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

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

Inria & LORIA, Nancy, France

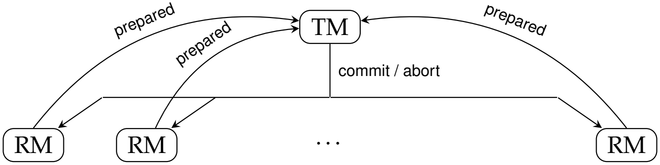
IFIP Working Group 2.3

Trento, October 2023

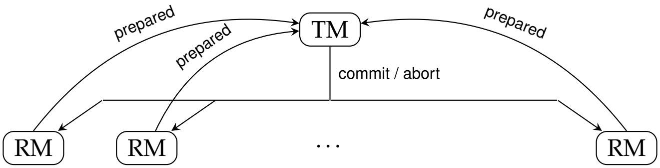
Motivation

- TLA$^+$ has good support for high levels of abstraction
  - verify properties using model checking or theorem proving
  - industry-strength approach to formal specification and verification
- Leverage specifications for gaining confidence in implementations
  - formally proving refinement is tedious
  - lightweight approach: validate individual executions
- Objective: framework for validating logs of distributed Java programs
  - instrument code to record relevant updates to system state
  - check that all transitions are allowed by the specification

Running Example: Two-Phase Commit



Running Example: Two-Phase Commit



- Two transitions described in TLA$^+$

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

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

Java Implementation of Two-Phase Commit

- Classes implementing the algorithm
  - TransactionManager listens for “prepared” messages, aborts after timeout
  - ResourceManager may send “prepared” message, listens for “abort” / “commit”
  - NetworkManager relays messages between processes, based on Java socket library
  - plus a few helper classes (message objects, handle system shutdown etc.)

Java Implementation of Two-Phase Commit

- Classes implementing the algorithm
  - TransactionManager listens for "prepared" messages, aborts after timeout
  - ResourceManager may send "prepared" message, listens for "abort" / "commit"
  - NetworkManager relays messages between processes, based on Java socket library
  - plus a few helper classes (message objects, handle system shutdown etc.)
- 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)) {
      spec.startLog();
      preparedRMs ++;        // implementation counts "prepared" messages
      specTmPrepared.add(msg.getFrom());
      spec.endLog("TMRcvPrepared", new Vector(msg.getFrom()));
   }
}

private void commit() throws IOException {    // assumes preparedRMs == resourceManagers.size()
   spec.startLog();
   for (String rm : resourceManagers) {
      networkManager.send(new Message(getName(), rm, TwoPhaseMessage.Commit));
   }
   specMessages.add(Map.of("type", TwoPhaseMessage.Commit.toString()));
   spec.endLog("TMCommit");
}
```

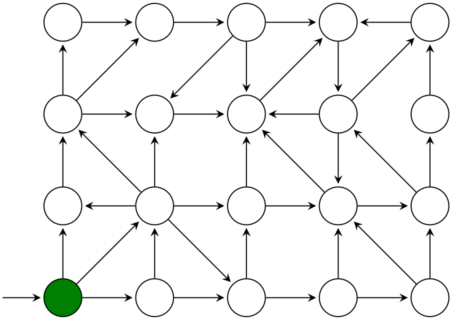
Logging Events

- An event collects relevant state updates
  - startLog obtains timestamp of event
  - record updates to one or more specification variables
  - do not require values to be provided for all variables
  - endLog collects updates and formats them as JSON entries
- Class `TLATracer` provides support for logging events
  - support for shared (physical) and logical clocks
  - convenience methods for recording (partial) updates of data structures
- When trace is complete, sort it according to clock values

Validating the Trace

Trace of implementation

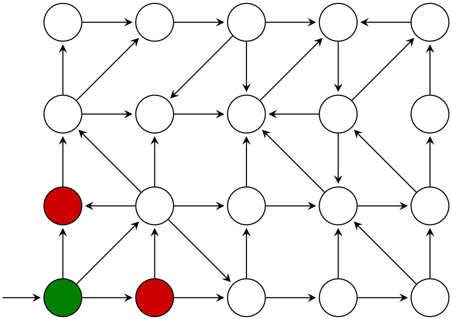
State space of TLA$^+$ specification



Validating the Trace

Trace of implementation

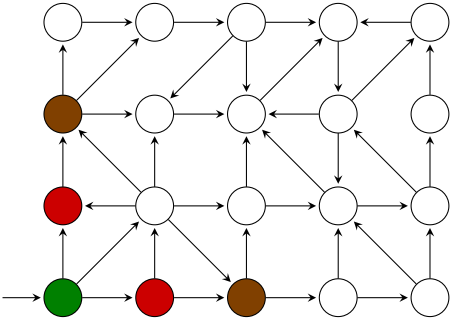
State space of TLA$^+$ specification



Validating the Trace

Trace of implementation

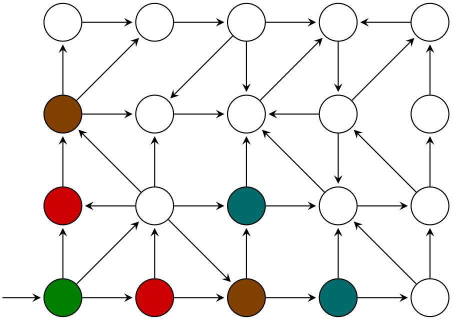
State space of TLA$^+$ specification



Validating the Trace

Trace of implementation

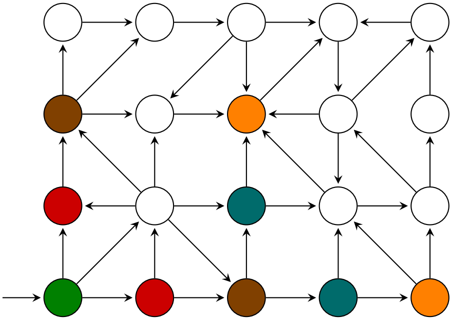
State space of TLA$^+$ specification



Validating the Trace

Trace of implementation

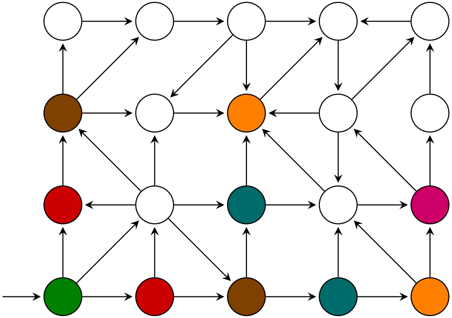
State space of TLA$^+$ specification



Validating the Trace

Trace of implementation

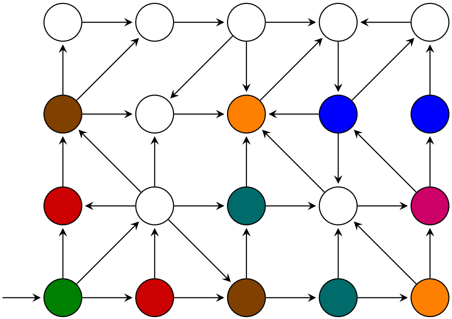
State space of TLA$^+$ specification



Validating the Trace

Trace of implementation

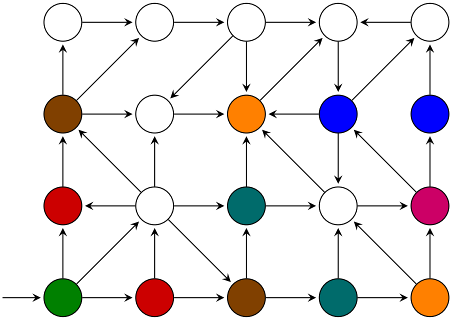
State space of TLA$^+$ specification



Validating the Trace

Trace of implementation

State space of TLA$^+$ specification



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

Generic Setup of Trace Checking Using TLC

MODULE *TraceSpec*

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

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

VARIABLE l * *current line in trace*

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

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

- load trace produced by system run
- 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 *TLC*, *TwoPhase*, *TVOperators*, *TraceSpec*

$MapVariables(t) \triangleq$
$\quad \land$ IF "rmState" $\in$ DOMAIN t
$\quad\quad$ THEN $rmState' = MapVariable(rmState, \text{"rmState"}, t.rmState)$
$\quad\quad$ ELSE TRUE
$\quad \land \ldots$

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

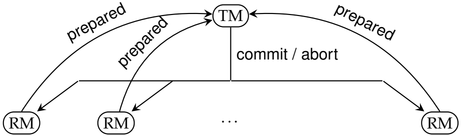
$IsTMRcvPrepared \triangleq$
$\quad \land IsEvent(\text{"TMRcvPrepared"})$
$\quad \land$ IF "event_args" $\in$ DOMAIN $Trace[l]$ THEN $TMRcvPrepared(Trace[l].event\_args[1])$
$\quad\quad$ ELSE $\exists r \in RM : TMRcvPrepared(r)$

...

$TraceInit \triangleq TPInit \land l = 1$

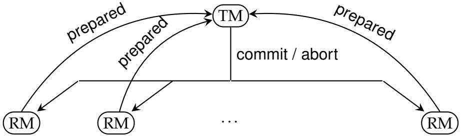
$TraceNext \triangleq IsTMCommit \lor IsTMRcvPrepared \lor \ldots$

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
  - this is allowed by the TLA$^+$ specification: *msg* variable records all sent messages

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
  - this is allowed by the TLA$^+$ specification: *msg* variable records all sent messages
- However, counting messages is no longer correct
  - TM cannot distinguish a resent message from an original message send
  - 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 four algorithms
  - two-phase commit protocol
  - distributed key-value store, implemented according to existing TLA$^+$ specification
  - MicroRaft implementation of Raft consensus protocol
  - consensus protocol used at Microsoft, also based on Raft

Experience with Trace Validation

- Considered four algorithms
  - two-phase commit protocol
  - distributed key-value store, implemented according to existing TLA$^+$ specification
  - MicroRaft implementation of Raft consensus protocol
  - consensus protocol used at Microsoft, also based on Raft
- Trace validation quickly found discrepancies in every case
  - instrumenting implementations was straightforward
  - some care is required for mapping code to atomic TLA$^+$ transitions
  - tradeoff between precision of logging and state reconstruction using TLC
  - problems may indicate implementation errors or overly strict specification

Conclusions and Perspectives

- Lightweight approach to verifying implementations
  - easy to apply, assuming that the programmer knows the high-level specification
  - generic, reusable framework mixing Java and TLA$^+$
  - use of model checker obviates need for tracking all specification variables
  - surprisingly effective for finding implementation errors
- Ongoing work
  - application to more use cases from industry
  - streamline the toolchain, aim for (even) more genericity
  - leverage model checker for steering the implementation?
  - explore online monitoring instead of off-line trace validation