResourceManagerState.PREPARED;
3   tracer.notifyChange("rmState", {name}, "Update", {"prepared"});
4   tracer.notifyChange("msgs", {}, "Add", {"type":"Prepared", "rm":name});
5   tracer.log("RMPrepare", {name});
6   networkManager.send(new Message(name, tmName, "Prepared", 0));
7 }
```

The remaining lines are used for tracing purposes. Line 3 records the change of the entry corresponding to the RM executing `sendPrepared` (i.e. `this.name`) of the TLA+ variable `rmState` to the new value `"prepared"`. Similarly, line 4 indicates that a message of type `"Prepared"` from the current RM is added to the set `msgs`. Finally, in line 5 these changes are logged as corresponding to the TLA+ action `RMPrepare` with the current RM as parameter. For an RM named `"rm-0"`, the above code produces the following log entry in JSON format:[8]

```
{ "clock": 4,
  "rmState": [ { "op": "Update", "path": ["rm-0"], "args": ["prepared"] } ],
  "msgs": [ { "op": "Add", "path": [],
              "args": ["type":"Prepared","rm":"rm-0"] } ],
  "event": "RMPrepare", "event_args": ["rm-0"] }
```

It should be noted that it is possible to trace updates to only a subset of specification variables. Similarly, indicating the name and the arguments of the TLA+ action is optional. For example, the size of the value of a variable might be prohibitively large, or the value might simply be unknown, for example because the implementation handles encrypted values. In the above example, either or both calls to `notifyChange` could have been omitted. As discussed in Section 4, the model checker will fill in suitable values for variables omitted from tracing. However, providing more detailed information can lead to more efficient validation and strengthen confidence in the results.

The library also provides Python scripts for merging the trace files produced by different processes and for validating the resulting trace using TLC.

4 Checking Program Traces Against TLA+ Specifications

Having obtained a log from an execution of the distributed program, we must check if this log matches some behavior of the TLA+ state machine specification that the program is expected to implement. We define the problem more formally, explain how we realize trace validation, report our experience with this approach, and discuss some technical aspects of trace validation.

[7] We take some syntactic liberties, such as writing `{...}` instead of `List.of(...)`.
[8] The JSON schema of a trace entry is given in the appendix.

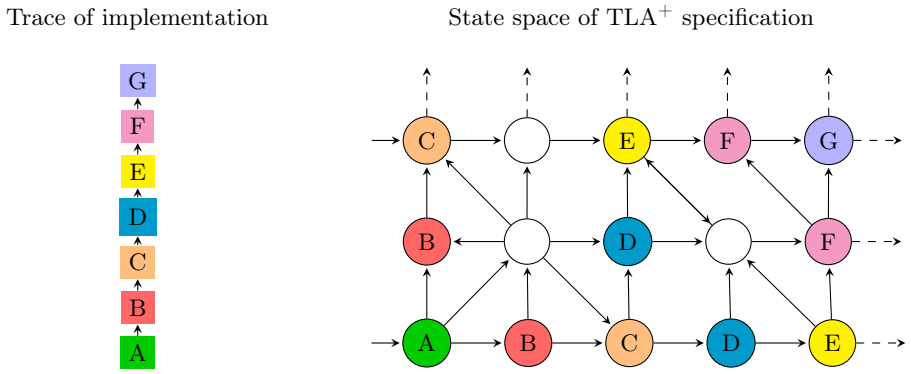

Fig. 4: Trace validation as a search for paths in the state space.

4.1 The trace validation problem

Our problem can be stated as follows. Let $\mathcal{S}$ be the set of finite behaviors that satisfy the specification *Spec*, and let $\mathcal{T}$ be the set of finite behaviors represented by the trace, with arbitrary values assigned to variables whose values are not recorded. The trace is compatible with the specification if $\mathcal{S} \cap \mathcal{T} \neq \emptyset$. Note in particular that we do not check for refinement of the high-level specification by the trace, expressed as $\mathcal{T} \subseteq \mathcal{S}$: this condition would be too strong in the presence of non-deterministically chosen values for variables not recorded in the trace.

Figure 4 illustrates the idea. On the left-hand side, the chain of nodes represents the trace obtained from the instrumented program. The graph on the right-hand side represents the state space of the TLA$^+$ specification, and we must check if the trace can be matched to some path in the state space. The first node of the trace must correspond to some initial state of the graph: in our example, we assume that this is the case for the state labeled A in the lower left-hand corner. Then, we try to match at least one successor of an already matched state with the corresponding successor in the trace. There may be several matching states, in particular due to incomplete information about variable values: in the example, we assume that two successors of state A match the second node of the trace. On the other hand, the state labeled C in the left-hand column of the state space does not have a successor matching the node labeled D in the trace. Overall, the trace of the example is compatible since there is at least one path in the state space that matches the trace.

The problem is actually a little more subtle: atomic steps of the implementation need not match precisely those of the specification but may correspond to zero or several steps of the TLA$^+$ specification. We explain how we reduce the problem to one of constrained model checking and how we realize it using the TLC model checker. In general, the programmer is in charge of tracking the correspondence between the event names in the trace and the actions in the specification as well as the correspondence between the names of the logged variables and those from the specification. In the specific case presented in the

next section, these correspondences are one to one and could be automatically generated.

4.2 Realizing trace validation using the TLC model checker

The set $\mathcal{S}$ of finite behaviors satisfying the original specification is defined by the TLA$^+$ formula *Spec*. In order to characterize the intersection $\mathcal{S} \cap \mathcal{T}$ of finite behaviors that also correspond to the trace obtained from the execution, we add constraints to *Spec*. Concretely, our framework provides a module `TraceSpec` that provides operators for defining the constrained specification.

```
------------------- MODULE TraceSpec -------------------
EXTENDS Naturals, Sequences, TLC, Json, IOUtils
VARIABLE l
Trace ≜ ndJsonDeserialize(IOEnv.TRACE_PATH)
IsEvent(e) ≜
  ∧ l ∈ 1 .. Len(Trace) ∧ l' = l + 1
  ∧ "event" ∈ DOMAIN Trace[l] => Trace[l].event = e
  ∧ UpdateVariables(Trace[l])
```

The module declares a variable `l` that will denote the number of the current line of the trace. The definition of the operator `Trace` causes the JSON representation of the trace to be internalized as a sequence of records whose fields correspond to the entries of the log file. The operator `IsEvent(e)` encapsulates processing the current line of the trace and generating the constraints imposed by it. It requires that `l` is a valid index into the trace. The trace may explicitly indicate the event corresponding to the current transition by including an `"event"` field, in which case the operator checks for the expected value. (Any event parameters indicated by the entry are taken into account below.) The operator increments the variable `l` and computes new values for the variables recorded in the current line of the trace, by evaluating the operator `UpdateVariables`:

```
  UpdateVariables(ll) ≜
    ∧ "rmState" ∈ DOMAIN ll =>
          rmState' = UpdateVariable(rmState, "rmState", ll)
    ∧ ...  \* similar lines for variables tmState, tmPrepared, msgs
```

That operator is defined as a conjunction that checks for each variable of the original specification if a corresponding entry exists in the current line of the trace and, if so, determines the new value of the variable from that entry. The operator `UpdateVariable` is predefined in our framework and computes the new value from the value of the first argument (i.e., the unprimed variable) and the operator to be applied according to the trace. For example, the JSON entry
`"rmState": [ { "op": "Update", "path": ["rm-0"], "args": ["prepared"] } ]`
will give rise to the TLA$^+$ value

```
[rmState EXCEPT !["rm-0"] = "prepared"]
```

representing the function `rmState` with the value at argument `rmState["rm-0"]` replaced by `"prepared"`. A single update in the JSON trace may correspond to changes to several parts of a complex value such as a function or a record. Predefined TLA$^+$ operators exist for the different operators that our framework currently supports, and this can be smoothly extended, both in the instrumentation library and at the TLA$^+$ level, should additional operators be desirable.

For every action of the original specification, we then construct a similar action of the trace specification by conjoining the predicate `IsEvent`. For example, the action of the trace specification corresponding to the `TMCommit` action of the specification of the two-phase commit protocol is defined as

```
IsTMCommit ≜ IsEvent("TMCommit") ∧ TMCommit
```

Because TLC evaluates formulas from left to right, the effect of these definitions is to first update state variables based on the information in the log and then evaluate the action predicate of the underlying specification. This evaluation checks that the predicate is satisfied while non-deterministically generating suitable values for any variables left open in the trace. For actions that take arguments, we define the constrained action such that any parameters provided in the trace indicate the instance of the original action that may occur:

```
IsTMRcvPrepared ≜
  ∧ IsEvent("TMRcvPrepared")
  ∧ IF "event_args" ∈ DOMAIN Trace[l] ∧ Len(Trace[l].event_args) >= 1
    THEN TMRcvPrepared(Trace[l].event_args[1])
    ELSE ∃ r ∈ RM : TMRcvPrepared(r)
```

The overall next-state relation `TraceNext` is defined as the disjunction of these actions. Writing the trace specification for a given algorithm specification is systematic and could be automated in most cases, except when `TraceNext` may include disjuncts corresponding to action composition as discussed in Sect. 4.3.

TLC evaluates the action `TraceNext` from the current state in order to compute all possible successor states. Even if no such state can be found (as in the case of state C in the leftmost column of Fig. 4), there may still exist matching behaviors elsewhere in the state space. Therefore, we should not direct TLC to check for deadlocks of the constrained specification. It would also be inappropriate to verify the liveness property `◇(l >= Len(Trace))`, which would not hold for incomplete prefixes of the constrained state space. On the other hand, checking the invariant `□(l < Len(Trace))` will cause TLC to output a behavior of the original specification that matches the trace if such a behavior exists. However, no useful information will be provided when the trace cannot be matched, i.e., when trace validation fails. Instead, we may observe that the length of the longest path in the constrained state space corresponds to the diameter of the graph,[9] which suggests defining the predicate

```
TraceAccepted ≜ TLCGet("stats").diameter - 1 = Len(Trace)
```

[9] The presence of the line counter `l` excludes cycles in the constrained state space.

as the postcondition to check for determining success of trace validation. If the postcondition is violated, $\mathcal{S}$ does not contain any behavior whose prefix of appropriate length is in $\mathcal{T}$, i.e., matches the log of the execution. Because there is not a single behavior explaining this failure, TLC cannot generate a counter-example. However, it can generate a maximal finite behavior in $\mathcal{S}$ that corresponds to the log but cannot be extended further. The *hit-based breakpoint* feature of the TLA$^+$ debugger [17] can be used to halt state-space exploration when the diameter reported by TLC is attained, and the user can then step back and forth in the state space in order to understand the discrepancy.

Regardless of whether we use a property or a postcondition, trace validation might incorrectly accept a trace if the trace provides incomplete information. A particularly extreme case would be a trace that does not log any variable updates or action occurrences and only provides the length of a finite execution.

4.3 Analyzing discrepancies

We have used trace validation for several case studies: besides the two-phase commit protocol presented here, we experimented with a distributed key-value store ensuring snapshot isolation [7] whose specification was taken from the standard collection of TLA$^+$ specifications [19], the distributed termination detection algorithm EWD 998 [4,15] from the same collection, two existing implementations of the Raft consensus algorithm [22], and the Microsoft Confidential Consortium Framework [11]. In all cases, trace validation quickly identified executions of the distributed implementations that could not be matched to the high-level specification. In the following, we identify several reasons for such discrepancies.

Implementation shortcuts. The implementation of the Two-Phase Commit protocol could use a counter to store the number (rather than the identities) of the RMs from which a `"Prepared"` message was received, and it could check if all RMs are prepared to carry out the transaction by simply comparing the counter value to the number of resource managers. Such an implementation will work correctly as long as no messages are lost. However, if some RM resends the `"Prepared"` message due to a timeout, the TM might count it twice and then commit prematurely. We found this kind of implementation error to be reliably detected using trace validation.

Overly strict specification. The implementation [16] of the token-based distributed termination detection algorithm EWD 998 allowed a node to send an ordinary (non-token) message to itself, which was ruled out in the specification. Again, trace validation quickly discovered this mismatch, which can be resolved by adapting either the specification or the implementation.

Mismatch of the grain of atomicity. Mainstream programming languages do not provide atomic transactions encapsulating several updates, so the choice of when to log an action corresponding to the TLA$^+$ specification requires consideration. Transitions that modify shared state such as sending or receiving network messages, acquiring or releasing global locks or committing state to stable storage are natural candidates. In most cases, we found it not too difficult to identify

suitable points in the code for instrumentation, but the choices are important for trace validation to be meaningful.

It frequently happens that the implementation takes steps that are invisible at the level of the abstract state space. In our Two-Phase Commit running example, sent messages are (permanently) stored in a set, so resending a message due to a timeout conceptually corresponds to a stuttering step of the specification. However, when the implementation resends, say, a `"Prepared"` message, it should not log an `RMPrepare` step because such a step is not allowed to take place when the RM is already in state `"prepared"`. For this example, one can either choose to omit logging an action when a message is resent or only log the new (in fact unchanged) variable values but no action. The fact that TLA$^+$ specifications are insensitive to stuttering steps is helpful in such situations.

An implementation may also combine two or more separate steps of the TLA$^+$ specification into a single transition. For example, in Raft implementations, nodes may update their term upon receiving an AppendEntries request for a higher term, whereas the two actions of updating the term and appending entries are distinguished in the Raft specification. In such cases, one may instrument the implementation such that it logs two transitions in succession. Alternatively, one may add an explicit disjunct to the next-state relation of the TLA$^+$ specification used for trace validation, making use of the TLA$^+$ action composition operator $A \circ B$, support for which has recently been added to TLC. Doing so is a way of explicitly documenting optimizations of the implementation with respect to the specification.

4.4 Experience with trace validation

Overall, we and the engineers we worked with found it quite straightforward to instrument existing code, notably when the TLA$^+$ specification was used as a guideline for writing the implementation. The instrumentation library presented in Sect. 3 with corresponding TLA$^+$ operators shown in Sect. 4.2 was found helpful but not strictly necessary: in fact, it was not used in all of our case studies. Writing the trace specification is generally systematic and straightforward.

The most significant case study to which we applied trace validation occurred in the context of reverse engineering a formal specification for a Raft-inspired consensus algorithm implemented within the Confidential Consortium Framework (CCF). The starting point was a specification written after analyzing the source code, which was then corrected and amended through trace validation based on an existing test suite that exercises non-trivial system behavior. Model checking the resulting specification revealed serious violations of key safety properties. The counterexamples obtained in this way were translated into new tests, which confirmed previously unknown problems with the implementation. After addressing these issues at the specification level, corresponding updates to the implementation were made. To ensure ongoing consistency between the specification and its implementation, trace validation is now part of CCF's continuous integration pipeline. A detailed analysis of our experiences with formally verifying CCF, including trace validation, is discussed in a separate paper [12].

We also leveraged trace validation when implementing the distributed termination detection algorithm EWD 998, starting from its preexisting TLA$^+$ specification. The algorithm is based on a token-passing scheme for detecting when global termination has occurred; its implementation consists of about 500 lines of code. Its specification includes an action for atomically passing the token from one node to its neighbor. The implementation is based on asynchronous message passing, and sending and receiving the token was logged as two separate transitions. Moreover, the implementation sends the token as soon as local termination occurs, whereas the specification has a separate action for termination detection. Trace validation pointed out both discrepancies, which are instances of differences in the grain of atomicity. Once these discrepancies were fixed by having the implementation log both termination detection and token sending, but not token receiving, the issue mentioned previously of a node possibly sending an ordinary message to itself was detected. Finally, it was found that the nodes continued to pass the token even after all nodes except for the initiator had terminated, which corresponded to an implementation error. Since then, thousands of traces have been successfully validated, and we are now confident that the implementation is indeed correct.

4.5 Implementation aspects

Our approach to instrumentation is flexible with regard to the detail of information recorded in the trace: only a subset of variables needs to be included in the trace, and names and parameters of corresponding TLA$^+$ actions may or may not be given. Less information in the trace increases the potential degree of non-determinism in the trace specification and may lead to a combinatorial explosion during model checking. To some extent, this problem can be alleviated using different exploration strategies. TLC's default breadth-first search (BFS) ensures shortest-length counter-examples and is most informative for debugging. However, depth-first search (DFS) constrained by the length of the trace can be more efficient because checking can be stopped as soon as some behavior of the expected length has been found.

We report in Fig. 5 the numbers of distinct states explored with BFS/DFS by TLC for several valid traces that contain more or less information; a single figure indicates that BFS and DFS generate the same number of states. We consider traces for the Two-Phase Commit protocol for 4, 8, 12, and 16 RMs. For the Key-Value Store we consider 4, 8 or 12 agents accessing a store with a maximum of either 10 keys and 20 values or 20 keys and 40 values. The column headings indicate the kind of information that was recorded in the traces: all variables and the events with their arguments (VEA), just the variables (V), the variables and some events (VpEA), only the events with their arguments (EA) or only event names (E). For Two-Phase Commit, VpEA records only the events of the TM, for Key-Value Store, only the start and end of transactions are logged.

As expected, tracing full information for variables and the events requires exploring the least number of states. Logging only the variables or only the event

Instance	length	VEA	V	VpEA	EA	E
TP, 4 RMs	17	19	211/35	19	48/22	246/58
TP, 8 RMs	33	35	8k/73	35	640/42	22k/695
TP, 12 RMs	73	74	∞/209	74	11k/86	2.5M/27k
TP, 16 RMs	90	91	∞/270	91	205k/107	∞/557k
KV, 4a, 10k, 20v	109	111	∞/158	13k/149	111	∞/35k
KV, 8a, 10k, 20v	229	231	∞/317	18k/307	231	∞/176k
KV, 12a, 10k, 20v	295	297	∞/423	678k/411	297	∞/300k
KV, 4a, 20k, 40v	131	133	∞/298	∞/285	133	∞/9.9M
KV, 8a, 20k, 40v	249	251	∞/1164	∞/1146	251	∞
KV, 12a, 20k, 40v	308	310	∞/552	∞/538	310	∞

Fig. 5: Number of distinct states explored for valid traces of Two-Phase Commit (TP) and Key-Value Store (KV), for various degrees of precision (1hr timeout).

names quickly leads to state-space explosion that makes trace validation infeasible. However, recording well-chosen partial information is sufficient for limiting the state space. For the Key-Value Store, logging events and their arguments is enough because they uniquely determine the corresponding variable values. For Two-Phase Commit, logging the events of the RMs is unnecessary.

Besides an exponential growth of the number of states, too imprecise logs may even lead to erroneous traces being accepted because the model checker may be able to infer suitable values that do not correspond to the actual ones. Nevertheless, such traces, which require little instrumentation, can still be useful at early stages of validation for finding issues. For instance, validating a trace containing only the event names for 16 RMs takes a considerable amount of time but the bug concerning the implementation using counters mentioned in Sect. 4.3 can still be detected with such a trace in less than a minute.

5 Related Work

The verification of execution traces against high-level properties or specifications has a long history in formal methods. Havelund [10] introduced runtime verification as a lightweight method for checking that a system's execution trace conforms to its high-level specification. Runtime verification typically involves the generation of a monitor from the specification, which consumes the trace events to check conformance [6]. Howard et al. [13] verified execution traces directly against the system's high-level specification. Their work also demonstrated that it was feasible to use standard model checkers (ProB and Spin) to check execution traces against these specifications, a technique that we also employ. However, they did not consider distributed programs that require the use of (centralized or distributed) clocks for preserving causality, and they did not consider traces with incomplete information.

Tasiran et al. [25] were the first to extract and validate traces obtained from a hardware simulator against a TLA$^+$ spec, demonstrating the practical appli-

cability of trace validation. The adoption of TLA$^+$ among distributed system practitioners, spurred by Newcombe et al. [20], and the formalization of trace validation as a refinement check by Pressler [23], caused trace validation to be applied to real-world distributed systems. For instance, Davis et al. [3] applied the technique to MongoDB, discovering a non-trivial implementation bug. However, they faced challenges in consistently logging the implementation state, and aligning different grains of atomicity, which we attribute to them not leveraging TLA$^+$'s non-determinism to infer implementation state, and action composition to align atomicity. Niu et al. [21] also validated traces of Zookeeper, ensuring that its implementation corresponds to its spec. Similarly, Wang et al. [28] revealed several implementation bugs by replaying TLA$^+$ behaviors against instrumented implementations. Furthermore, the work by Wang et al. serves as an example of the challenges of aligning the grains of atomicity, illustrated by the authors asserting two bugs in a widely used and well-established specification [22]. We contend that these are, in fact, common TLA$^+$ modeling patterns and can be handled with action composition. Nevertheless, all efforts found non-trivial bugs in real-world systems by comparing implementation traces to high-level TLA$^+$ behaviors, a testament to the effectiveness of this approach.

To facilitate a closer alignment between high-level specifications and their actual implementations, Hackett et al. [9] and Foo et al. [8] proposed extensions to PlusCal, an algorithmic language whose translator serves as a front-end to TLA$^+$. These aim at adding sufficient detail to specifications for code generation and enabling the generation of runtime monitors, respectively. Despite these advancements, the requirement for implementation-level shims may impede widespread industry adoption. Moreover, the projection from a global state machine, which is a common modeling pattern, to node-local state machines prevent the verification of global properties by monitors. Yet, both approaches still translate specifications written in their PlusCal extensions into TLA$^+$, allowing users to leverage all of the existing verification tools.

Notions of testing implementations against formal specifications, such as input-output conformance [26] are related to work on trace validation in that they also help establishing confidence in the correctness of implementations. A main difference is that in these approaches, implementations are considered black boxes that admit certain observations at the interfaces, whereas we assume having access to the implementation code. A similar remark holds for techniques of active learning of state machines [27]. Although we only have anecdotal evidence, we observed that programmers find it quite easy to instrument their code, and that they will apply the necessary engineering judgment to overcome mismatches between the grains of atomicity of the specification and the implementation.

6 Conclusion

Formal verification techniques are known to be most effective for specifications written at high levels of abstraction where the size of state spaces (for model checking) and the complexity of invariants (for deductive verification) are man-

ageable. High-level specifications can serve as guidelines for programmers when implementing an algorithm, and in some cases it may even be possible to generate code from a sufficiently detailed specification. Model-based testing is a collection of techniques that rely on formal specifications for generating test cases, aiming at coverage guarantees or at exploring parts of the state space deemed interesting based on an analysis of the specification.

In this paper we described an approach for relating traces of the executions of a distributed program to the state machine described by a high-level specification written in TLA$^+$. Our purpose with this approach is to identify discrepancies that can be analyzed using the TLC model checker in order to determine if they are due to an error in the implementation, a restrictive specification, or an artefact due to a mismatch in the grains of atomicity. Although the technique does not provide formal correctness guarantees, mainly due to the necessary decisions on when to log an event corresponding to an atomic action, we have found it to be surprisingly effective for finding serious bugs in implementations that had previously been validated using traditional quality assurance techniques.

We are certainly not the first to suggest that trace validation can be worthwhile for relating high-level specifications and programs. Original aspects, to the best of our knowledge, are our ability to handle different grains of atomicity, and that we do not require all specification variables or events to be traced in the log; instead, we use a model checker to reconstruct missing information. This leads to a tradeoff between the amount of detail included in the trace, the increase of the search space for model checking, and the reliability of the verdict. Our experiments suggest that it is enough to trace a suitably chosen subset of variables and/or events. We implemented a library of Java methods and TLA$^+$ operators to support collecting traces but their use is not strictly essential for applying our technique. In fact, the EWD 998 and the CCF case studies used ad-hoc code instrumentations for generating the logs.

We contribute industry-grade support for trace validation by implementing action composition, depth-first search, and debugging of trace validation in the TLC model checker and the TLA$^+$ debugger. This support enabled not only the successful adoption of trace validation by the previously mentioned CCF project but also by the etcd project [5], a widely-used distributed key-value store. Both projects verify that their implementations in C++ and Go, respectively, adhere to their TLA$^+$ specifications. We have exclusively used the explicit-state model checker TLC in our experiments. In principle, symbolic model checkers such as TLA$^+$'s Apalache tool [14] could also be used, but we suspect that the overhead of generating and evaluating constraints would be prohibitive in comparison to explicit-state model checking, in particular when the degree of non-determinism is low, as is the case when recording sufficiently informative traces.

At the moment, we exploit random variations in implementation parameters such as message delays or failures in order to generate meaningful traces of the program to be analyzed. In future work, we intend to use the TLA$^+$ specification in order to be able to steer executions towards "interesting" parts of the state space. We also intend to study the feasibility of applying trace validation online

during the execution of the program or even of using the technique as a run-time monitor to block unsafe transitions of the implementation.

References

1. H. Cirstea, M. A. Kuppe, B. Loillier, and S. Merz. Trace validation for TLA$^+$. *arXiv:xx.yy [cs.DC]*, 2024.
2. D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA$^+$ Proofs. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods*, volume 7436 of *LNCS*, pages 147–154, Paris, France, 2012. Springer.
3. A. J. J. Davis, M. Hirschhorn, and J. Schvimer. eXtreme Modelling in Practice. *Proceedings of the VLDB Endowment*, 13(9):1346–1358, May 2020.
4. E. W. Dijkstra. EWD 998: Shmuel Safra's Version of Termination Detection. `http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF`, Jan. 1987.
5. etcd project. TLA+ Specification and Trace Validation for Raft Library: A Brief Guide. `https://github.com/etcd-io/raft/tree/main/tla`, 2024.
6. Y. Falcone, K. Havelund, and G. Reger. A Tutorial on Runtime Verification. *Engineering Dependable Software Systems*, 34:141–175, 01 2013.
7. A. Fekete. Snapshot isolation. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 2659–2664. Springer, 2009.
8. D. Foo, A. Costea, and W.-N. Chin. Protocol Conformance with Choreographic PlusCal. In C. David and M. Sun, editors, *Theoretical Aspects of Software Engineering*, volume 13931, pages 126–145. 2023.
9. F. Hackett, S. Hosseini, R. Costa, M. Do, and I. Beschastnikh. Compiling Distributed System Models with PGo. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 159–175, Vancouver BC Canada, Jan. 2023. ACM.
10. K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In G. Goos, J. Hartmanis, J. Van Leeuwen, K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885, pages 245–264. Springer, 2000.
11. H. Howard, F. Alder, E. Ashton, A. Chamayou, S. Clebsch, M. Costa, A. Delignat-Lavaud, C. Fournet, A. Jeffery, M. Kerner, F. Kounelis, M. A. Kuppe, J. Maffre, M. Russinovich, and C. M. Wintersteiger. Confidential Consortium Framework: Secure Multiparty Applications with Confidentiality, Integrity, and High Availability. *Proceedings of the VLDB Endowment*, 17(2):225–240, Oct. 2023.
12. H. Howard, M. A. Kuppe, E. Ashton, A. Chamayou, and N. Crooks. Smart Casual Verification of CCF's Distributed Consensus and Consistency Protocols. *arXiv:2406.17455 [cs.DC]*, 2024.
13. Y. Howard, S. Gruner, A. Gravell, C. Ferreira, and J. C. Augusto. Model-Based Trace-Checking. *arXiv:1111.2825 [cs]*, Nov. 2011.
14. I. Konnov, J. Kukovec, and T.-H. Tran. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.
15. I. Konnov, M. Kuppe, and S. Merz. Specification and Verification with the TLA$^+$ Trifecta: TLC, Apalache, and TLAPS. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*, volume 13701, pages 88–105. Springer, 2022.
16. M. A. Kuppe. Implementing a TLA$^+$ Specification: EWD998Chan. `https://github.com/tlaplus/Examples/pull/75`, Apr. 2023.

17. M. A. Kuppe. The TLA$^+$ Debugger. In P. Masci, C. Bernardeschi, P. Graziani, M. Koddenbrock, and M. Palmieri, editors, *Software Engineering and Formal Methods. SEFM 2022 Co-located Workshops*, volume 13765, pages 174–180. Springer, 2023.
18. L. Lamport. *Specifying Systems.* Addison-Wesley, Boston, Mass., 2002.
19. L. Lamport, M. A. Kuppe, S. Merz, A. Helwer, W. Schultz, J. Hemphill, M. Ryndzionek, I. Konnov, T. H. Tran, J. Widder, J. Gray, M. Demirbas, G. Hu, G. Losa, R. Pressler, Y. Akhouayri, L. Dong, Z. Niu, L. N. X. Terry, G. Gandhi, I. DeFrain, M. Harrison, S. Raju, C. G. Mathew, F. Andriani, and L. Yvoz. TLA$^+$ Examples. `https://github.com/tlaplus/examples/`.
20. C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, Mar. 2015.
21. Z. Niu, L. Dong, Y. Zhu, and L. Chen. Verifying Zookeeper Based on Model-Based Runtime Trace-Checking Using TLA$^+$. In *Proceedings of the 7th International Conference on Cyber Security and Information Engineering*, pages 13–18, Brisbane QLD Australia, Sept. 2022. ACM.
22. D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.
23. R. Pressler. Conjunction Capers: A TLA$^+$ Truffle. `https://conf.tlapl.us/2020/`, Sept. 2020.
24. W. Schultz, S. Zhou, I. Dardik, and S. Tripakis. Design and Analysis of a Logless Dynamic Reconfiguration Protocol. In Q. Bramas, V. Gramoli, and A. Milani, editors, *25th Intl. Conf. Principles of Distributed Systems (OPODIS 2021)*, volume 217 of *LIPIcs*, pages 26:1–26:16, Strasbourg, France, 2021. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
25. S. Tasiran, Y. Yu, B. Batson, and S. Kreider. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *In Proceedings of the 3rd IEEE Workshop on Microprocessor Test and Verification, Common Challenges and Solutions*, 2002.
26. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Softw. Concepts Tools*, 17(3):103–120, 1996.
27. F. W. Vaandrager. Model Learning. *Commun. ACM*, 60(2):86–95, 2017.
28. D. Wang, W. Dou, Y. Gao, C. Wu, J. Wei, and T. Huang. Model Checking Guided Testing for Distributed Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 127–143, Rome Italy, May 2023. ACM.
29. Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA$^+$ Specifications. In L. Pierre and T. Kropf, editors, *10th IFIP WG 10.5 Conf. Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer.

A Specification of the Two-Phase Commit protocol

Fig. 6 contains the TLA$^+$ specification of the well-known Two-Phase Commit. In this version of the protocol, aborting the transaction is left to the TM.

B Implementation of the Two-Phase Commit protocol

A possible Java implementation of the transaction manager and of the resource managers is presented in Fig. 7. Only a simplified version of the main method is presented, the auxiliary methods are faithful Java translations of the actions in the TLA$^+$ specification.

A TM is identified by a `name` and uses a `network` (manager) to send and receive messages. It stores the collection of `resourceManagers` that it manages as well as the collection of `preparedRMs` that have already indicated their availability (empty at the beginning). The TM continuously reads messages and when the message corresponds to a prepared RM, the respective manager is added to `preparedRMs` (in the method `handleMessage`). The `receive` is blocking unless a timeout is reached. When all RMs announced to be prepared, i.e. `resourceManagers` and `preparedRMs` contain the same elements (checked in method `checkAllPrepared`), the TM sends a message to each managed resource manager (from `resourceManagers`) to inform them that the transaction has been committed (method `commit`). The TM can decide to abort, for example because there are still some RMs who have not announced to be prepared before some deadline, and in this case the TM informs all the RMs that the transaction should be aborted (method `abort`).

C Detailed description of the instrumentation library

The instrumentation library provides primitives that allow each component of the implementation to log events reflecting its behaviour. The main class of the library, TLATracer, provides the methods for logging events and state variable updates:

```
1  static TLATracer getTracer(String tracePath, Clock clock);
2  static TLATracer getTracer(String tracePath);
3
4  void notifyChange(String var, List<String> path,
5                    String operator, List<Object> args);
6  VirtualField getVariableTracer(String var);
7
8  long log(String eventName, Object[] args, long clockValue);
9  long log(String eventName, Object[] args);
10 long log(String eventName)
11 long log();
```

The method `getTracer` creates a tracer that logs events into a file specified by the `tracePath` parameter. Each tracer records the time of each event using a shared `clock`, ensuring that events are recorded in chronological order both locally (within individual components) and globally (across all components). While each component uses a unique `tracePath` for its tracer, all tracers synchronize their timing using the same type of clock. The library offers various types of clocks suitable for different scenarios: an in-memory clock is used when components are threads within the same process; a file-based clock is used for processes on the same machine; and a server-based clock is appropriate for distributed components. When a distributed clock is handled explicitly by the components (e.g. a Lamport or a vector clock) there is no need to use a centralized clock; for this case the library proposes the `getTracer` method with only one parameter.

Updates to variables are tracked by the method `notifyChange`, which records operations that have been applied to a given variable. The parameter `var` refers to a variable from the TLA$^+$ specification but reflects the operations executed at the implementation level and thus, `notifyChange` implicitly links the variables from the implementation to the ones in the specification. Our library supports standard `operators` such as updating the variable by a new value, adding or removing a value to or from a set or bag (multi-set), overriding the value of individual fields (identified using the `path` argument) of functions or records, etc. The path to the field of the variable the operator applies on is specified as the list of field names leading to it; for example, `["address","city"]` to specify the city of residence of a person having a field `address` which, in turn, has a field `city`. The list of arguments for the respective operator is specified with `args`.

For example, the update of the control state of an RM occurring in the method `sendPrepared` could be recorded as follows (for simplicity, in what follows, we write `{...}` instead of `List.of(...)` and `new Object[]{...}`):

```
1  state = ResourceManagerState.PREPARED;
2  tracer.notifyChange("rmState", {name}, "Update", {"prepared"});
```

In the TLA$^+$ specification, the RM states are modeled using the `rmState` function and thus, for an RM named *rm-0* the above statement tracks the fact that `rmState["rm-0"]` is set to the new value `"prepared"`; the subsequent entry in the final trace file would contain an element of the form:

```
{ "rmState": [ { "op": "Update", "path": ["rm-0"], "args": ["prepared"] } ]
```

The `log` method is used to produce one log entry in the trace file that reflects all the variable changes recorded with `notifyChange` since the last call to `log` (or since the start of the process if `log` was never called before). In its general form, the `log` method records the variable changes as well as the event name and its parameters, provided as arguments of the `log` method. Variants of the `log` method ignore the event or its parameters. The time when the log has been performed is used as a timestamp for the corresponding entry; this value is returned by the method. The `clockValue` is only relevant if the (distributed) clock is managed explicitly by the involved processes. In our example, since a

centralized clock is used, the clock value doesn't need to be specified explicitly and one of the other three versions of the `log` method can be used.

For example, when a commit message is received by an RM, the state change is recorded using `notifyChanges` and the `log` statement indicates that this corresponds in the specification to the action `RMRcvCommitMsg` for the corresponding RM:

```
void handleMessage(Message message) {
  if (message.getContent().equals("Commit")) {
    state = ResourceManagerState.COMMITTED;
    tracer.notifyChange("rmState", {name}, "Update", {"committed"});
    tracer.log("RMRcvCommitMsg", {name});
  } ...
}
```

The trace entry obtained in the final trace file when executing the above code for the RM named *rm-0* has the form

```
{ "clock": 4,
  "rmState": [ { "op": "Update", "path": ["rm-0"], "args": ["committed"] } ],
  "event":"RMRcvCommitMsg",
  "event_args":["rm-0"]  }
```

The other variants of the method `log` ignore the arguments of the event or even the event. For example, if in the method `handleMessage` we used a simple `log` without arguments then, the corresponding entry in the trace wouldn't contain the lines `"event"` and `"event_args"`. As discussed in Section 4.5, more detailed information traced by the implementation can lead a more efficient trace validation.

In general, the same variable is changed several times during the execution and each change is recorded using the relatively verbose `notifyChange` method. The library proposes a `VirtualField` which can be used to trace the changes of a variable, or of a field in a variable, in a more compact way; such a variable tracer can be obtained using the method `getVariableTracer` from `TLATracer`. To trace a specific field of the variable, we can use the method `getField` on the obtained `VirtualField`.

For example, we can obtain the tracers for the TLA$^+$ variables `msgs` and `rmState`:

```
VirtualField traceMessages = tracer.getVariableTracer("msgs");
VirtualField traceStateRMs = tracer.getVariableTracer("rmState");
```

and then the tracer for the field corresponding to *rm-0* in `rmState`:

```
VirtualField traceState = traceStateRMs.getField("rm-0");
```

A `VirtualField` can then be used to register a given operation on the respective variable, potentially using some arguments:

```
 void apply(String op, Object... args){...}
```

Several shortcut methods for the operators mentioned previously are also provided: `update`, `add`, `remove`, `clear`, ...

For example, we can use the variable tracer defined above and replace the line 4 in the method `handleMessage` either with

```
1  traceState.apply("Update", "committed");
```

or with

```
1  traceState.update("committed");
```

As shown in Section 3, several variable changes can be recorded successively and all these updates are combined into an entry through a `log` call. We could have used the `VirtualField`s in the `sendPrepared` method and use a simpler `log` call (for simplicity, in what follows, we write `k:v` instead of `Map.of(k,v)`):

```
1 void sendPrepared()  {
2   state = ResourceManagerState.PREPARED;
3   traceState.update("prepared");
4   traceMessages.add({"type":"Prepared", "rm":name});
5   tracer.log("RMPrepare");
6   networkManager.send(new Message(name, tmName, "Prepared", 0));
7 }
```

When executing the process for the RM named *rm-0*, the tracing instructions lead to same line in the trace file as shown in Section 3 except for the `"event_args"` part:

```
{ "clock": 4,
  "rmState": [ { "op": "Update", "path": ["rm-0"], "args": ["prepared"] } ],
  "msgs": [ { "op": "Add", "path": [],
              "args": ["type":"Prepared","rm":"rm-0"] } ],
  "event": "RMPrepare" }
```

D JSON Schema for a trace entry

An entry in the JSON file consists in the timestamp of the log and in zero or several variable updates, each of which is identified by a key having the name of the respective variable. Each variable update is an array of objects, each of its elements consisting of the name of the update operation, the array of fields indicating the path to reach the targeted field the update is applied on, and the potentially empty array of arguments for the update; an empty array of fields indicates that the whole variable is updated. The entry can also specify the `"event"` concerned by the update and the arguments (`"event_args"`) of the corresponding action.

Fig. 8 presents the JSON Schema for an entry in the trace file. The field `additionalProperties` stands for names of variables logged in the entry.

E Debugging violations of *TraceAccepted*

Section 4.2 introduced *TraceAccepted*, corrresponding to $\mathcal{S} \cap \mathcal{T} = \emptyset$, as the verification condition. Contrary to ordinary model checking, a violation of the condition *TraceAccepted* does not produce a counterexample. However, provided that $\mathcal{T} \neq \emptyset$, the behaviors within $\mathcal{T}$ help explain why an implementation trace fails to conform to the specification. In instances of zero non-determinism in the constrained specification, where $|\mathcal{T}| = 1$, it is generally advisable to examine the final state of the behavior and the corresponding line in the trace to identify the source of the mismatch. For more complex specifications involving multiple variables and actions, the *hit-based breakpoint* feature of the TLA$^+$ debugger can be used to halt state-space exploration when the diameter reported by TLC matches the value of a reported violation. Once halted, the debugger allows the user to step back and forth through the evaluation of action formulas to pinpoint the discrepancy. Additionally, the TLA$^+$ debugger displays the values of variables at both the current and successor states, facilitating a comparison with the trace values. With $|\mathcal{T}| > 1$, we provide a new *unsatisfied breakpoint* that activates for each state in $\mathcal{T}$ that is found to be unreachable. Furthermore, $\mathcal{T}$ can be visualized as a graph that not only includes all unreachable states but also references the subformula responsible for the state being unreachable.

In figure 9, the TLA$^+$ debugger reached the *unsatisfied breakpoint* that was triggered when the conjunct on line 133 evaluated to false (1). The conjunct evaluated to false because the (JSON) trace indicated a `"RMRcvCommitMsg"` action ($2a$) that changed the state of resource manager `"rm-1"` to `"prepared"` ($2b$). However, the Two-Phase specification on line 133 defined the `"RMRcvCommitMsg"` action to change the state of the resource manager `"r"` to `"committed"`. The view on the right shows the partial state graph up to the unreachable state, and the indicator that the formula on line 133 evaluated to false (3). The "Status" view at the bottom displays the previous TLC run, which reported the violation of *TraceAccepted* at a diameter of 16 (4).

```
CONSTANT RM
VARIABLES rmState, tmState, tmPrepared, msgs
vars ≜ ⟨rmState, tmState, tmPrepared, msgs⟩
TypeOK ≜ rmState ∈ [RM -> {"working","prepared","committed","aborted"}]
Messages ≜ [type : {"Prepared"}, rm : RM] ∪ [type : {"Commit","Abort"}]
TPInit ≜
   ∧ rmState = [r ∈ RM ↦ "working"]
   ∧ tmState = "init"
   ∧ tmPrepared = {}
   ∧ msgs = {}
RMPrepare(r) ≜
   ∧ UNCHANGED ⟨tmState, tmPrepared⟩
   ∧ rmState[r] = "working"
   ∧ rmState' = [rmState EXCEPT ![r] = "prepared"]
   ∧ msgs' = msgs ∪ {[type ↦ "Prepared", rm ↦ r]}
RMRcvCommitMsg(r) ≜
   ∧ UNCHANGED ⟨tmState, tmPrepared, msgs⟩
   ∧ [type ↦ "Commit"] ∈ msgs
   ∧ rmState' = [rmState EXCEPT ![r] = "committed"]
RMRcvAbortMsg(r) ≜
   ∧ UNCHANGED ⟨tmState, tmPrepared, msgs⟩
   ∧ [type ↦ "Abort"] ∈ msgs
   ∧ rmState' = [rmState EXCEPT ![r] = "aborted"]
TMRcvPrepared(r) ≜
   ∧ UNCHANGED ⟨rmState,tmState,msgs⟩
   ∧ tmPrepared' = tmPrepared ∪ {r}
   ∧ tmState = "init"
   ∧ [type ↦ "Prepared", rm ↦ r] ∈ msgs
TMCommit ≜
   ∧ UNCHANGED ⟨rmState, tmPrepared⟩
   ∧ tmState = "init"
   ∧ tmPrepared = RM
   ∧ tmState' = "done"
   ∧ msgs' = msgs ∪ {[type ↦ "Commit"]}
TMAbort ≜
   ∧ UNCHANGED ⟨rmState, tmPrepared⟩
   ∧ tmState = "init" ∧ tmState' = "done"
   ∧ msgs' = msgs ∪ {[type ↦ "Abort"]}
TPNext ≜
   ∨ TMCommit ∨ TMAbort
   ∨ ∃ r ∈ RM : ∨ RMPrepare(r) ∨ TMRcvPrepared(r)
                ∨ RMRcvCommitMsg(r) ∨ RMRcvAbortMsg(r)
Spec ≜ TPInit ∧ □[TPNext]_vars
Consistent ≜
   ∀ r1,r2 ∈ RM: ¬(rmState[r1] = "aborted" ∧ rmState[r2] = "committed")
THEOREM Spec => □(TypeOK ∧ Consistent)
```

Fig. 6: TLA$^+$ Specification of Two-Phase Commit.

```
public class TransactionManager {
  String name;
  Collection<String> resourceManagers, preparedRMs;
  NetworkManager network;
  public void run() throws IOException {
     while (true) {
        try {
           Message message = network.receive(name, RECEIVE_TIMEOUT);
           handleMessage(message);
        } catch (TimeOutException e) {}
        if (checkAllPrepared()) {
           commit();
           return;
        } else if (shouldAbort()) {
           abort();
           return;
} } }   }

public class ResourceManager {
  String name, tmName;
  ResourceManagerState state;
  NetworkManager network;
  public void run() throws IOException {
     working();
     while (true) {
        sendPrepared();
        try {
           Message message = network.receive(name, RECEIVE_TIMEOUT);
           handleMessage(message);
           return;
        } catch (TimeOutException e) {}
} } }
```

Fig. 7: Java implementation of the managers of the Two-Phase Commit protocol.

```
{
  "title": "TraceEntry",
  "description": "An␣entry␣in␣a␣trace",
  "type": "object",
  "properties": {
    "clock": {
      "description": "The␣timestamp␣of␣the␣event",
      "type": "integer",
      "minimum": 0
    },
    "additionalProperties": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "op": { "type": "string" },
          "path": { "type": "array" },
          "args": { "type": "array" }
        },
        "required": [ "op", "path", "args"]
      },
      "minItems": 1
    },
    "event": {
      "description": "Name␣of␣the␣event",
      "type": "string"
    },
    "event_args": {
      "type": "array",
      "items": { "type": "string" }
    }

  },
  "required": [ "clock"]
}
```

Fig. 8: JSON Schema for an entry in the trace file.

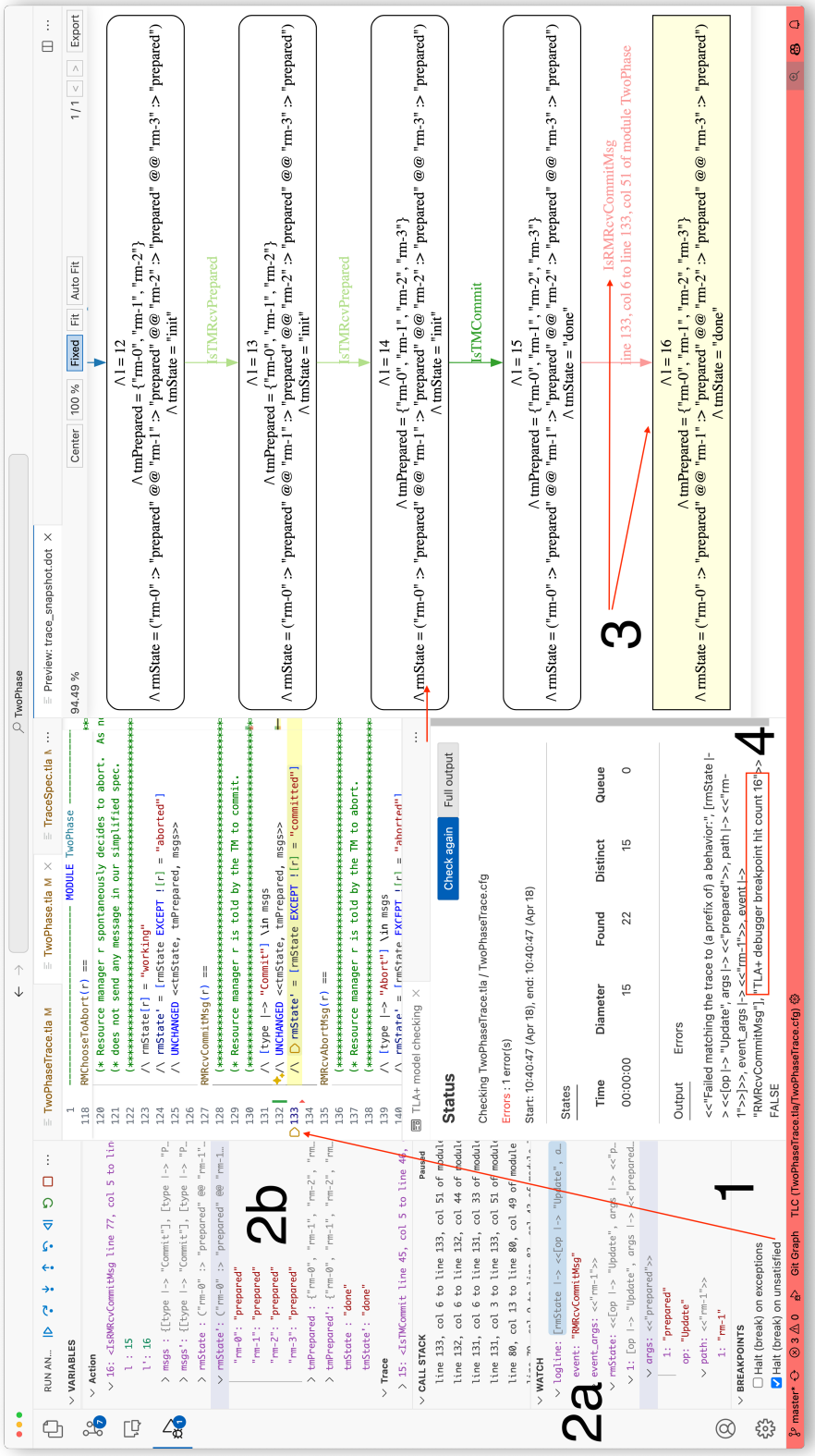

Fig. 9: A screenshot of the TLA+ debugger halted at the *unsatisfied breakpoint*.