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

Stephan Merz

(joint work with Horatiu Cirstea, Markus Kuppe, Benjamin Loillier)

Univ. of Lorraine, CNRS, Inria, LORIA, Nancy, France

IFIP Working Group 2.2
Aachen, September 2025

Motivation

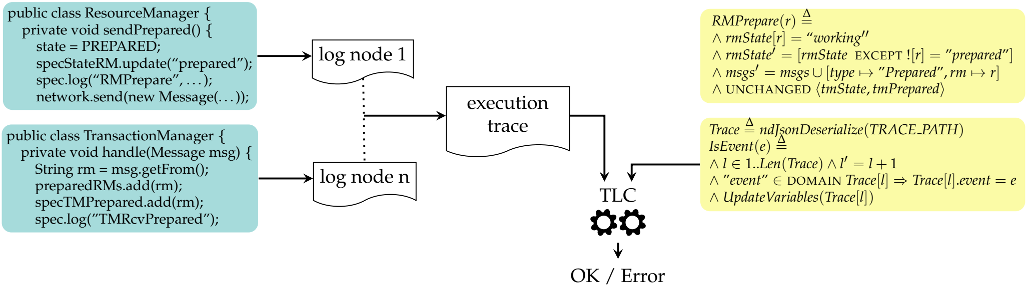
- TLA$^+$ specifications: mathematics for describing state machines
  - data structures and operations represented in set theory
  - next-state relation written as the disjunction of atomic actions
  - temporal logic for expressing fairness and liveness hypotheses
- Verification support
  - TLC — explicit-state model checker
  - Apalache — bounded SMT-based model checker
  - TLAPS — interactive proof system

Motivation

- TLA$^+$ specifications: mathematics for describing state machines
  - data structures and operations represented in set theory
  - next-state relation written as the disjunction of atomic actions
  - temporal logic for expressing fairness and liveness hypotheses
- Verification support
  - TLC explicit-state model checker
  - Apalache bounded SMT-based model checker
  - TLAPS interactive proof system
- Relate TLA$^+$ specifications and distributed programs
  - significantly different level of detail and grain of atomicity
  - formal refinement proofs are tedious, if possible at all

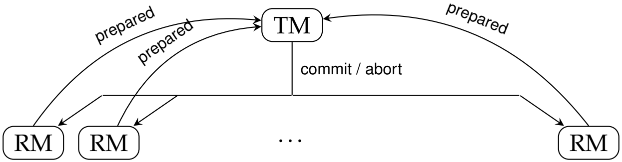
Trace Validation in a Nutshell

- Lightweight approach for finding bugs
  - instrument (Java) code to record transitions at specification level
  - obtain traces of runs and check if they correspond to some allowed behavior

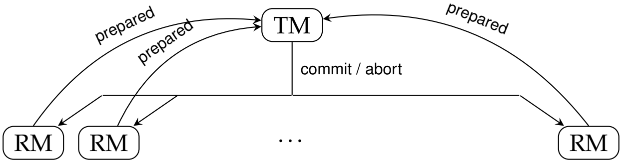


- Assumption: implementation and TLA+ specification are aligned

Running Example: Two-Phase Commit Protocol



Running Example: Two-Phase Commit Protocol



- TLA$^+$ definitions of two transitions of the transition manager

handle "prepared" message from RM r

$$
\begin{aligned}
&TMRcvPrepared(r) \triangleq \\
&\quad \land\ tmState = \text{"init"} \\
&\quad \land\ [type \mapsto \text{"prepared"}, rm \mapsto r] \in msgs \\
&\quad \land\ tmPrepared' = tmPrepared \cup \{r\} \\
&\quad \land\ \text{UNCHANGED}\ \langle tmState, rmState, msgs \rangle
\end{aligned}
$$

send "commit" order to all RMs

$$
\begin{aligned}
&TMCommit \triangleq \\
&\quad \land\ tmState = \text{"init"} \\
&\quad \land\ tmPrepared = RMs \\
&\quad \land\ tmState' = \text{"done"} \\
&\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{"commit"}]\} \\
&\quad \land\ \text{UNCHANGED}\ rmState
\end{aligned}
$$

- Specification of overall transition system, no processes or communication primitives

Excerpts from the Java Implementation of the Protocol

Two methods in class `TransactionManager`

```
protected void receive(Message msg) {
  if (msg.getContent().equals(TwoPhaseMessage.Prepared)) {
    preparedRMs ++;       // implementation counts "prepared" messages


  }
}

private void commit() throws IOException {    // assumes preparedRMs == resourceManagers.size()
  for (String rm : resourceManagers) {
    networkManager.send(new Message(getName(), rm, TwoPhaseMessage.Commit));
  }


}
```

Excerpts from the Java Implementation of the Protocol

Two methods in class `TransactionManager` instrumented for tracing

```
protected void receive(Message msg) {
  if (msg.getContent().equals(TwoPhaseMessage.Prepared)) {
    preparedRMs ++;       // implementation counts “prepared” messages
    spec.notifyChange(“tmPrepared”, “AddElement”, msg.getFrom());      // record variable update
    spec.log(“TMRcvPrepared”, msg.getFrom());      // log action occurrence
  }
}

private void commit() throws IOException {    // assumes preparedRMs == resourceManagers.size()
  for (String rm : resourceManagers) {
    networkManager.send(new Message(getName(), rm, TwoPhaseMessage.Commit));
  }
  spec.notifyChange(“messages”, “AddElement”, “commit”);
  spec.log(“TMCommit”);
}
```

Framework for Tracing Java Implementations

- Record transitions corresponding to TLA$^+$ actions
  - notifyChange: collects updates of (some) specification variables
  - instrumentation computes and records specification values
  - log: assembles updates, adds time stamp, and optionally records action

Framework for Tracing Java Implementations

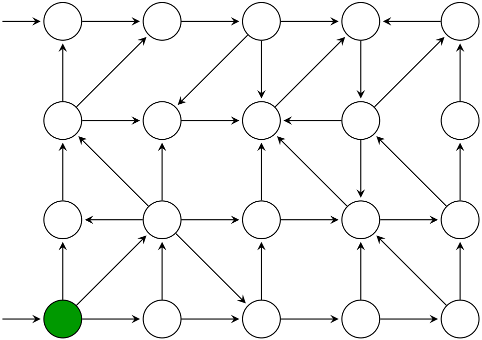
- Record transitions corresponding to TLA$^+$ actions
  - notifyChange: collects updates of (some) specification variables
  - instrumentation computes and records specification values
  - log: assembles updates, adds time stamp, and optionally records action
- Class `TLATracer` facilitates the instrumentation
  - convenience methods for recording updates of data structures

    ```
    specTMPrepared.add(msg.getFrom());
    ```
  - support for shared (physical) and logical clocks
  - output log as sequence of JSON entries
- Scripts for merging traces of individual nodes, sorted by timestamps

The Trace Validation Problem

Trace of implementation

State space of TLA$^+$ specification (fragment)

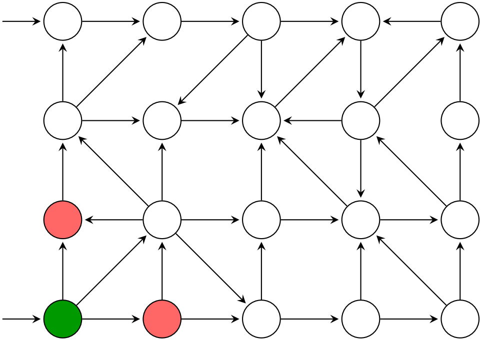


Does the trace correspond to some execution allowed by the TLA$^+$ specification?

The Trace Validation Problem

Trace of implementation

State space of TLA$^+$ specification (fragment)

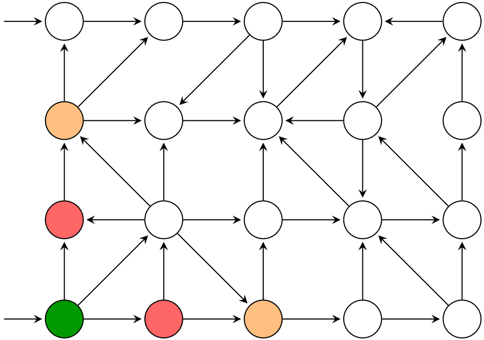


Does the trace correspond to some execution allowed by the TLA$^+$ specification?

The Trace Validation Problem

Trace of implementation

State space of TLA$^+$ specification (fragment)

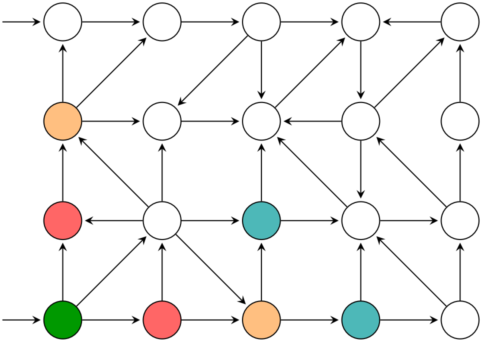


Does the trace correspond to some execution allowed by the TLA$^+$ specification?

The Trace Validation Problem

Trace of implementation

State space of TLA$^+$ specification (fragment)

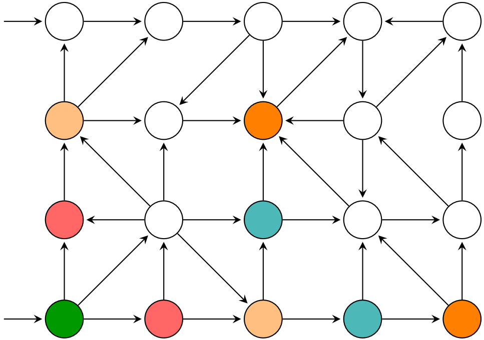


Does the trace correspond to some execution allowed by the TLA$^+$ specification?

The Trace Validation Problem

Trace of implementation

State space of TLA$^+$ specification (fragment)

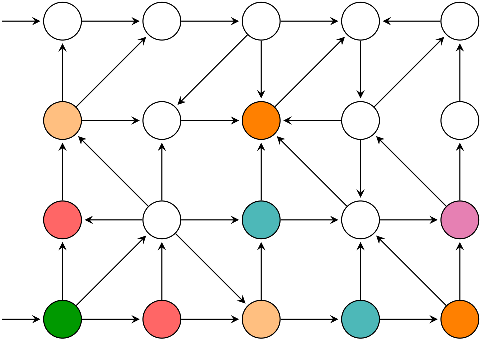


Does the trace correspond to some execution allowed by the TLA$^+$ specification?

The Trace Validation Problem

Trace of implementation

State space of TLA$^+$ specification (fragment)

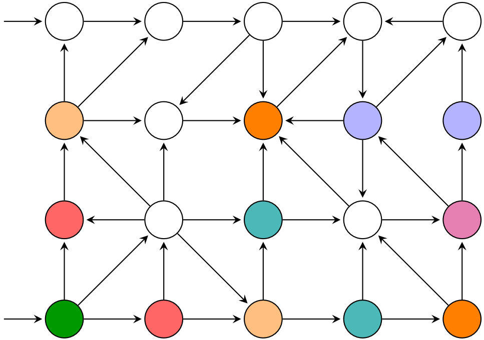


Does the trace correspond to some execution allowed by the TLA$^+$ specification?

The Trace Validation Problem

Trace of implementation

State space of TLA$^+$ specification (fragment)



Does the trace correspond to some execution allowed by the TLA$^+$ specification?

Reduce the question to a model checking problem, using the trace as a constraint

Setting up Trace Validation for the Model Checker

- Process the trace one line at a time
  - retrieve the trace as a TLA$^+$ sequence of records
  - use a line counter to track progress of validation
  - *IsEvent* operator reads one entry, checks for expected event, and updates state variables

MODULE *TraceSpec*

EXTENDS *TLC*, *Integers*, *Sequences*, *Json*, *IOUtils*

$Trace \triangleq ndJsonDeserialize(IOEnv.TRACE\_PATH)$

VARIABLE l * *current line in trace*

$$
\begin{aligned}
IsEvent(e) \triangleq\ & \land\ l \in 1 \,..\, Len(Trace) \\
& \land\ \text{“event”} \in \text{DOMAIN}\ Trace[l] \Rightarrow Trace[l].event = e \\
& \land\ l' = l + 1 \\
& \land\ UpdateVariables(Trace[l])
\end{aligned}
$$

Full Trace Specification for Two-Phase Commit

```
──────────────── MODULE TwoPhaseTrace ────────────────
EXTENDS TwoPhase, TVOperators, TraceSpec
UpdateVariables(ll) ≜
   ∧ IF "rmState" ∈ DOMAIN ll
      THEN rmState' = UpdateVariable(rmState, ll.rmState)
      ELSE TRUE
   ∧ ...
TraceInit ≜ l = 1 ∧ TPInit
IsTMCommit ≜ IsEvent("Commit") ∧ TMCommit
IsTMRcvPrepared ≜
   ∧ IsEvent("TMRcvPrepared")
   ∧ IF "event_args" ∈ DOMAIN Trace[l]
      THEN TMRcvPrepared(Trace[l].event_args[1])
      ELSE ∃r ∈ RM : TMRcvPrepared(r)
...
TraceNext ≜ IsTMCommit ∨ IsTMRcvPrepared ∨ ...
──────────────────────────────────────────────────────
```

UpdateVariable(old, upd)
predefined operator, applies the update from the JSON entry

TMCommit, TMRcvPrepared, TPInit
operators from original two-phase commit specification

Overall trace specification
schematic operator definitions, could largely be mechanized

What Property Should TLC Check?

- Liveness: the trace will eventually be processed

$\Diamond(l = Len(Trace))$

$\Rightarrow$ too strong: some executions may terminate early

What Property Should TLC Check?

- Liveness: the trace will eventually be processed $\Diamond(l = Len(Trace))$

  $\Rightarrow$ too strong: some executions may terminate early

- Impossibility: the trace cannot be processed $\Box(l \neq Len(Trace))$

  $\Rightarrow$ negative logic: doesn't provide feedback when validation fails

What Property Should TLC Check?

- Liveness: the trace will eventually be processed

  $\Diamond(l = Len(Trace))$

  $\Rightarrow$ too strong: some executions may terminate early

- Impossibility: the trace cannot be processed

  $\Box(l \neq Len(Trace))$

  $\Rightarrow$ negative logic: doesn't provide feedback when validation fails

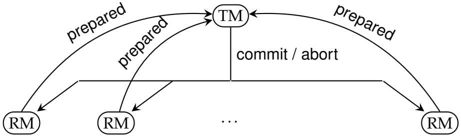
- There exists some path of full length in the constrained state space
  - expressed as a post-condition *TraceAccepted*

    $$TraceAccepted \triangleq Len(Trace) = TLCGet(\text{"stats"}).diameter - 1$$

  - counter-example: maximum-length prefix that cannot be extended
  - TLA$^+$ debugger can be used to navigate the state space

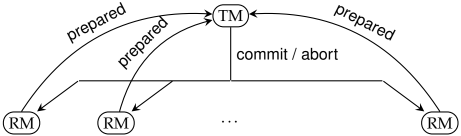
Example of Trace Validation at Work

- Implementation accounts for message loss



  - RM resends message after some timeout if no order from TM has arrived
  - resending corresponds to stuttering in TLA^+ since messages are stored in a set

Example of Trace Validation at Work



- Implementation accounts for message loss
  - RM resends message after some timeout if no order from TM has arrived
  - resending corresponds to stuttering in TLA^+ since messages are stored in a set
- However, counting messages is no longer correct
  - TM cannot distinguish between original and resent messages
  - trace validation quickly reveals the problem: commit may be sent prematurely
  - modify implementation to store identities of RMs instead of counting

Experience with Trace Validation

- Approach applied to several algorithms
  - two-phase commit protocol
  - distributed key-value store, implemented according to existing TLA^+ specification
  - distributed termination detection (EWD998)
  - two open-source implementations of Raft consensus protocol
  - Microsoft Confidential Consortium Framework: reverse-engineered TLA^+ specification[1]
- Instrumenting the implementations was easy

[1] Howard et al.: *Smart Casual Verification of CCF's Distributed Consensus and Consistency Protocols.* NSDI'25.

Experience with Trace Validation

- Approach applied to several algorithms
  - two-phase commit protocol
  - distributed key-value store, implemented according to existing TLA+ specification
  - distributed termination detection (EWD998)
  - two open-source implementations of Raft consensus protocol
  - Microsoft Confidential Consortium Framework: reverse-engineered TLA+ specification[1]
- Instrumenting the implementations was easy
- Trace validation quickly found discrepancies in every case
  - problems may indicate implementation errors or overly strict specification
  - identified serious bugs in CCF implementation
  - spurious discrepancies due to mismatch in "grain of atomicity"

[1] Howard et al.: *Smart Casual Verification of CCF's Distributed Consensus and Consistency Protocols.* NSDI'25.

Accommodating different grains of atomicity

- Implementation steps may be invisible at the specification level
  - essentially harmless: stuttering transitions
  - avoid indicating action name, e.g. message resending from wrong sender state

Accommodating different grains of atomicity

- Implementation steps may be invisible at the specification level
  - essentially harmless: stuttering transitions
  - avoid indicating action name, e.g. message resending from wrong sender state
- Implementation step may correspond to several abstract transitions
  - e.g., combine *UpdateTerm* and *LearnEntries* actions in Raft
  - have instrumentation emit two actions in succession
  - may provide explicit disjunct in trace specification using action composition

Accommodating different grains of atomicity

- Implementation steps may be invisible at the specification level
  - essentially harmless: stuttering transitions
  - avoid indicating action name, e.g. message resending from wrong sender state
- Implementation step may correspond to several abstract transitions
  - e.g., combine *UpdateTerm* and *LearnEntries* actions in Raft
  - have instrumentation emit two actions in succession
  - may provide explicit disjunct in trace specification using action composition
- Decide when and what to log
  - programming languages do not explicitly indicate atomic steps
  - typically: log when shared state is updated (network, locks, data bases etc.)

Precision vs. Numbers of Explored States (Valid Traces)

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	∞

VEA	variables and actions with arguments
V	only variables
VpEA	variables and some actions
EA	only actions with arguments
E	only action names

bfs / dfs exploration

Conclusions and Perspectives

- Lightweight approach to validating implementations
  - easiest to apply when the TLA$^+$ specification is known to the programmer
  - model checker can fill in values when specification variables are not recorded
  - surprisingly effective for finding implementation errors

- Future / ongoing work
  - streamline the toolchain, aim for (even) more genericity
  - support for analysis and visualization of counter-examples
  - let model checker fill in missing actions
  - leverage formal specification for generating “interesting” traces
  - online monitoring instead of off-line trace validation?