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

Stephan Merz
joint work with Horatiu Cirstea, Markus Kuppe, Benjamin Loillier

Inria & LORIA, Nancy, France

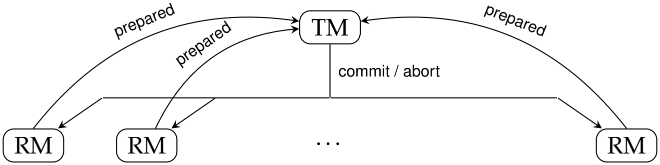
TLA$^+$ Community Meeting

Milan, September 2024

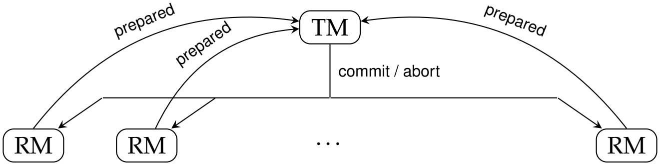
Motivation

- Relate TLA$^+$ specifications and distributed programs
  - TLA$^+$ specifications abstract from implementation details
  - implementations usually have much finer grain of atomicity
  - also need to cater for networking semantics, exception handling etc.
  - formal refinement proofs are tedious
- Lightweight approach for finding bugs
  - instrument (Java) code to record transitions at specification level
  - log updates of specification variables and/or occurrences of actions
  - use TLC to check if the trace corresponds to some allowed behavior

Running Example: Two-Phase Commit from GitHub Examples



Running Example: Two-Phase Commit from GitHub Examples



- Two possible TM transitions

handle "prepared" message from RM r

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

send "commit" order to all RMs

$$
\begin{array}{l}
TMCommit \triangleq \\
\quad \land\ tmState = \text{“init”} \\
\quad \land\ tmPrepared = RMs \\
\quad \land\ tmState' = \text{“done”} \\
\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{“commit”}]\} \\
\quad \land\ \text{UNCHANGED}\ rmState
\end{array}
$$

Java Implementation of Two-Phase Commit

- Classes implementing the algorithm
  - ResourceManager may send "prepared" message, listens for "abort" / "commit"
  - TransactionManager listens for "prepared" messages, aborts after timeout
  - NetworkManager relays messages between processes, based on Java sockets
- Harness running the algorithm
  - read configuration from JSON file and set up processes
  - simulate system execution, including delays and failures
- Structurally quite different from the TLA^+ specification

Instrumenting the Java Implementation for Logging Traces

Two methods from class `TransactionManager`

```
protected void receive(Message msg) throws IOException {
  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));
  }


}
```

Instrumenting the Java Implementation for Logging Traces

Two methods from class `TransactionManager` with instrumentation

```
protected void receive(Message msg) throws IOException {
  if (msg.getContent().equals(TwoPhaseMessage.Prepared)) {
    preparedRMs ++;        // implementation counts "prepared" messages
    specTmPrepared.add(msg.getFrom());       // record variable update
    spec.log("TMRcvPrepared", new Vector(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));
  }
  specMessages.add(Map.of("type", TwoPhaseMessage.Commit.toString()));
  spec.log("TMCommit");
}
```

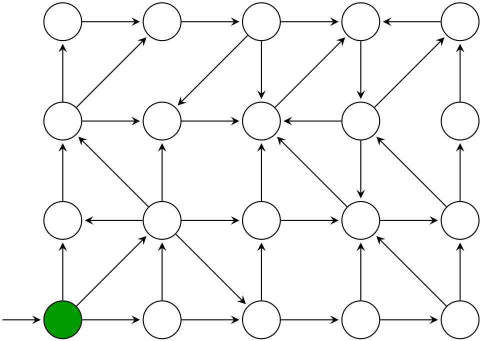
Java API and Python Scripts for Collecting Traces

- Collect updates of specification variables
  - programmer maps implementation data to values of TLA$^+$ specification
  - need not specify updates for all variables
  - log method assembles updates, adds time stamp, and optionally records action
- Class `TLATracer` facilitates the instrumentation
  - support for shared (physical) and logical clocks
  - convenience methods for recording (partial) updates of data structures
  - record log as sequence of JSON entries
- Merge traces of individual processes and sort them by timestamps
  - centralized clock: easy to use for simulation, e.g. continuous integration
  - logical clocks when running on separate nodes

Validating the Trace

Trace of implementation

State space of TLA$^+$ specification

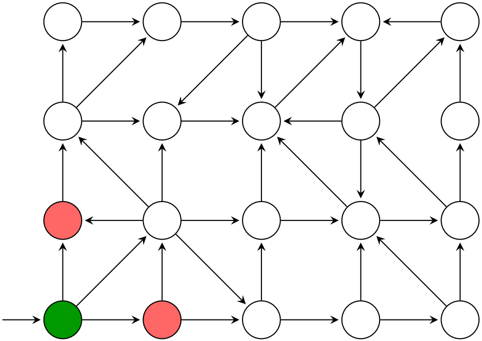


- Does the trace correspond to some execution allowed by the TLA$^+$ specification?

Validating the Trace

Trace of implementation

State space of TLA$^+$ specification

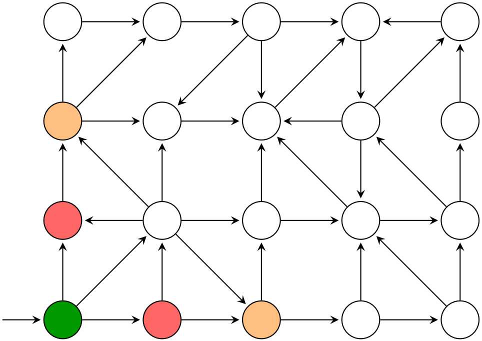


- Does the trace correspond to some execution allowed by the TLA$^+$ specification?

Validating the Trace

Trace of implementation

State space of TLA+ specification

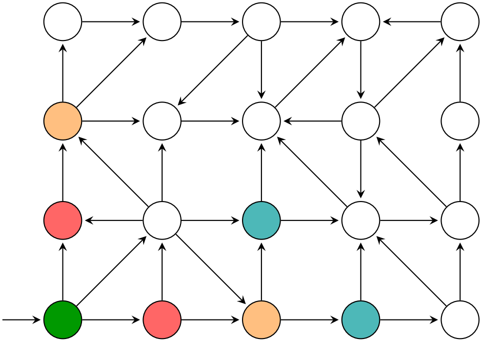


- Does the trace correspond to some execution allowed by the TLA+ specification?

Validating the Trace

Trace of implementation

State space of TLA$^+$ specification

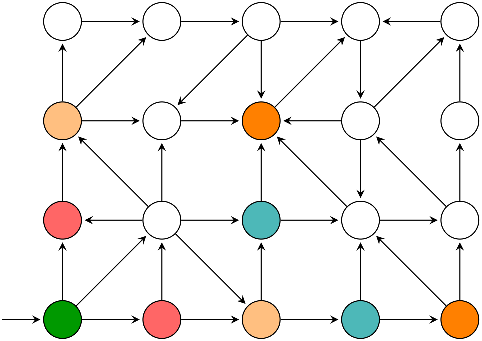


- Does the trace correspond to some execution allowed by the TLA$^+$ specification?

Validating the Trace

Trace of implementation

State space of TLA$^+$ specification

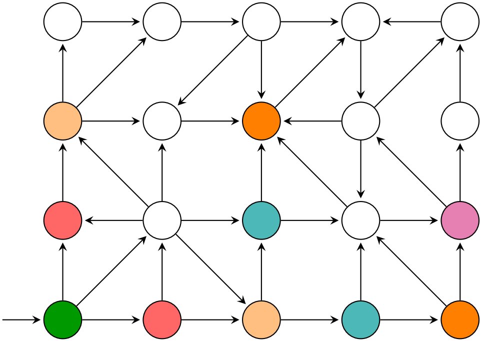


- Does the trace correspond to some execution allowed by the TLA$^+$ specification?

Validating the Trace

Trace of implementation

State space of TLA$^+$ specification

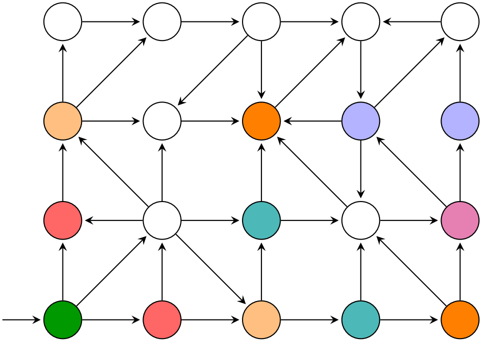


- Does the trace correspond to some execution allowed by the TLA$^+$ specification?

Validating the Trace

Trace of implementation

State space of TLA$^+$ specification



- Does the trace correspond to some execution allowed by the TLA$^+$ specification?
- Can be reduced to a model checking problem, using the trace as a constraint

Generic Setup of Trace Checking Using TLC

```
──────────────────────── MODULE TraceSpec ────────────────────────
EXTENDS TLC, Integers, Sequences, Json, IOUtils
Trace ≜ ndJsonDeserialize(IOEnv.TRACE_PATH)
VARIABLE l        \* current line in trace
IsEvent(e) ≜ ∧ l ∈ 1 .. Len(Trace)
             ∧ "event" ∈ DOMAIN Trace[l] ⇒ Trace[l].event = e
             ∧ l' = l + 1
             ∧ UpdateVariables(Trace[l])
TraceAccepted ≜ Len(Trace) = TLCGet("stats").diameter − 1
──────────────────────────────────────────────────────────────────
```

- load trace produced by system run as a TLA$^+$ value
- action *IsEvent* tracks progress through the trace
- post-condition *TraceAccepted* ensures that at least one matching behavior was found

Trace Checking for Two-Phase Commit

MODULE *TwoPhaseTrace*

EXTENDS *TwoPhase*, *TVOperators*, *TraceSpec*

$UpdateVariables(ll) \triangleq$
$\quad \land$ IF "rmState" $\in$ DOMAIN ll
$\quad\quad$ THEN $rmState' = UpdateVariable(rmState, ll.rmState)$
$\quad\quad$ ELSE TRUE
$\quad \land \ldots$

$IsTMCommit \triangleq IsEvent(\text{"Commit"}) \land TMCommit$

$IsTMRcvPrepared \triangleq$
$\quad \land IsEvent(\text{"TMRcvPrepared"})$
$\quad \land$ IF "event_args" $\in$ DOMAIN $Trace[l]$
$\quad\quad$ THEN $TMRcvPrepared(Trace[l].event\_args[1])$
$\quad\quad$ ELSE $\exists r \in RM : TMRcvPrepared(r)$

$\ldots$

$TraceInit \triangleq TPInit \land l = 1$

$TraceNext \triangleq IsTMCommit \lor IsTMRcvPrepared \lor \ldots$

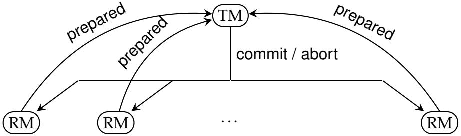
UpdateVariable(old, upd)
predefined operator, applies the update from the JSON entry

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

Overall trace specification
schematic operator definitions, could largely be mechanized

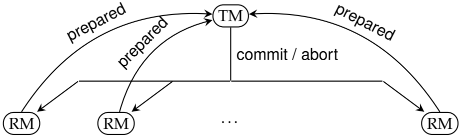
Extending the Implementation for Supporting Failures

- Take into account potential message loss



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

Extending the Implementation for Supporting Failures



- Take into account potential message loss
  - RM resends message after a 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

- Considered 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
  - instrumenting the implementations was quite easy

Experience with Trace Validation

- Considered 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
  - instrumenting the implementations was quite 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”

Accommodating different grains of atomicity

- Implementation steps may be invisible for the specification
  - 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 for the specification
  - 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 *AppendEntries* actions in Raft
  - must decide if this acceptable or not
  - provide explicit disjunct in trace specification using action composition

Accommodating different grains of atomicity

- Implementation steps may be invisible for the specification
  - 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 *AppendEntries* actions in Raft
  - must decide if this acceptable or not
  - provide explicit disjunct in trace specification using action composition
- Decide when and what to log
  - programming languages do not provide atomic transitions
  - typically: log when shared state is updated (network, locks, data bases etc.)

Technical Aspects

- Tradeoff between precision and efficiency
  - track only some specification variables, indicate TLA^+ actions or not
  - less information in the trace may lead to state explosion during validation
  - consider using constrained depth-first rather than breadth-first search

Technical Aspects

- Tradeoff between precision and efficiency
  - track only some specification variables, indicate TLA$^+$ actions or not
  - less information in the trace may lead to state explosion during validation
  - consider using constrained depth-first rather than breadth-first search
- Explaining failures
  - counter-example: longest prefix of execution that cannot be extended
  - it would be desirable to also show other failures to complete
  - TLC debugger can be used to explore the constrained state graph

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 some variables and actions

EA only actions with arguments
E only action names
bfs / dfs exploration

Conclusions and Perspectives

- Lightweight approach to validating implementations
  - easy to apply when the TLA$^+$ specification is known to the programmer
  - generic, reusable framework mixing Java, TLA$^+$, and scripts for running the tools
  - model checker can fill in values for specification variables left open
  - surprisingly effective for finding implementation errors
- Future / ongoing work
  - streamline the toolchain, aim for (even) more genericity
  - improve analysis and visualization of counter-examples
  - leverage model checker for steering the implementation?
  - explore online monitoring instead of off-line trace validation