Model Checking the Deferred Update Replication Protocol

Odorico Machado Mendizabal[1,2], **Fernando Luís Dotti**[2]

[1]Universidade Federal do Rio Grande – FURG
Rio Grande – RS – Brazil

[2]Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre – RS – Brazil

odoricomendizabal@furg.br, fernando.dotti@pucrs.br

Abstract. *As the number of distributed applications and the volume of generated data increase, support from robust data management systems becomes even more necessary. Since these management systems possibly have to deal with heavy workloads, in this paper we analyze the deferred update replication, a successful technique to implement highly available and performing transactional databases. Although it offers a strong consistency semantics, no enough efforts have been invested to prove its properties. Due to its critical requirements, in this paper we verify the deferred update replication protocol using model checking. A model of the deferred update replication protocol and a comprehensive set of safety and liveness properties are presented. According to our investigation, all properties hold for the presented model, leading to a higher confidence in the protocol's correctness.*

1. Introduction

The increase in the number of distributed applications and the high volume of data being generated demand support from robust data management systems. Besides providing high availability, these data management systems must be capable to scale up without affecting performance nor consistency. Underlying protocols for reliable communication and replication techniques are largely adopted in the development of those kind of systems. Of special interest in this case is the deferred update technique, that aims to achieve both high availability and performance of transactional dependable data management systems [Pedone et al. 1997].

Like distributed algorithms in general, guaranteeing that update replication protocols run correctly and efficiently is not trivial. Although this class of protocols suggests a strong consistency semantics, little efforts have been invested to formally prove their properties. Most of the authors in the literature argue about the correctness of the deferred update protocol reasoning in natural language [Garcia et al. 2011, Pedone and Schiper 2012]. Even though this approach gives a general understanding of the reasons behind design aspects of the implemented protocol, it hardly will detect non trivial failures presented in the solution.

A few works introduce some kind of formal proof of correctness for the deferred update replication technique [Pedone et al. 1997, Kemme and Alonso 2000, Garcia et al. 2011], and a few contributions use the theorem proving approach

This work is partially sponsored by FAPERGS and CNPq projects SAN Model Checking.

[Schmidt and Pedone 2007, Armendáriz-Iñigo et al. 2009]. However we could not identify the use of semi-automated reasoning in the later cases. For the theorem proving approach, a formal description of the model is needed for the reasoning process and the reasoning itself can be conducted in a semi-automated or non automated way, the last one being more error prone than the first. In either case, the verification process would be arduous and time consuming.

This paper presents a formal verification of the deferred update replication protocol by using model checking. The model checking technique is very attractive due to its simplicity of use combined with a solid theoretical foundation on verification approach. While it does not require high skilled specialists as usually needed for theorem proving, it offers a very expressive resource for properties specification through temporal logic. Since the deferred update protocol strongly relies on the atomic broadcast protocol to ensure correct progress of the servers, we first present properties and a model for the atomic broadcast which will be used as building block for the specification of the deferred update protocol. We took Promela [Holzmann 1991] and Spin [Holzmann 1997] as modeling language and model checking tool respectively. Promela offers abstractions very close to the ones typically used while building distributed algorithms, such as sequential processes, messages and channels.

The rest of the paper is structured as follow: Sections 2 and 3 illustrate the models and verification for atomic broadcast and deferred update replication protocols. A discussion about related work is presented in Section 4 and Section 5 concludes this paper.

2. A Promela Model for the Atomic Broadcast Protocol

The atomic broadcast protocol provides reliable communication channels for message exchange in distributed environments. It guarantees that a group of communicating processes delivers the same set of messages to every process following the same delivery order [Défago et al. 2004]. Due to the ordering guarantees, this protocol is often used by distributed algorithms such as replication algorithms. The total ordering delivery provides deterministic update on database replicas [Agrawal et al. 1997, Pedone et al. 1997, Kemme and Alonso 2000].

Delivery guarantees are preserved since the protocol satisfies the following properties [Hadzilacos and Toueg 1994]:

- *Validity*: If a correct process broadcasts a message m, then it eventually delivers m.
- *Uniform Agreement*: If a process delivers a message m, then all correct processes eventually deliver m.
- *Uniform Integrity*: For any message m, every process delivers m at most once, and only if m was previously broadcast by $sender(m)$.
- *Uniform Total Order*: If both processes p and q deliver messages m and m', then p delivers m before m', if and only if q delivers m before m'.

Next, we describe a simple model for the atomic broadcast and verify the protocol's properties using model checking. In our model, every process p_i running the protocol uses a channel $abcast_i$ for reliable communication. External processes that do not participate in the protocol have no access to $abcast$ channels.

Algorithm 1 shows the primitive *send* modeled in Promela [Holzmann 1991]. For representation of message passing, operators ! and ? are used for write and read over channels, respectively. An atomic block is used to ensure that no other process will execute while the send operation in execution has not finished. The number of channels used by the protocol is represented by $num\_servers$.

Algorithm 1 Send(m)

```
1:  atomic {
2:      do
3:          :: i < num_servers →
4:          assert(nfull(abcast));
5:          abcast[i]!m;
6:          i++;
7:          :: i == num_servers →
8:          i = 0;
9:          break;
10:     od
11:     abcast_send_count++;
12: }
```

Since Promela channels are bounded, write operation in a full channel causes the waiting process to block and the atomic block, if any, looses atomicity. The assert command (line 4) is used to check if the *abcast* channel is not full before writing in the channel. This means that the channel size has to be calculated such that it does not become full during verification. With this the sender process will not block and atomicity is assured. If the channel is full, the assert statement will produce an error during the verification. It is important because atomicity is preserved only if no blocking commands are executed in the atomic block. In case of assertion error, it is necessary to set a larger buffer size to the channel and resume the verification.

Primitive *deliver* performs a read on $abcast$ channel. According to Promela semantics, reading messages from a channel follows a FIFO order. Thus, during a reading operation, p_i consumes the oldest message from channel $abcast_i$. As described in Algorithm 2, an array of channels allows processes p_0 to p_{n-1} to read messages from channels $abcast[0]$ to $abcast[n-1]$, respectively.

Algorithm 2 Deliver(m)

```
1: do
2:     :: abcast[id]?m →
3:     skip
4: od
```

In this example, the content of a message is represented by m. Although it has not been described in the algorithm, right after delivering a message, a process $p[i]$ copies m to a list of received messages ($p[i].msg\_list[]$). That step just mimics an application adding each delivered message to a list. Moreover, in order to differentiate messages, an incremental monotonic counter is used to generate exclusive message's content.

We set up a scenario with 3 processes executing and a maximum number of 8 atomic broadcast messages being sent by them. Due to the exhaustive combination among processes execution, the model checker tool creates a total state space containing all possible execution behaviors for this model. That is, traces considering every possible order of execution among the processes are represented.

In order to specify atomic broadcast properties we use Linear Temporal Logic (LTL) formulas [Manna and Pnueli 1991]. LTL allows representation and reasoning about propositions qualified in terms of time. Thus, it is possible to express formulas considering the future of the paths. Next we state atomic broadcast properties using LTL formulas:

Validity: This property states that if a correct process p_i broadcasts a message m, then m will be eventually delivered to p_i. We added arrays $sent[]$ and $delivered[]$ to our model for verification purposes. $sent[i]$ ($delivered[i]$) is updated whenever a new message is sent (delivered) by the i^{th} process. Propositions $p1\_send\_m1$ and $p1\_deliver\_m1$ are defined as $sent[1] = m1$ and $delivered[1] = m1$, respectively. Thereafter, we write validity property in LTL as follows: $\Box(p_1\_send\_m1 \rightarrow \Diamond p_1\_deliver\_m1)$.

Operators $\Box$ and $\Diamond$ represent the globally and eventually conditions. Thus, for all states in generated traces, if $p_i\_send\_m1$ is true, then in a future state $p_i\_deliver\_m1$ must be true.

Although we have illustrated formulas for individual instances m_1 and m_2, we also checked the formulas for general messages m_i and m_j with i and j representing other scenarios. The processes in this model are identical, that means there is no difference on their behaviors. Therefore, we do not need to check every property for processes individually. Once a property holds for p_1, it also holds for other processes p_i.

Uniform Agreement: If a process delivers a message m, then all correct processes eventually deliver m. In other words, uniform agreement states that a message m will be delivered to every process sooner or latter. Thus, we can write uniform agreement formula in LTL as follow: $(\Diamond p1\_deliver\_m1) \rightarrow (\Diamond p2\_deliver\_m1 \wedge \Diamond p3\_deliver\_m1)$.

The use of operator $\Diamond$ (eventually) indicates that there is a time in the future in which message $m1$ will be delivered to each process.

Uniform Integrity: For any message m, every process delivers m at most once, and only if m was previously sent by a process. To check this property, we first verify that a message m is delivered iff m was previously sent by a process using the precedence pattern described in [Salamah et al. 2005]: $\neg p1\_deliver\_m1\ U\ (m1\_sent \vee \Box\neg p1\_deliver\_m1)$. That means $m1$ will never be delivered until $m1$ has been sent by some process ($m1\_sent$ is defined as $sent[0] = m1 \vee sent[1] = m1 \vee sent[2] = m1$).

In order to check that every process delivers m at most once, we used Promela's assert statements which verify whether a boolean condition specified holds. Since we have distinct messages being sent, and each process has a list with delivered messages ($msg\_list[]$), we express the following boolean condition: $\forall 0 \leq i, j < total\_messages \wedge i \neq j, (p[id].msg\_list[i] \neq p[id].msg\_list[j]) \vee p[id].msg\_list[i] = \varnothing$. The assert statement checks the boolean condition whenever a message is delivered. As expected, the assertion is always true, *i.e.* the same message is not delivered twice or more.

Uniform Total Order: If processes p_i and p_j both deliver messages m and m', then p_i delivers m before m', iff p_j delivers m before m'. That means all processes must deliver all messages at the same order.

In addition to proposition $p1\_deliver\_m1$, we define $p1\_deliver\_m2$ as $delivered[1] = m2$, $p2\_deliver\_m1$ as $delivered[2] = m1$, and $p2\_deliver\_m2$ as $delivered[2] = m2$. Thus, we can state the uniform total order property through the LTL formula: $\Box(p1\_deliver\_m1 \rightarrow \Diamond p1\_deliver\_m2) \rightarrow \Box(p2\_deliver\_m1 \rightarrow \Diamond p2\_deliver\_m2)$.

All formulas presented above are satisfied by our model. That means the specification holds atomic broadcast properties. The state space generated by this model was easily tractable by Spin and the executions took less than 1 minute and allocated at most 400 MB of memory.

3. Deferred Update Replication

In this paper we focus on the deferred update replication technique. When compared to primary-backup or state-machine replication, this technique presents better performance [Pedone et al. 1997, Kemme and Alonso 2000, Garcia et al. 2011, Luiz et al. 2011]. Its success stems from the independence of coordination between servers during the execution phase. Initially, a single server is chosen to serve a given transaction and is responsible for the execution of the transaction's operations locally. Only when a client requests the commit of a transaction, the request and some additional transaction's information are broadcast to all servers for certification. This optimistic concurrency control reduces dramatically the communication and synchronization between replicated servers.

3.1. Deferred Update Replication Model

Our model is based on algorithms defined in [Pedone and Schiper 2012] and represents transaction and server processes. The transaction process models a transaction being executed by a client, while the server process reproduces a server replica.

Figure 1 depicts a single transaction execution. A transaction life cycle is separated in two phases: execution and termination. *Execution Phase* encompasses all read and write operations, whilst *Termination Phase* certificates a commit request.

Each transaction keeps a read and a write set. The write set (ws) is a set of tuples with $<item, value>$ and the read set (rs) is a set of tuples with $<item, value, version>$. Write operations are executed locally by the transaction. Every updated item is kept local to the client in the ws until the transaction enters the termination phase. If a read operation accesses an item already in ws, the data value is copied directly from ws. Otherwise, if read item is not in ws, the read operation requests the item value from a server replica.

After executing all read and write operations, the *Termination Phase* starts by sending a *commit request* to all replicated servers. Besides containing client and transaction identifiers, the *commit request* also propagates the rs and ws. This information is used by the server in the certification test. If the transaction's rs contains stale information, then the server decides to abort the transaction.

Differently from messages sent during the *Execution Phase*, the *commit request* is sent through the atomic broadcast protocol (solid lines in Figure 1). That is necessary to

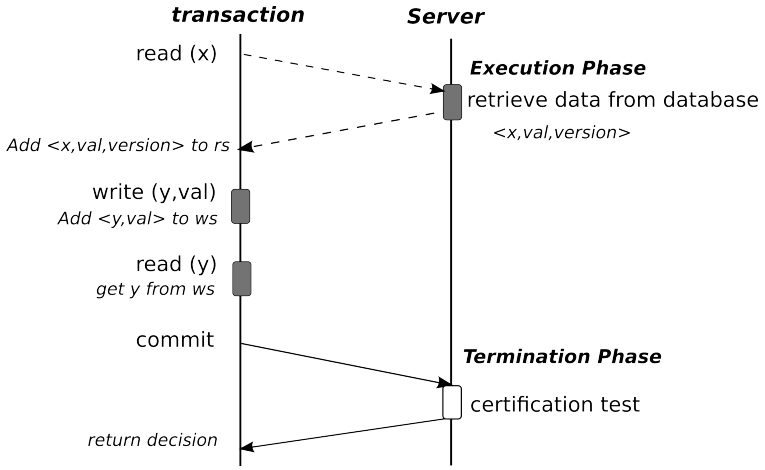

Figure 1. Transaction phases

enforce replicas to receive ongoing transactions in the same order. Once all servers are fully replicated and receive commit messages in the same order, correct servers will take the same decisions in the same order, committing or aborting transactions.

Algorithms 3 and 4 are high level descriptions of transaction and server processes. Message passing over common channels are represented by operators ! and ?. Notice, though, atomic broadcast messages are sent through the *abcast* building block described in previous section. The reuse makes the modeling process simpler and the generated model relies on the building block properties previously verified.

Algorithm 3 T(cid, t)

1: $ws \leftarrow \varnothing$; $rs \leftarrow \varnothing$; $i \leftarrow 0$;
2: choose randomly one of the replica servers s
3: **while** $t.getOp(i) \neq commit \wedge t.getOp(i) \neq abort$ **do**
4: **if** $t.getOp(i) = write$ **then**
5: $ws \leftarrow ws \cup (t.getItem(i), t.getValue(i))$
6: **if** $t.getOp(i) = read$ **then**
7: **if** $t.getItem(i) \in ws$ **then**
8: return v, s.t. $(t.getItem(i), v) \in ws$
9: **else**
10: $c2s[s]!read, t.getItem(i), cid$
11: $s2c[cid]?v, version$ from s
12: $rs \leftarrow rs \cup (t.getItem(i), v, version)$
13: i++;
14: **if** $t.getOp(i) = commit$ **then**
15: $abcast.send(com\_req, cid, t.id, rs, ws)$
16: $s2c[cid]?outcome, s$
17: $t.result = outcome$
18: **else**
19: $t.result = abort$

Before executing operations, one server is randomly selected (l. 2 of Algorithm

3). Write operations do not require communication with the server initially. Instead, they are stored in the write set (ws) (l. 3-4). If a read operation accesses a data item previously updated by the current transaction, then the data value is retrieved from ws (l. 7-8). Otherwise, a read request is sent to the selected server and the received value is added to rs (l. 10-12). When there is no additional read or write operations to execute, the transaction either requests for commit or abort (l. 14 and 18). Further, a commit request message is sent to all replicated servers by atomic broadcast (l. 15). This optimistic approach avoids several communication's rounds throughout the transaction execution.

The server side awaits for messages *read request* or *commit request* (l. 4 and 6 of Algorithm 4). Upon receiving a read request, the server retrieves its value and version for the data item requested (l. 5). For simplicity and reduction of the final size of generated state space, there are only two items in our model (x and y).

Upon receiving a commit request, a certification test verifies if the ongoing transactions ensures serializability [Bernstein et al. 1987]. The server checks if the transaction's read set contains stale items comparing each item's version from rs to the respective item's version in the database. If at least one received item is out of date, the transaction must be aborted (l. 8-11). Otherwise the server decides to commit the transaction. That consists in update item's local version in database, performs all updates according to ws and sent a commit outcome to the requesting transaction (l. 12-19).

Algorithm 4 Server(id)

```
1:  lastCommitted ← 0
2:  db[id].setVersion(x, 0); db[id].setVersion(y, 0)
3:  while true do
4:      :: c2s[id]?read, item, cid →
5:      s2c[cid]!db[id].getVal(item), db[id].getVersion(item)
6:      :: abcast.deliver(com_req, cid, t.id, rs, ws) →
7:      i ← 0; j ← 0;
8:      while rs[i].getItem() ≠ ∅ do
9:          item = rs[i].getItem()
10:         if db[id].getVersion(item) > rs[i].getVersion() then
11:             s2c[cid]!abort, t.id
12:         else
13:             lastCommitted++
14:             db[id].addVersion(item)
15:             while ws[j].getItem() ≠ ∅ do
16:                 item = ws[i].getItem()
17:                 db[id].setItem(item, ws[j].getVal())
18:                 j++
19:             s2c[cid]!commit, t.id
20:         i++
```

Although Pedone *et. al* [Pedone and Schiper 2012] propose optimizations for execution of read only transactions, this aspect has not been yet analyzed thoroughly in our research. Thus, we use the certification test to ensure serializability without distinction between read only or updating transactions.

3.2. Properties Verification

The correct behavior of the protocol can be described by a set of properties. In our investigation, properties were separated in two groups. The first one refers to aspects of the replication approach, such as termination, consistency and agreement in replicated databases. The second specifies scenarios with particular conflicting transactions in order to check if the protocol preserves conflict serializability isolation level.

The scenarios for verification are set up with 2 replicated database servers and 3 transactions t_1, t_2, and t_3, all executing concurrently. Read and write operations are given by $r_i(item)$ and $w_i(item, value)$, respectively, and they are followed by a commit (c_i) or abort (a_i) operation. Transactions t_1 and t_2 are used to specify a particular case of concurrency, while t_3 exploit the non-determinism of model checking to generate any combination of read and write operations over data items x and y. Therefore, during the verification, the transaction t_3 will execute any possible sequence of 3 operations before requesting for commit or abort.

Although t_1 and t_2 express conflicting transactions, the non-determinism of model checking combined with the random generation of t_3 increases the concurrency among transactions' operations. It is a very effective approach, once it increases the verification coverage by inserting automatically execution histories that could hardly be perceived.

3.2.1. Replication Properties

The properties introduced in this section refer to the correct behavior expected by the replication protocol. These safety and liveness properties express termination of transactions, consistency among replicated databases and agreement in transaction's result by the replicas.

For verification we set up transactions t_1 and t_2 as $w_1(x, 11), r_1(y), w_1(y, 21), c_1$, and $r_2(y), r_2(x), w_2(x, 12), c_2$, respectively and t_3 executes up to 3 operations chosen non deterministically before request for commit or abort.

T1 – Transaction Termination: If a transaction is started, then it eventually is decided, *i.e.* it receives a result (commit or abort) from servers.

For verification purposes, we first define propositions $t1\_started = true$ as $t1.started = true$ and $t1\_decided$ as $t1.result \neq 0$, where $t1.started$ is a boolean variable that is set up to true when t_1 starts. Then we state the property T1 as an LTL formula in the form of: $\Box(t1\_started \rightarrow \Diamond t1\_decided)$

The formula specifies that whenever t_1 is started it is eventually decided.

T2 – Uniform Total Order: If two servers s_i and s_j execute transactions t and t', then s_i executes t before t', if and only if s_j executes t before t'.

A transaction finishes its execution in a server once the server answers a commit or abort to that transaction in response to a commit request. Proposition $s_i\_finishes\_t_j$ is defined as $s_i.decided[j] = commit \vee s_i.decided[j] = abort$ and it represents the server decision for a given transaction. Then, the property T2 is specified by the formula $\Box(s1\_finishes\_t1 \rightarrow \Diamond s1\_finishes\_t2) \rightarrow \Box(s2\_finishes\_t1 \rightarrow \Diamond s2\_finishes\_t2)$.

Despite the formula presented above, the use of compositional reasoning would provide a systematic approach to modular verification. Once the deferred update replication model reuses the atomic broadcast model to broadcast messages, the properties verified in previous section would be valid in deferred update replication model as well.

DB1 – Uniform Consistency (Versions): Whenever a server s_i updates an item x to a version v, then every replicated server s_j updates the item x to version v in its instance of the database.

Database replicas consistency is provided once all replicas eventually update the same data items in the same order. In order to check consistency among replica's versions, we verify if (i) a given item x can be updated in a single replica db_i; (ii) for those cases where x is updated, there is no other version of x between versions v_i and v_{i+1}; (iii) all replicated servers execute the same sequence of updates over an item x.

We first need to prove that there are some traces in which the item x is updated to versions v_1 and v_2 (all data items start with version v_0 in our model). Once the model checker generates traces for every possible concurrent execution among transactions as well as all combinations of read and write operations for transaction t_3, it is expected to have situations in which x is updated to v_1 and v_2.

The demonstration of a witness path showing both updates is done by contradiction. Let's say that data item x in replica db_1 will never be updated to versions v_1 and v_2. We can describe it in LTL using the absence pattern [Salamah et al. 2005]: $\Box\neg(\Diamond db1xv1 \wedge \Diamond db1xv2)$, where propositions $db1xv1$ and $db1xv2$ are defined as $db[0].x.version[1] \neq \varnothing$ and $db[0].x.version[2] \neq \varnothing$, respectively. As expected, this property does not hold and a counterexample showing a trace with item x being updated to versions v_1 and v_2 is generated by the model checker tool.

From now on we can check properties (ii) and (iii) against those traces where item x is updated. The correct order for successive updates is verified by the formula $(\Diamond db1xv1 \wedge \Diamond db1xv2) \rightarrow \Box(db1xv1 \rightarrow \Diamond db1xv2)$. Proposition on the left hand side of the logical implication guarantees that traces in which x is not updated twice will not invalidate the formula. On the right hand side of implication we use a response pattern [Salamah et al. 2005], where $db1xv2$ comes after $db1xv1$ globally.

After checking that a single replica is able to update items and that items' version increments in a sequential order, we verify that different replicas perform the same updates in the same order. We state this property as: $(\Box(db1xv_i \rightarrow \Diamond db1xv_{i+1})) \rightarrow \Box(db2xv_i \rightarrow \Diamond db2xv_{i+1})$. That means always the replica db_1 updates x from version v_i to v_{i+1}, then the replica db_2 also updates x in the same order.

DB2 – Uniform Consistency (Values): Two replicated database db_i and db_j have the same value for a same item's version v_i.

That means the value updated in a replica must be the same for the respective version in other replicas. Once we have already verified that different replicas perform the same sequence of updates (property DB1), we check whether the updated values are the same. We use contradiction to show that two replica servers db_1 and db_2 eventually update an item x to the same version. The formula $\Box(db1xv_i \rightarrow \Box\neg dbx\_same\_version)$ states that if db_1 has x at version $v_i : 1 \leq i \leq 3$, then both replicas db_1 and db_2 will

never have item x at the same version. As expected this property is invalid and the model checker shows a witness path with cases where both replicas update x to the same version.

Thereafter, we can check whether item values for the same version in different databases are equal: $\Box(dbx\_same\_version \rightarrow dbx\_same\_value)$, where $dbx\_same\_version$ is defined as $db[0].x.current\_version = db[1].x.current\_version$ and dbx_same_value is defined as $db[0].x.current\_value = db[1].x.current\_value$.

TDB1 – Agreement: Whenever a transaction t has been decided, all replicated servers s_i have previously decided t with the same result.

The transaction result observed by the client must be the same decided by all replicas. In order to check this formula, the following statements must be valid: (i) if replica $db1$ decides to commit, then replica $db2$ also decides to commit; and (ii) if any replica decided to commit, then transaction's result must be *commit*. Before describing LTL formulas, we define the propositions $s_i\_commits\_t_j$ as $s_i.decided[j] = commit$, and $t1\_commit$ as $t1.result = commit$. Then we split property TDB1 in two LTL formulas:

TDB1(i): $\Box(\neg s1\_finishes\_t1 \rightarrow \Diamond s1\_commits\_t1) \rightarrow \Box(\neg s2\_finishes\_t1 \rightarrow \Diamond s2\_commits\_t1)$;

TDB1(ii): $\Box(s1\_commits\_t1 \rightarrow \Diamond t1\_commit)$

3.2.2. Isolation Level Properties

The set of properties presented so far focused mainly on verification of termination and consistency guarantees for the database replicas. However, transactions with conflicting operations would still incur in inconsistencies due to wrong data being manipulated by conflicting operations. Those inconsistencies may happen depending on isolation level provided by the concurrency management protocol [Adya et al. 2000, Kemme and Alonso 2000].

A conflict serializability isolation level requires a history to be conflict–equivalent to a serial history. Lower levels of isolation are less restrictive in terms of concurrency, but they may present inconsistencies. Berenson *et al.* [Berenson et al. 1995] presents a study of the isolation level semantics based on observations of some phenomena, like *dirty read*, *lost update*, *non repeatable read*, and *read/write skew*. Less restrictive isolation levels are susceptible to anomalies caused by some of those phenomena.

The verification scenarios discussed next were devised in order to validate the serializability isolation level provided by the deferred update replication protocol. The model was set up with 3 concurrent transactions. Transactions t_1 and t_2 are set up with conflicting operations that might lead to a specific phenomenon. Transaction t_3 can execute every possible sequence of 3 operations over x or y before requesting for commit or abort. Next we describe some phenomena and then verify if the deferred update protocol disallows those anomalous effects.

Non repeatable read: Transaction t_1 reads a data item. A transaction t_2 then modifies that data item and commits. If t_1 then attempts to reread the data item, it receives a modified value. Transactions are set up as $t_1 : r_1(x), w_1(y, 21), r_1(x), c_1$ and $t_2 : w_2(x, 12), r_2(y), w_2(y, 22), c_2$. A possible history that exemplifies this phenomenon

is $h : r_1(x)..w_2(x,12)..r_1(x)..$ (c_1 and c_2 occur in any order)[1].

In order to avoid this anomaly, t_1 must abort whenever it reads two different versions for a same data item. We check this behavior with an LTL formula in the form of: $\Diamond(t1rx\_v1 \wedge \Diamond t1rx\_v2) \rightarrow \Diamond t1abort$ where $t_krx\_v_j$ is true iff $\exists i \mid i \in t_k.rs \wedge i.item = x \wedge i.version = j$. The property holds, *i.e.* all histories in which a transaction reads two different versions for a same data item result in an abort decision.

Lost Update: Transaction t_1 reads a data item and then t_2 updates the data item. Based on its earlier read value, t_1 updates the data item and commits. The value updated by t_2 will be lost. Transactions are set up as $t_1 : r_1(x), w_1(x,11), w_1(y,21), c_1$ and $t_2 : w_2(x,12), r_2(y), r_2(x), c_2$. A possible history that exemplifies this phenomenon is $h : r_1(x)..w_2(x,12)..w_1(x,11)..c_1..c_2$.

By keeping updated items in the write set, each transaction isolates its local updates from potential updates being performed by other transactions until the transaction terminates. We verify this isolation property through the LTL formula $\Box(t2wx\_val12 \rightarrow \Diamond t2rx\_val12)$. The formula is true once $w_2(x,12)$ happens before $r_2(x)$ and values wrote by other transactions (e.g. $w_1(x,11)$ or any update from t_3) should not be visible by t_2 during its execution. Propositions $t2wx\_val12$ and $t2rx\_val12$ are given by $\exists i \mid i \in t_2.ws \wedge i.item = x \wedge i.value = 12$, and $\exists j \mid j \in t_2.rs \wedge j.item = x \wedge j.value = 12$.

Dirty read: Transaction t_1 modifies a data item. A transaction t_2 then reads that data item before t_1 finishes the transaction. If t_1 then aborts, t_2 has read a data item that had never really existed. Transactions are set up as $t_1 : w_1(x,11), r_1(y), a_1$ and $t_2 : r_2(y), r_2(x), r_2(x), c_2$. A possible history that exemplifies this phenomenon is $h : w_1(x,11)..r_2(x)..a_1..c_2$.

Once transactions keep their updated values locally, there is no way to interfere in the values read by other transactions. The updated values will just be perceived by others after the transaction commits. Considering transactions t_1, t_2, and t_3, we can check that t_2 will never read a value 11 for x describing the absence pattern in the formula: $\Box\neg(t2rx\_val11)$, where the proposition $t2rx\_val11$ is given by $\nexists i \mid i \in t_2.rs \wedge i.item = x \wedge i.value = 11$.

Write skew: Transaction t_1 reads x and y. Another transaction t_2 reads x and y, writes x, and commits. Then t_1 writes y. If there were a constraint between x and y, it might be violated. Transactions are set up as $t_1 : r_1(x), r_1(y), w_1(y,21), c_1$ and $t_2 : r_2(x), r_2(y), w_2(x,12), c_2$. A possible history that exemplifies this phenomenon is $h : r_1(x)..r_2(y)..w_2(x,12)..w_1(y,21)..$ (c_1 and c_2 occur in any order).

Once the deferred update replication protocol preserves serializability isolation level, it must avoid this anomaly by, for example, aborting one of the conflicting transactions. Whenever the operation $r_2(y)$ happens before c_1, and c_1 happens before t_2 request for commit, then servers s_1 and s_2 must, obligatorily, decide to abort t_2. This can be checked with the formula $(\Diamond(s1\_commits\_t1 \wedge \neg s1\_finishes\_t2 \wedge \neg s2\_finishes\_t2) \wedge ((\neg s1\_commits\_t1 \wedge \neg s2\_commits\_t1)\ U\ t2ry)) \rightarrow \Diamond t2abort$. The formula shows that if t_1 is committed in s_1, and t_2 not yet committed, and t_2 reads y before t_1 is committed,

[1]For the sake of simplicity, irrelevant operations for illustration of the phenomenon are omitted in h. The .. in the history suppresses any possible sequence of interleaved operations from t_1 and t_2.

then t_2 will abort.

Except by contradiction formulas (DB1(i) and DB2(i)), which were intentionally used to expose witness paths for desirable behaviors, all other specified formulas hold. That means the deferred update replication model satisfied the properties enunciated above. The model checker Spin was set up with partial order reduction and compression enabled. The experiments were performed in a computer with CPU of 6 cores and 2.66 GHz, and 32 GB of main memory. Table 1 exhibits the verification results.

Table 1. Resources allocated for Deferred Update Replication model

Formula	Stored States	Memory (MB)	Time
T1	50453993	3670.071	12 min
T2	60330993	4204.511	16 min
DB1(i)	3551454	350.263	33 sec
DB1(ii)	55748067	3959.893	13 min
DB1(iii)	68001328	4612.778	21 min
DB2(i)	47	129.106	1 sec
DB2(ii)	26440649	1898.649	4 min
TDB1(i)	53556032	3842.877	12 min
TDB1(ii)	32962609	2743.100	6 min
Non Repeatable Read	63838571	5032.491	13 min
Lost Update	40962043	3575.372	8 min
Dirty Read	20564107	1476.934	3 min
Write Skew	53473325	4162.453	8 min

4. Related Work

Most of the authors in literature justify correctness of replication protocols in a rather informal way without support of formal methods. Specially for the deferred update replication, most researchers sketch proofs of their designs in a natural language description [Kemme and Alonso 2000, Garcia et al. 2011, Luiz et al. 2011, Pedone and Schiper 2012]. However, due to the high concurrency level and inherent complexity of environments in which those protocols execute, a formal verification of those protocols is required.

Armedáriz-Iñigo *et al.* [Armendáriz-Iñigo et al. 2009] present a formal specification and correctness proof for replicated database systems. They carefully describe database replicas and the underlying replication protocol. Similar to our work, they check properties for atomic broadcast communication as well as database consistency provided by a certification-based protocol. A small difference between their and our work is that the protocol they analyzed assumes snapshot isolation level of consistency while the replication protocol we presented preserves serializability. They used I/O automata to describe system's behavior and verified system's properties through theorem proving.

Schmidt *et al.* [Schmidt and Pedone 2007] formally proved that a generic deferred update protocol preserves the serializability property. First they modeled a serial database and the termination phase of the protocol using TLA+. Then, by using refinement mapping, they proved that the states generated by the termination phase of the protocol are

equivalent to those generated by the serial database model. The authors described a deductive correctness proof, combining theorem proving and refinement mapping techniques.

In this paper we illustrated how useful model checking is for verification of data replication management protocols. Compared to [Schmidt and Pedone 2007] and [Armendáriz-Iñigo et al. 2009], the main advantages of this approach is the automatic verification and a very expressive description of properties by using of temporal logic. Moreover, due to its abstraction level, Promela models are closer to implementation and more familiar to distributed system developers if compared to I/O automata and TLA$^+$. This can help to better bridge the gaps between model and implementation.

5. Conclusion

This paper illustrated the use of model checking for database replication techniques. Specially, we recall the deferred update replication algorithm, that has been successfully adopted to increase availability with good performance. The experiments presented in this paper consolidate a first step on formal verification of deferred update replication protocol using model checking.

The use of Promela language provides a concise specification of the protocol in an algorithmic style. Properties are represented in LTL and their verification allowed us to observe the correct behavior expected by the protocol. In fact, temporal logic provided a very natural way to specify temporal relationships among operations executed throughout the complete history of the model computation. Section 3.2 demonstrated how to specify some common concurrency scenarios and how to check if undesirable phenomena affect the correctness of the protocol.

Another subject addressed by this paper is how to create models from smaller models previously verified. Modular approaches are common in distributed systems development, where specialized modules are coupled to a same system. For instance, a reliable system would be equipped with failure detectors, reliable channels (e.g. implementing atomic broadcast), or security components. Although we checked the uniform total order property in both, atomic broadcast and deferred update replication models, further development through compositional reasoning [Dotti et al. 2006] would provide a systematic approach to modular verification using building block models.

Extensions of the deferred update protocol tolerate crash or byzantine failure models [Luiz et al. 2011, Pedone and Schiper 2012]. In this direction, a next step for our work is to check protocol correctness under certain failure behaviors. It can be done by automatic insertion of failures to the model, followed by model checking [Dotti et al. 2005]. Fault injection combined to model checking is very attractive, since it is capable to represent non trivial faulty scenarios and disclosure misbehaviors hardly perceptible.

References

Adya, A., Liskov, B., and O'Neil, P. (2000). Generalized Isolation Level Definitions. In *Data Engineering, 2000. Proceedings. 16th International Conference on*.

Agrawal, D., Alonso, G., El Abbadi, A., and Stanoi, I. (1997). Exploiting atomic broadcast in replicated databases. In *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*.

Armendáriz-Iñigo, J. E., González, D. M., Garitagoitia, J. R., and Muñoz-escoí, Francesc, D. (2009). Correctness proof of a database replication protocol under the perspective of the I/O automaton model. *Acta Informatica*, 46(4).

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95. ACM.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4).

Dotti, F. L., Mendizabal, O. M., and dos Santos, O. M. (2005). Verifying Fault-Tolerant Distributed Systems Using Object-Based Graph Grammars. In *LADC*, volume 3747 of *Lecture Notes in Computer Science*. Springer.

Dotti, F. L., Ribeiro, L., Santos, O. M., and Pasini, F. (2006). Verifying object-based graph grammars. *Software & Systems Modeling*, 5.

Garcia, R., Rodrigues, R., and Preguiça, N. (2011). Efficient middleware for byzantine fault tolerant database replication. In *6th Conference on Computer systems*, EuroSys '11. ACM.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to fault-tolerant broadcasts and related problems. Technical report, Ithaca, NY, USA.

Holzmann, G. (1991). *Design and Validation of Computer Protocols*. Prentice Hall.

Holzmann, G. J. (1997). The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295.

Kemme, B. and Alonso, G. (2000). A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3).

Luiz, A., Lung, L. C., and Correia, M. (2011). Byzantine fault-tolerant transaction processing for replicated databases. In *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*.

Manna, Z. and Pnueli, A. (1991). Completing the temporal picture. *Theoretical Computer Science*, 83(1).

Pedone, F., Guerraoui, R., and Schiper, A. (1997). Transaction Reordering in Replicated Databases. In *16th IEEE Symposium on Reliable Distributed Systems*.

Pedone, F. and Schiper, N. (2012). Byzantine fault-tolerant deferred update replication. *Journal of the Brazilian Computer Society*, 18.

Salamah, S., Gates, A., Roach, S., and Mondragon, O. (2005). Verifying pattern-generated LTL formulas: A case study. In *Model Checking Software*, volume 3639 of *Lecture Notes in Computer Science*.

Schmidt, R. and Pedone, F. (2007). A formal analysis of the deferred update technique. In *11th International Conference on Principles of distributed systems*, OPODIS'07, Berlin, Heidelberg. Springer-Verlag.