Automated Validation of State-Based Client-Centric Isolation with TLA$^+$

Tim Soethout[1,2(✉)], Tijs van der Storm[2,3], and Jurgen J. Vinju[2,4]

[1] ING Bank, Amsterdam, The Netherlands
tim.soethout@ing.com
[2] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
tim.soethout@cwi.nl
[3] University of Groningen, Groningen, The Netherlands
[4] Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. Clear consistency guarantees on data are paramount for the design and implementation of distributed systems. When implementing distributed applications, developers require approaches to verify the data consistency guarantees of an implementation choice. Crooks *et al.* define a state-based and client-centric model of database isolation. This paper formalizes this state-based model in TLA$^+$, reproduces their examples and shows how to model check runtime traces and algorithms with this formalization. The formalized model in TLA$^+$ enables semi-automatic model checking for different implementation alternatives for transactional operations and allows checking of conformance to isolation levels. We reproduce examples of the original paper and confirm the isolation guarantees of the combination of the well-known 2-phase locking and 2-phase commit algorithms. Using model checking this formalization can also help finding bugs in incorrect specifications. This improves feasibility of automated checking of isolation guarantees in synthesized synchronization implementations and it provides an environment for experimenting with new designs.

Keywords: Distributed systems · Model checking · Isolation guarantees

1 Introduction

Automatically generating correct and performant implementations from high-level specifications is an important challenge in computer science and software engineering. Ideally one makes high-level specifications, which completely describe the functional and relevant parts of an application, without having to bother with low-level implementation details at the same time. Implementation is left to specialized tools and approaches that benefit from automated model checking and other debugging tools.

A benefit of high-level specifications is that they enable more specialized and fine-tuned implementations than general purpose implementation strategies,

L. Cleophas and M. Massink (Eds.): SEFM 2020 Workshops, LNCS 12524, pp. 43–57, 2021.
https://doi.org/10.1007/978-3-030-67220-1_4

which in essence have to take into account all possible variations of operations users can define. High-level domain knowledge offers the potential to automatically generate and optimize code, e.g. removing locks and blocking for improved performance when it can derive that this is never necessary the specific situation.

Such optimizations often involve managing concurrency and parallelism on accessing data. These optimizations of course need to be correct w.r.t. the specification: data consistency needs to be guaranteed. Application logic defines the functional consistency and transaction isolation manages the consistency of concurrent operations. Historically, isolation concerns have been outsourced to database systems, using general purpose transactions and similar constructs. These databases generally support ACID transactions, with a variety of isolation guarantees [2,7], where Serializability is the strongest guarantee.

In order to optimize the performance of specialized implementations, some parts of the general purpose transaction mechanism incorporated either in the application itself or in the database implementation. When developing these specialized implementations of higher-level specifications, we need to be sure that they guarantee the ACID properties, or, if not, to what extent. The seminal definition of isolation levels is given by Adya [1]. Adya uses transaction histories, where transactions have dependencies on each other based on accessing the same data. If a cycle can be found in the graph of these dependencies, an isolation anomaly is present. Crooks *et al.* [6] model a state-based and client-centric approach to isolation and prove that it is equivalent to Adya's formalization.

Various tools are available which try to find or visualize isolation anomalies [14,16,18]. Many rely on specific scripted error scenarios to show anomalies. The ELLE tool [14] can be used to validate of traces of implementations using Adya's formalization, but still required careful setup and tuning of a test setup. It infers the histories Adya requires from client-centric observed transactions. Crooks' formalization is defined from a client-centric perspective and is directly defined in terms of observed transactions. The state-based and client-centric isolation definitions of Crooks *et al.* are referenced as Crooks' Isolation (CI) throughout this paper.

This paper describes an approach using formal methods to (semi-) automatically validate the isolation level of observed transactions using CI. First, we give an introduction to CI and a formalization of it in TLA$^+$. Next we discuss how this formalization is used to validate the consistency guarantees of a transaction algorithm using two-phase commit (2PC) with two-phase locking (2PL), and use it to find a specification bug.

The formalization of CI and the TLA$^+$ model checker enable rapid checking of multiple isolation levels of different synchronization algorithms. This technique can be used to both validate observed transactions from run-time systems and of formalizations of algorithms.

The main contributions of this paper are:

1. Formalization of the core of CI in TLA$^+$ and updated definitions to allow incremental model checking (Sect. 3).
2. Reproduction of the claims and properties [6] using model checking (Sect. 4).

$$\overset{S_0}{\left\{\begin{matrix} A \mapsto 100 \\ B \mapsto 100 \end{matrix}\right\}} \xrightarrow{T_1} \overset{S_1}{\left\{\begin{matrix} A \mapsto 150 \\ B \mapsto 50 \end{matrix}\right\}} \xrightarrow{T_2} \overset{S_2}{\left\{\begin{matrix} A \mapsto 165 \\ B \mapsto 55 \end{matrix}\right\}}$$

Fig. 1. Example execution with initial state S_0 for transactions $T_1 = \langle r(A, 100), r(B, 100), w(A, 150), w(B, 50)\rangle$ and $T_2 = \langle r(A, 150), r(B, 50), w(A, 165), w(B, 55)\rangle$.

3. Formalization of 2PL/2PC in TLA$^+$ and validation of Serializability using model checking of the CI TLA$^+$ formalization (Sect. 5).
4. An example of finding isolation bugs in the algorithm specification of 2PL/2PC (Sect. 5.3).

Section 6 discusses results, limitations and future work based on this approach. We conclude in Sect. 7. All source code can be found on Zenodo [24].

2 Background: State-Based Client-Centric Consistency

Crooks *et al.* [6] define a state-based and client centric consistency model (CI) for reasoning about isolation levels. It defines predicates to state if a set of observed transactions occurs under a given isolation level. The main concepts of CI are transactions and executions. A transaction is a sequence of operations, consisting of reads and writes which includes observed keys and values: $r(k, v)/w(k, v)$. An execution represents a possible ordering of a set of transactions with the resulting intermediate database states. A state is a mapping from all database keys to a specific value. Within an execution each following state only differs in the values written by the intermediate transaction on the previous or parent state.

Figure 1 shows an example execution of two bank accounts A and B, which both have a balance of €100 in the initial state S_0. Transaction T_1 is money transfer: €50 is deposited from account A and withdrawn from account B, realized using two reads and two writes. Transaction T_2 is paying of interest: 10% of the balance is added to both accounts; this transaction also involves two reads and two writes. Note that from a starting state and an ordering of transactions the other states can be derived by applying the intermediate transaction's writes.

For a set of observed transactions T to satisfy an isolation level I, a commit test CT for I should hold for a possible execution e of T: $\exists e : \forall t \in T : CT_I(t, e)$. The commit test for serializability, for example, is that all reads in a transaction must be able to have read their value from the direct parent state. In our example all the values of T_1's and T_2's read operations are the same as their parent state's values for each corresponding key, e.g. T_1's $r(A, 100)$ can read from T_1's parent S_0's $A \mapsto 100$.

Another isolation level is Snapshot Isolation, where the commit test requires that all reads of a single transaction can be read from the same earlier, not necessarily parent, state, which represents the database snapshot.

3 Formalizing CI in TLA$^+$

TLA$^+$ [20] is a formal specification language for action-based modeling of programs and systems. PlusCal [19] is an abstraction on top of TLA$^+$ for concurrent and distributed algorithms and compiles to TLA$^+$. In practice TLA$^+$ is used to model distributed algorithms and systems [5,9,11,21,22]. TLA$^+$ models states and transitions. A specification defines an initial state and atomic steps to a next state. Complex state machines and their transitions can be represented this way. Multiple concurrently-running state machine define their local steps and the global next step non-deterministically picks one machine to progress each step. This captures all possible interleavings of these multiple machines.

CI is formalized as properties that hold on a TLA$^+$ state. This enables querying the system if an initial database state together with an a set of observed transactions satisfies an isolation level, e.g., `Serializability(initalState,setOfTransactions)`. When using TLA$^+$ to formally specify an algorithm, this isolation property is added as an invariant during model checking. TLA$^+$'s model checker TLC can then check the isolation guarantees at every state in the algorithm's execution and produce a counter example if the invariant is violated.

To formalize CI, we assume the following TLA$^+$ definitions:

```
State       == [Keys → Values]
Operation   == [op: {"read", "write"}, key: Keys, value: Values]
Transaction == Seq(Operation)
ExecutionElem == [parentState: State, transaction: Transaction]
Execution   == Seq(ExecutionElem)
```

The system `State` is modeled as a mapping from keys to values. `Keys` and `Values` are left abstract on purpose here, since they differ per concrete model. In TLA$^+$ sets and set membership are often used. `[Keys → Values]` represents the set of possible tuples of `Keys` and `Values`, we bind this to `State` to easily reference this later in the specification. `Operations` are a read or write of a value on a key and a `Transaction` is a sequence of these operations. An `Execution` is represented as a sequence of transactions with their parent state.

As intuitively sketched earlier CI checks if values could have been read from earlier states. The following definition of $\mathcal{RS}$ ("read states") captures this for an execution e and an operation $o = r(k, v)$:

$$\mathcal{RS}_e(o) = \left\{ s \in S_e \,\middle|\, \overbrace{s \xrightarrow{*} s_p}^{\text{a}} \wedge \left(\overbrace{(k,v) \in s}^{\text{b1}} \vee \overbrace{(\exists w(k,v) \in \Sigma_T : w(k,v) \xrightarrow{to} r(k,v))}^{\text{b2}} \right) \right\}$$

Read states are a subset of the states in the execution S_e, which are: (a) up to and including the parent state s_p in the execution; (b1) have the same key and value as the operation $o = r(k, v)$; or (b2) there exists a write operation $w(k, v)$ with the same key and value earlier in the same transaction's operations (Σ_T).

Listing 1. TLA$^+$ ReadStates

```
1 ReadStates(execution, operation, transaction) ==
2   LET Se == SeqToSet(executionStates(execution))
3       sp == parentState(execution, transaction)
4   IN { s ∈ Se: \* s ∈ S_e
5       ∧ beforeOrEqualInExecution(execution, s, sp) \* a: s →* s_p
6       ∧ ∨ s[operation.key] = operation.value \* b1: (k, v) ∈ s
7         \* b2: ∃w(k, v) ∈ Σ_T
8         ∨ ∃ write ∈ SeqToSet(transaction):
9           ∧ write.op = "write" ∧ write.key = operation.key
10          ∧ write.value = operation.value
11          \* b2: w(k, v) →to r(k, v)
12          ∧ earlierInTransaction(transaction, write, operation)
13        ∨ operation.op = "write"
14    }
```

The TLA$^+$ version of this definition is shown in Listing 1. These read states are defined for each operation given an execution. TLA$^+$'s syntax allows grouping of conjunctions ($\wedge$) and disjunctions ($\vee$) by vertical indentation. The function `executionStates` denote the sequence of states in an `execution`. `parentState` extracts the parent state of a `transaction` given an `execution`. `LET .. IN` has the standard semantics. The rest of `ReadStates` (Lines 4 to 5) follows the CI definition quite literally, except that the third alternative (Line 13) is not captured in the CI definition for $\mathcal{RS}$ above, but represents the "convention [that] write operations have read states too" [6] to include all states up until the parent state for writes.

A state is complete when all reads of a transaction could have read their values from it. It is the intersection of the states in which each operation of the transaction could read from. The following definition is extended to take into account transactions without operations to support the iterative construction of transactions, starting with the empty ones:

$$\text{COMPLETE}_{e,T}(s) \equiv s \in \left(\bigcap_{o \in \Sigma_T} \mathcal{RS}_e(o) \cap \left\{ s' \in S_e | s' \xrightarrow{*} s_p \right\} \right)$$

We omit the TLA$^+$ version (`Complete`) for the sake of brevity, but it closely follows the mathematical definition, just like `ReadStates` did compared to $\mathcal{RS}$.

A *commit test* $CT_I(T, e)$ determines if a set of transactions T is valid under an isolation level I and execution e. For a set of transactions to satisfy an isolation level, there needs to exist at least one possible ordering, for which the commit test holds for all transactions. Transactions describe the values that a client observes including the actual values read and written. The values observed by the client are compatible with an ordering of the transactions that satisfies the isolation level. This is why it is sufficient for a single possible execution ordering to satisfy the commit test. The specific commit test for an isolation level I abstracts over which reads are valid for I.

Listing 2. TLA$^+$ helper definitions for CI.

```
WriteSet(transaction) == \* W_T = {k | w(k,v) ∈ Σ_T}
  LET writes == { operation ∈ SeqToSet(transaction) : operation.op = "write" }
  IN  { operation.key : operation ∈ writes }

NoConf(execution, transaction, state) ==  \* NO-CONF_T(s) ≡ Δ(s, s_p) ∩ W_T = ∅
  LET Sp    == parentState(execution, transaction)
      delta == { key ∈ DOMAIN Sp : Sp[key] ≠ state[key] }
  IN  delta ∩ WriteSet(transaction) = {}

Preread(execution, transaction) == \* PREREAD_e(T) ≡ ∀o ∈ Σ_T : RS_e(o) ≠ ∅
  ∀ operation ∈ SeqToSet(transaction): ReadStates(execution, operation, transaction) ≠ {}

strictBefore(t1, t2, timestamps) == timestamps[t1].commit < timestamps[t2].start \* T1 <_s T2
beforeOrEqualInExecution(execution, state1, state2) ==  \* s1 →* s2
  LET states == executionStates(execution)
  IN  Index(states, state1) <= Index(states, state2)
```

Different isolation-level commit tests are shown in Table 1, both mathematically and in TLA$^+$. Note that the CI definitions and their TLA$^+$ counterparts are very similar. The definitions of `NoConf`, `Preread`, `strictBefore` and `beforeOrEqualInExecution` can be found in Listing 2.

4 CI Examples

The static examples of the CI-paper are reproduced using TLA$^+$'s model checker TLC and the `ASSUME` operator. The model checker checks if the assumed property is valid. Figure 2 shows a minimal example of transactions `ta` to `te`, which are checked for four different isolation levels given initial state `s0`. TLC checks the assumptions and all evaluate to **`TRUE`**. The source code [24] reproduces more checks on this example.

Bank Transfer Example. The bank transfer example introduced by Crooks *et al.*, shows the difference between Snapshot Isolation and Serializability. Alice and Bob simultaneously take money out of their joint current and savings accounts, both from the other account. The bank requires the sum of the balances of both accounts to stay positive.

The following execution contains the transactions $T_{alice} = \langle r(S,30), r(C,30), w(C,-10)\rangle$ and $T_{bob} = \langle r(S,30), r(C,-10), abort\rangle$. A serializable implementation requires T_{bob} to abort. T_{alice} reads both balances of C and S and withdraws €40 from C. T_{bob} reads the result and aborts because not enough balance is available for his withdraw of €40 from S:

$$\overset{S_1}{\left\{\begin{array}{l} C \mapsto 30 \\ S \mapsto 30 \end{array}\right\}} \xrightarrow{T_{alice}} \overset{S_2}{\left\{\begin{array}{l} C \mapsto -10 \\ S \mapsto 30 \end{array}\right\}} \xrightarrow{T_{bob}} \overset{S_3}{\left\{\begin{array}{l} C \mapsto -10 \\ S \mapsto 30 \end{array}\right\}}$$

The TLA$^+$ code to check this is shown on the right of Fig. 2.

Table 1. Commit tests and corresponding TLA$^+$ definitions.

Isolation Level	Commit Test	TLA$^+$ definition
Serializability	$\text{COMPLETE}_{e,T}(s_p)$	`Complete(e, T, parentState(e, T))`
Snapshot Isolation	$\exists s \in S_e.$ $\text{COMPLETE}_{e,T}(s_p) \wedge$ $\text{NO-CONF}_T(s)$	`∃ s ∈ toSet(states(e)): Complete(e, T, s) ∧ NoConf(e, T, s)`
Read Committed	$\text{PREREAD}_e(T)$	`Preread(e ,T)`
Read Uncommitted	True	`TRUE`
Strict Serializability	$\text{COMPLETE}_{e,T}(s_p) \wedge$ $\forall T' \in \mathcal{T}:$ $T' <_s T \Rightarrow s_{T'} \xrightarrow{*} s_T$	`LET Sp == parentState(e, t) IN Complete(e, T, Sp) ∧ ∀ otherT ∈ transactions(e): strictBefore(otherT, T, timestamps) ⇒ beforeOrEqualInExecution( e, parentState(e, otherT), Sp)`

```
\* Initial State, all 0
s0 == [k ∈ {x,y,z} ↦ 0]
\* Helper functions for operations
r(k,v) ==
  [op ↦ "read",  key ↦ k, value ↦ v]
w(k,v) ==
  [op ↦ "write", key ↦ k, value ↦ v]

ta == << w(x,1) >>
tb == << r(y,1), r(z,0) >>
tc == << w(y,1) >>
td == << w(y,2), w(z,1) >>
te == << r(x,0), r(z,1) >>

trs == {ta, tb, tc, td, te}
ASSUME Serializability(s0, trs)
ASSUME SnapshotIsolation(s0, trs)
ASSUME ReadCommitted(s0, trs)
ASSUME ReadUncommitted(s0, trs)
```

```
\* Initial state of Current and
     ↪ Savings accounts.
bInit == (C :>  30) @@ (S :> 30)

talice ==
  << r(S,30), r(C, 30), w(C,-10) >>
tbob   ==
  << r(S,30), r(C,-10)
     (* w(S,-10) does not happen *) >>
bTrx == {talice, tbob}

ASSUME Serializability(bInit, bTrx)
ASSUME SnapshotIsolation(bInit, bTrx)
ASSUME ReadCommitted(bInit, bTrx)
ASSUME ReadUncommitted(bInit, bTrx)
```

Fig. 2. Running example (left) and serializable bank account example (right) from Crooks *et al.* [6] in TLA$^+$.

The same example is considered under Snapshot Isolation with transactions $T_{alice} = \langle r(S,30), r(C,30), w(C,-10)\rangle$ and $T_{bob} = \langle r(S,30), r(C,30), w(S,-10)\rangle$. Both T_{alice} and T_{bob} read from S_1 and find that there is enough total balance available. They both withdraw €40 from respectively C and S:

$$\overset{S_1}{\left\{\begin{array}{l} C \mapsto 30 \\ S \mapsto 30 \end{array}\right\}} \xrightarrow{T_{alice}} \overset{S_2}{\left\{\begin{array}{l} C \mapsto -10 \\ S \mapsto \ \ 30 \end{array}\right\}} \xrightarrow{T_{bob}} \overset{S_3}{\left\{\begin{array}{l} C \mapsto -10 \\ S \mapsto -10 \end{array}\right\}}$$

Snapshot Isolation allows this because both T_{alice} and T_{bob} read from a valid snapshot or complete state and there is no conflict in their writes, because they write to different accounts. However, this violates the overall invariant that the sum of the balances should remain positive. This is the write skew isolation anomaly [1]. This can be checked by using a specification similar to the right-hand side of Fig. 2, with modified transactions, and assuming Serializability is `FALSE` .

5 Model Checking Algorithms Using CI

In contrast to the previous, static examples, where TLA$^+$'s state steps are not used, we now look at a TLA$^+$ specification of a transactional protocol (2PL/2PC) using states. At each step of the algorithm TLC checks if the isolation guarantees hold.

5.1 Formalizing 2PL/2PC

Two-Phase Commit (2PC) combined with Two-Phase Locking (2PL) forms a protocol used to implement ACID transactions. 2PC takes care of atomicity of a transaction and 2PL provides Serializable isolation. We extend the formalization of 2PC by Gray and Lamport [9] to support multiple parallel transactions via 2PL.

We model 2PL/2PC in the PlusCal algorithm language, which is compiled down to regular TLA$^+$, but provides a higher-level notation, closer to imperative programming languages. PlusCal describes multiple possibly different processes with atomic steps. During model checking, one of the processes takes a single step, which allows processes to be interleaved. The model checker makes sure all possible interleavings are explored.

The PlusCal encoding of 2PL/2PC consists of two types of processes: transaction managers and transaction resources. The actual number of processes is defined by model constants `transactions` and `resources`. Message passing is modeled by a monotonically growing set of messages. This means that messages are never lost, but a recipient process might handle them out of order or not at all.

Listing 3 shows the definition of the transaction manager. There is a `tm` process for each of the `transactions`. PlusCal processes do atomic steps, each represented by a label such as `INIT:`. A label can intuitively be viewed as a state in the process' state machine. All statements within a step are done as a single step.

Listing 3. PlusCal specification of 2PL/2PC manager

```
fair process tm ∈ transactions
begin
  INIT: sendMessage([id ↦ self, type ↦ "VoteRequest"]);
  WAIT: either \* receive commit votes
    await ∀ rm ∈ resources: [id ↦ self, type ↦ "VoteCommit", rm ↦ rm] ∈ msgs;
    sendMessage( [id ↦ self, type ↦ "GlobalCommit"]);
    goto COMMIT;
  or \* receive at least 1 abort votes
    await ∃ rm ∈ resources: [id ↦ self, type ↦ "VoteAbort", rm ↦ rm] ∈ msgs;
    sendMessage([id ↦ self, type ↦ "GlobalAbort"]);
    goto ABORT;
  or \* or timeout, solves deadlock when transactions lock each others resources
    sendMessage([id ↦ self, type ↦ "GlobalAbort"]);
    goto ABORT;
  end either;
  ABORT: goto Done; COMMIT: goto Done;
end process
```

A transaction manager first sends out the VoteRequest message by adding a tuple with the transaction's identifier `self` and the message label `"VoteRequest"` to the `msgs` set. Then its next step is `WAIT` in which three alternatives (`either ... or`) can occur: 1) either it receives messages of type `"VoteCommit"` of each resource occurring in the set of messages, and sends GlobalCommit; 2) or one message of type `"VoteAbort"` and sends GlobalAbort; 3) or it times out and aborts (to prevent deadlock). The `await` construct ensures that a step only happens if its precondition is fulfilled. TLC makes sure that all alternatives are explored. `goto`'s are added to explicitly label the steps for readability in the model checker's execution. `Done` is a special PlusCal label, which represents the process being completed.

The PlusCal specification of a transaction resource, shown in Listing 4, is slightly more involved. The resource process has local variables (Lines 1 to 6) to keep track of stopping, votes, commits, aborts and resource state. The state is used for CI as a symbolic state, represented by an integer.

When the resource is started (Lines 7 to 18), it `either` does nothing (`skip`) and decrements `maxTxs`, `or` receives a `"VoteRequest"` message. `with tId ∈ transactions\ voted` denotes choosing a transaction ID `tId` from the set of `transactions` minus the transactions already `voted` for. The resource can then either VoteCommit or VoteAbort. The `voted` local variable keeps track of the transactions it has already voted for and is updated to make sure to only vote once per transaction.

Next, it becomes `READY` (Lines 19 to 30) and waits on either GlobalCommit or GlobalAbort, but only for transactions which it voted for, and has not committed yet. It keeps track of the `committed` and `aborted` transactions in order to not send duplicate messages and to later check the atomicity of the transactions. Each `while` iteration, decrements `maxTxs` to ensure termination.

Listing 4. 2PL/2PC resource in TLA$^+$

```
1  fair process tr ∈ resources
2  variables maxTxs = 5,      \* Limit number of transactions to limit search space
3            voted = {},      \* Transactions which this resource voted for
4            committed = {}, \* Committed transactions
5            aborted = {},    \* Aborted transactions
6            state = 0;       \* Counter to represent state changes for CC
7  begin TR_INIT:
8  while maxTxs >= 0 do
9    either skip; \* skip to not deadlock
10   or \* Wait on VoteRequest
11     with tId ∈ transactions \ voted do
12       await [id ↦ tId, type ↦ "VoteRequest"] ∈ msgs;
13       either \* If preconditions hold, VoteCommit, else VoteAbort
14          sendMessage([id ↦ tId, type ↦ "VoteCommit", rm ↦ self]);
15          voted := voted ∪ {tId};
16       or sendMessage([id ↦ tId, type ↦ "VoteAbort", rm ↦ self]);
17          voted := voted ∪ {tId}; aborted := aborted ∪ {tId}; goto STEP;
18       end either; end with;
19   READY: \* Wait on Commit/Abort
20     either with tId ∈ voted \ committed do  \* receive GlobalCommit
21         await [id ↦ tId, type ↦ "GlobalCommit"] ∈ msgs;
22         committed := committed ∪ {tId};
23         operations[tId] := operations[tId] ∘ << r(self, state), w(self, state+1) >>;
24         state := state + 1;
25       end with;
26     or with tId ∈ voted \ aborted do \* receive GlobalAbort
27          await [id ↦ tId, type ↦ "GlobalAbort"] ∈ msgs;
28          aborted := aborted ∪ {tId};
29       end with; end either; end either;
30   STEP: maxTxs := maxTxs - 1;
31 end while; end process;
```

In order to model check CI it captures the read and written values in `operations` (Line 23) and updates its local `state`. Both reads and writes are added on commit and not on vote, because if reads are added on vote, it could be the case that the resource reads a later committed value when responding to the VoteRequest later which will always be aborted anyway. This results in a violation of Serializability for the CI check, while it is technically never an observed value.

5.2 Model Checking 2PL/2PC

As sanity check for the formalization of 2PL/2PC, first atomicity and termination are checked:

```
Atomicity ==
  ∀ id ∈ transactions: pc[id]=Done ⇒
    ∀ a1, a2 ∈ resources: ¬ id ∈ aborted[a1] ∧ id ∈ committed[a2]

AllTransactionsFinish == <>(∀ t ∈ transactions: pc[t] = Done)
```

For atomicity, when all transactions are completed (process counter `pc` is `Done`), for all pairs of resources it should not (¬) be the case that a transaction is aborted by one resource, but committed by the another. So all should either committed or aborted the transaction. Property `AllTransactionsFinish` makes sure that eventually (<>) all transactions complete.

To model check the isolation guarantees an instance of the CI formalization is added, which gives access to the previously defined isolation level tests (see Sect. 4), given the initial state and the observed transactions.

```
ccTransactions == Range(operations)    \* Operations without transaction ids
InitialState == [k ∈ resources ↦ 0] \* Initial state is 0 for all resources

CC == INSTANCE ClientCentric WITH Keys ← resources, Values ← 0..10
Serializable      == CC!Serializability(InitialState, ccTransactions)
SnapshotIsolation == CC!SnapshotIsolation(InitialState, ccTransactions)
ReadCommitted     == CC!ReadCommitted(InitialState, ccTransactions)
ReadUncommitted   == CC!ReadUncommitted(InitialState, ccTransactions)
```

In this case all cases are valid when we run the TLC model checker for `transactions == {t1, t2}` and `resources == {r1, r2, r3}`.

The model checker then checks the isolation guarantees for each step of the algorithm. When the isolation test fails, it presents a counter example. Table 2 gives an intuition on the relative time durations of the TLC model checker on different numbers of transactions and resources. The model checker checks the four CI isolation levels (Serializability, Snapshot Isolation, Read Committed, Read Uncommitted) on each of the model's steps. It never invalidates the checks, so it traverses the entire state space.

Table 2. Run time durations of TLC on CI checks for different number of transactions and resources n of 2PL/2PC. Results on MacBook Pro (13-inch, 2016) with 3,3 GHz Intel Core i7 with 4 worker threads and allocated 8 GB RAM on AdoptOpenJDK 14.0.1+7, on TLC 2.15 without profiling and using symmetry sets for constants.

#tx	$n = 1$	$n = 2$	$n = 3$
1	7 s	9 s	19 s
2	8 s	21 s	5 m 55 s
3	11 s	1 m 53 s	3 h 21 m 54 s

5.3 2PL/2PC Bug Seeding

To additionally stress the formalization presented above, we have introduced a subtle, but realistic bug in the definition of transaction resource. When the resource is in the ready state and waiting on a GlobalCommit or GlobalAbort message from the transaction manager, the resource should only wait for

these messages when it is the actual transaction it voted for. This is guaranteed by `with tId ∈ voted \ committed` in Listing 4 Line 20. The bug is to replace this with `with tId ∈ transactions \ committed`. This means `tId` can faultily represent a never-seen before transaction as well.

When this model is checked with two transactions and resources, all of the invariants hold and no problem is found. However, with three transactions and two resources the Serializability invariant is violated and a counter example with 20 steps is found within half a minute; this trace shown in Fig. 3. The example shows that due to this bug it is possible for a resource to side-step an in progress transaction, by responding to the GlobalCommit of a different transaction.

First `t1` and `t2` request to vote and `r1` votes to commit for `t1`, then `t2` aborts due to timeout with GlobalAbort(t2). `r1` then uses this abort to abort its waiting on `t1`. This is possible because `with tId ∈ transactions \ committed` allows `r1`. It receives the GlobalAbort(t2), aborts and steps to receive the next transaction. The model checker requires some more steps to find non-serializable behavior, when the other transactions `t1` and `t3` commit and their effects are applied in different order on `r1` and `r2`, hence the system is not serializable.

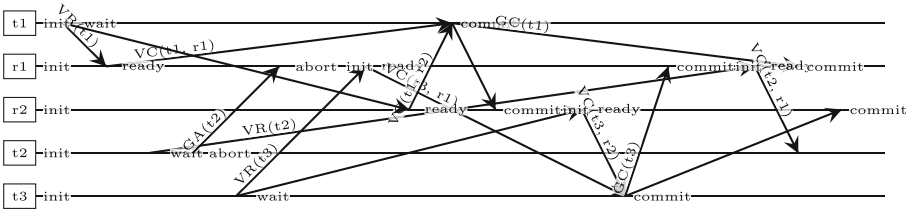

Fig. 3. Non-serializable trace found for bugged 2PL/2PC specification. Horizontal lines represent processes over time with state changes. Arrows represent messages sent and received. Message labels are abbreviations of 2PC messages: VoteRequest, VoteCommit, GlobalAbort and GlobalCommit.

These kinds of bugs during specification can occur naturally, for example when specializing algorithms for specific applications with the goal of added efficiency [25]. Using CI in model checking helps us find bugs while designing new algorithms and also for validating claims of existing algorithms.

6 Discussion and Future Work

The formalization of CI in TLA$^+$ is relatively straightforward. The definitions for the base abstractions, such as `State` and `Execution`, influence the whole formalization. Staying as close as possible to the mathematical model however, results in quite verbose output, since there are no labels on transactions. The definition on read states was improved to support incremental model checking, starting with empty transactions.

The main limitation of using model checking to find isolation violations is the state explosion when the numbers of processes grows. As seen in Table 2, running times grow rapidly and model checking becomes infeasible when more transactions are added. Since the model checker evaluates the isolation guarantees in every algorithm state we assume, however, that most isolation violations can be found in small examples. The small scope hypothesis [13] supports this saying that most bugs have small counterexamples. Nevertheless, we can not be entirely sure that anomalies that only occur in larger interactions and longer traces are found by the current approach, but it gives us confidence in the the checked isolation level, while keeping it feasible.

There is a lot of research focusing on proving distributed consistency properties. Model checking tools, such as Uppaal [3], Spin [12], LTSMin [4], mCRL2 [10] and TLA$^+$ [20] are used to verify distributed systems and algorithms as well as real-world implementations and protocols [8,11,21,22].

There are also many approaches [2,17,23,28] that try to balance the trade-off between performance and data-consistency by choosing different isolation guarantees. Our work adds to this knowledge by providing a reusable framework to investigate and model check distributed consistency protocols.

To further evaluate the usefulness of our approach for real-life systems, it would be insightful to reproduce known isolation bugs in older versions of database implementations, such as found by Jepsen [14,15] and Bailis *et al.* [2]. In order to do this we could either create one or more clients that capture the observed transactions, or instrument the database to store this information for offline model checking.

Furthermore the scripts of the isolation anomalies of Hermitage [18] can be reproduced as TLC model checks to strengthen (our formalization of) CI. The TLA$^+$ Toolbox also features a theorem prover. The CI formalizations could be extended by proving certain properties, such as reproducing the proofs on equivalence with Adya's formalization and proving conformity to isolation levels for specific algorithms.

Generating performant and correct implementations from high-level specifications is an attractive goal in software engineering, as it would bring the benefits of (semi-)automatic verification to correct-by-construction implementation.

For instance, the Rebel domain-specific language has been used to specify realistic systems (for instance, in the financial domain), from which highly scalable implementations are generated using novel consistency algorithms [25–27]. It is however, a far from trivial endeavour to state and prove isolation guarantees of some of these algorithms. CI can be extended to support operations on a semantically higher level than reads and writes, such as the semantically richer operations used in Rebel. A TLA$^+$ formalization can then be used to allows for rapid prototyping of synchronization implementation alternatives for Rebel, while leveraging the higher-level semantics [29]. The checking of isolation guarantees can then be automated.

7 Conclusion

This paper formalizes Crooks' state-based client-centric isolation model (CI) in TLA$^+$ in order to check conformance to isolation levels using model checking. The running examples of Crooks *et al.* [6] are reproduced and validated in TLA$^+$. An example of a transaction implementation using two phase locking (2PL) and two phase commit (2PC) is formalized in TLA$^+$. The TLC model checker is used to automatically show conformance to the CI formalization. The CI formalization is also used to find a bug in the algorithm's formalization.

Formalizing CI in TLA$^+$ enables automatic validation of isolation guarantees of synchronization implementations by mapping their algorithms to read and write operations. It can be used both for checking isolation conformance of run-time traces of (distributed) systems and of formal specification of algorithms.

References

1. Adya, A.: Weak consistency: a generalized theory and optimistic implementations for distributed transactions. Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science (1999)
2. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions - virtues and limitations. Proc. VLDB Endow. **7**(3), 181–192 (2013). https://doi.org/10.14778/2732232.2732237
3. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL—a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0020949
4. Blom, S., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_31
5. Brooker, M., Chen, T., Ping, F.: Millions of tiny databases. In: Bhagwan, R., Porter, G. (eds.) 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, 25–27 February 2020, pp. 463–478. USENIX Association (2020)
6. Crooks, N., Pu, Y., Alvisi, L., Clement, A.: Seeing is believing. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, pp. 73–82. ACM, July 2017. https://doi.org/10.1145/3087801.3087802
7. Fekete, A., Liarokapis, D., O'Neil, E.J., O'Neil, P.E., Shasha, D.E.: Making snapshot isolation serializable. ACM Trans. Database Syst. **30**(2), 492–528 (2005). https://doi.org/10.1145/1071610.1071615
8. Gomes, V.B., Kleppmann, M., Mulligan, D.P., Beresford, A.R.: Verifying strong eventual consistency in distributed systems. Proc. ACM Program. Lang. **1**(OOPSLA), 1–28 (2017). https://doi.org/10.1145/3133933
9. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (2006). https://doi.org/10.1145/1132863.1132867
10. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
11. Gustafson, J., Wang, G.: Hardening Kafka replication (2020). https://github.com/hachikuji/kafka-specification

12. Holzmann, G.J.: The SPIN Model Checker - Primer and Reference Manual. Addison-Wesley, Boston (2004)
13. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press, Cambridge (2006)
14. Kingsbury, K., Alvaro, P.: Elle: inferring isolation anomalies from experimental observations. CoRR abs/2003.10554 (2020)
15. Kinsbury, K.: Jepsen: distributed systems safety research (2020). http://jepsen.io/
16. Kinsbury, K.: Knossos (2020). https://github.com/jepsen-io/knossos
17. Kleppmann, M.: Designing Data-Intensive Applications: The Big Ideas behindReliable, Scalable, and Maintainable Systems. O'Reilly, Sebastopol (2016)
18. Kleppmann, M.: Hermitage: testing transaction isolation levels (2020). https://github.com/ept/hermitage
19. Lamport, L.: The PlusCal Algorithm Language - Microsoft Research. https://www.microsoft.com/en-us/research/publication/pluscal-algorithm-language/
20. Lamport, L.: Specifying Systems, the TLA+ Language and Tools for Hardwareand Software Engineers. Addison-Wesley, Boston (2002)
21. Microsoft: High-level TLA+ specifications for the five consistency levels offered by Azure Cosmos DB (2020). https://github.com/Azure/azure-cosmos-tla
22. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015). https://doi.org/10.1145/2699417
23. Preguiça, N.M., Baquero, C., Shapiro, M.: Conflict-free replicated data types CRDTs. In: Sakr, S., Zomaya, A.Y. (eds.) Encyclopedia of Big Data Technologies. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-77525-8_185
24. Soethout, T.: TimSoethout/tla-ci: TLA+ specifications used in "Automated Validation of State-Based Client- Centric Isolation with TLA+". Zenodo (2020). https://doi.org/10.5281/zenodo.3961617
25. Soethout, T., van der Storm, T., Vinju, J.: Path-sensitive atomic commit. Programming **5**(1) (2020). https://doi.org/10.22152/programming-journal.org/2021/5/3
26. Soethout, T., van der Storm, T., Vinju, J.J.: Static local coordination avoidance for distributed objects. In: Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2019, pp. 21–30. ACM Press, Athens (2019). https://doi.org/10.1145/3358499.3361222
27. Stoel, J., van der Storm, T., Vinju, J., Bosman, J.: Solving the bank with Rebel: on the design of the Rebel specification language and its application inside a bank. In: Proceedings of the 1st Industry Track on Software Language Engineering - ITSLE 2016, pp. 13–20. ACM Press (2016). https://doi.org/10.1145/2998407.2998413
28. Tanenbaum, A.S., van Steen, M.: Distributed Systems - Principles and Paradigms, 2nd edn. Pearson Education, Upper Saddle River (2007)
29. Weikum, G.: Principles and realization strategies of multilevel transaction management. ACM Trans. Database Syst. **16**(1), 132–180 (1991). https://doi.org/10.1145/103140.103145