Analysis and Verification of Two-Phase Commit & Three-Phase Commit Protocols

Muhammad Atif
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
Email: m.atif@tue.nl

***Abstract*—This paper introduces a formal model of the distributed commit protocols in the process algebra mCRL2 and also their general requirements in the modal μ-calculus. We show how to make straightforward models of protocols and by doing so, how it becomes easy to identify problems. We apply this to the well-known Two-Phase Commit Protocol (2PC) and prove it problematic for single site failure. We also apply our method to its "amended" variant, the Three-Phase Commit Protocol (3PC) and prove it to be erroneous for simultaneous site failures. We present 2PC and 3PC in different communication settings and verify them with respect to their requirements.**

I. Introduction

The purpose of this paper is to analyze and verify the correctness of commit protocols. Distributed database systems like airline reservation systems, banking applications, credit-card systems, stock-market transactions, widely use these protocols for their transactions over the network. So, undoubtedly, it is essential to verify their correctness. For this purpose, we have applied model checking techniques using the tool-set for mCRL2 [1]. The models of the commit protocols are given in [2] in two different communication settings: synchronous and asynchronous.

With respect to a distributed transaction, atomicity means that the commit update must be made on an all-or-none basis. That is, if the transaction gets committed, then all the participants exhibit its effects. However these effects are undone if the concerned transaction is aborted. A commit protocol is an algorithm to ensure the atomicity in a distributed transaction with the help of synchronized locking. Commit protocols are used in distributed database systems to show a well-defined behaviour of each and every node, particularly for commit or roll back transactions. Among commit protocols, Two-Phase Commit Protocol (2PC) [3] is said to be blocking because a transaction is blocked due to the coordinator's failure when the participant is in the ready-to-commit state. On the other hand, Three-Phase Commit Protocol (3PC) is said to be non-blocking [4], [5] but both of these protocols violate the important property of atomicity [6] at the time of multiple site failures for 3PC and single site failure for 2PC. In this paper, we have proved these properties with other functional requirements by writing them in the modal μ-calculus [7]. We apply the analysis and verification techniques to very simple models, i.e., the models having only two nodes, one of them is designated as coordinator and other as participant.

Structure of the paper. Section II and III introduce the 2PC and 3PC respectively. General requirements are stated in section IV. Section V presents model specifications, and experiments with their results are reported in section VI.

II. Two-Phase Commit Protocol (2PC)

A. General Description

The Two-Phase Commit Protocol (2PC) [8] is a distributed algorithm used in computer networks and distributed database systems. It is used when a simultaneous data update should be applied within a distributed database. In this protocol, one node acts as the coordinator, which is also called master and all the other nodes in the network are called participants or slaves. In its first phase, all these participants agree or disagree with the coordinator to commit, i.e., vote *yes's* or *no's* and in 2nd phase they complete the transaction simultaneously by getting the commit or the abort signal from the coordinator. This is also clear from Figure 1, where there is only one participant and one coordinator. The messages shown over the line are received messages and below the line are sent messages. Double circles show the final states.

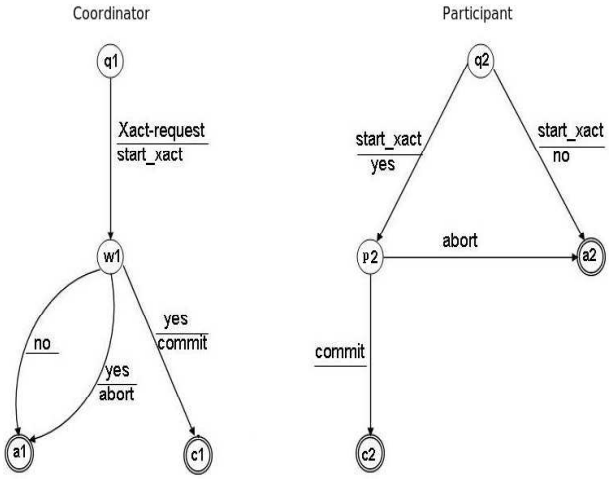

Figure 1. 2PC Protocol Actions [3]

978-1-4244-5633-8/09/$26.00 ©2009 IEEE

Figure 1 shows that coordinator receives the $xact\_request$, i.e., transaction request and forwards it to the participant. The participant either partially executes it and sends *yes* to goto pre-commit state (p2) or sends *no* to goto aborted state (a2). If the coordinator receives *yes*, then non-deterministically it can send *commit* or *abort* and depending upon this message, the participant reaches to its final state either by committing or aborting the current transaction. A participant can have three possible distinct states:

1) *Initial:* When it is ready to receive the transaction, shown by "q2" in Figure 1.
2) *Uncertain:* When it has voted in *yes* but doesn't know its result while holding locks on resources. It is also called the pre-commit state (p2) as shown in Figure 1.
3) *Decided:* When it knows about the *commit* or *abort* decision (denoted by $c2$ and $a2$, respectively), i.e., it leaves the pre-commit state.

Global commit or abort means all participants must *commit* or *abort*, even if there is failure or timeout at any one of the nodes. Timeout means the failure of the other site. The coordinator plays the central role and flags either global commit or global abort. The former is only shown if all the participants vote to *commit* and the latter is shown if at least one of the participants votes to abort or the coordinator decides to abort the current transaction.
In case there is no failure at any site, the protocol is correct but it is highly desirable to consider the functionality in the presence of failure of any site at any state.

B. Problems with 2PC

There are two problems with the above-described Two-Phase Commit Protocol.

1) *Blocking:* The Two-Phase Commit Protocol goes to a blocking state by the failure of the coordinator when the participants are in uncertain state. The participants keep locks on resources until they receive the next message from the coordinator after its recovery.
2) *State Inconsistency:* Global state vector in commit protocols works as a container of states for every participating node regarding a single transaction. The global transition state comprises this state vector and outstanding messages in the network. Dale Skeen and Michael Stonebraker [3] call a state *inconsistent* when its global state vector contains both the commit and abort states. This inconsistency can be observed using a state vector, particularly when the participant is at its pre-commit state (p2) and fails. The coordinator shows the committed state after sending *commit* message but for the failed participant the protocol is declared non-resilient in [3] for assigning new state .

III. The Three-Phase Commit Protocol (3PC)

A. General Description

Three-Phase Commit Protocol (3PC) is a non-blocking protocol, contrary to the 2PC. Here a new state called "pre-commit" is introduced for the coordinator in [3]. The coordinator gets to this "pre-commit" state only if all other participants have voted to commit, i.e., *yes*. In case this state is not reached, the participant will abort and release the blocked resources after a specific time. When the coordinator gets the "pre-commit" state then there is only one option to abort the transaction and that is a timeout, which corresponds to a failure of a participant, otherwise the transaction gets completed with an acknowledgement from the participants. It is also possible that the coordinator fails at this state, even then it will proceed for global commit as shown in Figure 2.

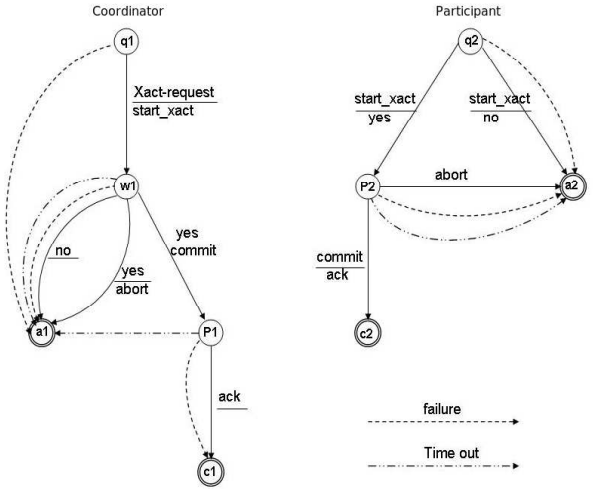

Figure 2. 3PC with failure and timeout transitions [3]

B. Problems with 3PC

Three-Phase Commit Protocol is problematic only when there are multiple sites failures (proved in section VI-B). For example, let's consider a case where the coordinator is in "pre-commit" state and fails just after sending a *commit* message and the slave also fails just before or after receiving this message as shown in Figure 5. So by its failure, the slave moves to the aborted state but according to the protocol specifications given in [3], the coordinator goes to the committed state, either it fails or receives acknowledgement. Hence, the coordinator moves to the committed state without receiving acknowledgement and the failed slave moves to the aborted state without sending the acknowledgement. In this way, coordinator and participant show different final states due to their failures.

IV. General Requirements

The general requirements of a commit protocol are presented below which have also been specified in the modal μ-calculus in section V-B1.

R1. The coordinator aborts the transaction, if at least one participant votes to abort.
R2. The coordinator commits a transaction, only if all the participants vote to commit.

R3. All non-faulty participants including the coordinator should eventually decide to abort or commit.

R4. If any one of the participants (including the coordinator) decides to abort (or commit), then no other participant will decide to commit (or abort).

V. Tools for Specifications

A. mCRL2

mCRL2 [1] (micro Common Representation Language 2) is a formal specification language based on process algebra [9] for modeling, validation and verification of concurrent systems and protocols [10]. It is a language for behavioural description that includes data and time. We used it due to the available expertise and the fact that this tool-set has already been applied successfully for the behavioural analysis of various protocols [10], [11] and distributed systems [12]. Its accompanying tool-set supports different tools, which are used for linearization (a simplified form of process suitable for analysis and state space generation), simulation, reduction and state-space generation (for visualization and analysis). The mCRL2 language is the successor of μCRL [13]. The models of both 2PC and 3PC in different communication settings have been specified in mCRL2 syntax. Using this syntax, we can declare actions (parametrized and non parametrized), data types and processes. The reserved words $proc$, $sort$ and act are used for defining processes, actions and data types respectively. Whereas, with the help of certain operators, we develop models. For example '.' is an operator for sequential composition, so $p.q$ means first p then q where p and q are two arbitrary actions or processes. In the same way '+' is a alternative composition operator, used as $p + q$ that means either p or q. To generalize it, sum operator is applied, e.g., $\sum_{s=S} p(s)$, i.e., p(s_0)+p(s_1)+.....) where S is some $sort$ and S=$\{s_0,s_1,.....\}$. For the parallel execution of processes or actions, a parallel operator '||' is denoted by $p||q$. It means the actions of p and q will happen concurrently and/or in an interleaved fashion. Interleaving means the arrangement of actions in non-contiguous fashion of all parallel processes. For example, if $p = p1.p2$ and $q = q1.q2$ are two parallel processes then there can be multiple adjustments of actions but in every trace, the second action of each process will never occur before its first one. We can also declare the communication of actions like $comm\ a|b \rightarrow c$, which means that c is the result of synchronization between a and b.

1) Specification Settings for 2PC and 3PC: In this paper, the following approaches of communication between the coordinator (herein called master) and a participant (herein called slave) are presented.

1) Synchronous Communication.
2) Asynchronous Communication through reliable channels.

Processes for each component like master, slave and channels are separately defined and their execution starts in parallel as shown in the "init" part of the specification. As specific messages are exchanged between master and slave so for such messages, enumerated data types (denoted by keyword $struct$) M2S (master to slave) and S2M (slave to master) are introduced as given below.

$$sort$$
$$M2S = struct\ start\_xact \mid abort \mid commit \mid failure\_M;$$
$$S2M = struct\ yes \mid no \mid ack \mid failure\_S;$$

Here $start\_xact$ from master to slave means forwarding of the transaction and in its result, slave either sends *yes* or *no*. In the same way, the messages *commit* and ack (acknowledgement) are exchanged for the completion of the transaction. The message $failure\_S$ is for the slave failure which acts as timeout for the master and $failure\_M$ is for master failure that is timeout for slave.

2) Processes: Although in this paper, we have defined a number of processes for different fashions of communications for both 2PC and 3PC, in this section we discuss only 3PC with synchronous communication scheme, because its initial actions are similar to 2PC just before the pre-commit state. The slave side in 3PC also matches 2PC besides one additional message, i.e., acknowledgement for master as shown in Figures 1 and 2. We start describing the processes first with synchronous and then asynchronous communications.

Process for Master (Synchronous): In synchronous communication, master process starts with a non-deterministic choice between receiving the transaction or its failure. Figure

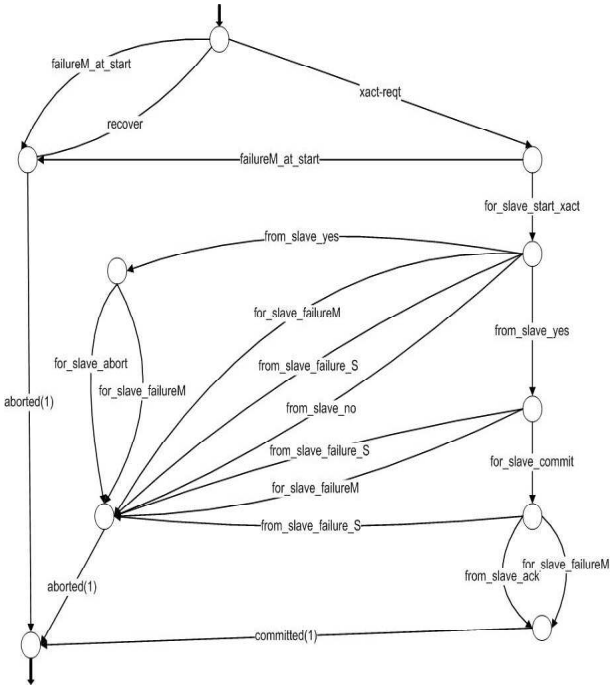

Figure 3. Transition system for Master in 3PC

3 shows the same in a transition system. There are two types of failures: one is $failureM\_at\_start$ and other is $for\_slave\_failureM$. They are treated separately because the failure of master will effect the slave only when they are both coupled for a specific transaction or in other words, the

slave receives the transaction as first message. So the former failure is before forwarding the transaction while the latter is after that, which causes the timeout for slave. In other words we can say that failure of one site will cause timeout only when the other is waiting for next message. Let's consider the path to the committed state as a happy scenario. So, the unique trace for it is:
$xact\_request.for\_slave\_start\_xact.from\_slave\_yes.$
$for\_slave\_commit.\ from\_slave\_ack/for\_slave\_failure.$
Any other choice at any state leads to the aborted state.

Process for Slave (Synchronous): The slave process is executed in parallel with the master process and synchronizes with master according to the protocol description. Figure 4 shows the transition system for the slave process that terminates either by committing or aborting the transaction, exactly like the master process. It is also clear from Figure 4 that failure of slave can occur at any state.

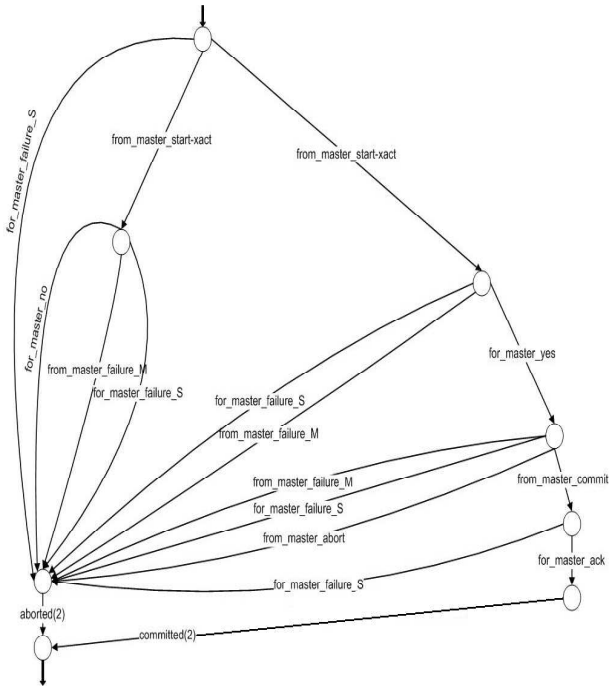

Figure 4. Transition system for Slave in 3PC

Processes for Asynchronized Communication: In the asynchronized communication, the processes of both master and slave use two different channels for sending and receiving messages from either side. The channel for transferring messages from master to slave is Channel_M2S and for the other way round is Channel_S2M. The following processes define these channels.

$proc$

$$Channel\_M2S = \sum_{message=M2S} rcv\_from\_master(message).$$
$$send\_to\_slave(message).Channel\_M2S;$$

$$Channel\_S2M = \sum_{message=S2M} rcv\_from\_slave(message).$$
$$send\_to\_master(message).Channel\_S2M;$$

So due to these channels in asynchronized communication both master and slave processes use parametrized actions as given in [2]. These channels are applied by putting their processes in parallel with the processes of master and slave as:
Master || Slave ||Channel_M2S || Channel_S2M.

B. μ-calculus

Modal μ-calculus of Kozen [7] is used for expressing the general requirements of a commit protocol. It has been extended with regular formulae, multiactions, action formulae and data-dependent processes. It uses '*' for cardinality that means occurring (of actions) any number of times and '!' is used for negation. The modalities "[]" and "<>" followed by a formula are used for necessarily and possibly respectively. Necessarily means the formula holds for every action enclosed within brackets while the possibly means that there exists an action (enclosed in "<>") where the formula is satisfied. For instance consider the following formulae, expressing few requirements in the specific syntax. Here ν is for the greatest or maximal fixed point while the μ is used for the least or minimal fixed point.

- $[true*] < true > true$
  It means that after any action occurring any number of times,an action is possible, in short there is no deadlock.
- $[a]!$ <b> $true$
  After every a action, b cannot occur or the trace $a.b$ is not possible but the formula
  $[true * .a.true * .b]false$
  means that after any a action, there can be any other actions except b.
- $< true* > \nu X < a > X$
  This formula expresses that an infinite traces of $a's$ should be possible at any state.
- $[!a*][b]false$
  The formula says that action b is only possible if the action a has also occurred before somewhere.
- $[true * .a]\ \mu X.([!b]X\ \&\&\ < true > true)$
  This formula means that every trace containing action a must have action b eventually.
- $[true * .a.true * .b.true*]false$
  It means that occurring of action b anywhere after the action a will result in false, i.e., each trace that contains action a, should not contain action b.
- $[true*][a + b][true*][c]\ false$
  It means the action c cannot occur after actions a or b.

1) Requirement Specification in μ-calculus: The purpose of specifying the requirements in μ-calculus, is their verification using the model checking tool-set which is part of mCRL2. The following is the description of each requirement with its respective formula. The results of the verification of each formula are presented in section VI-B. As mentioned earlier, we have modelled the protocols with respect to one master

and one slave, so in each of the following formulae argument '1' means the master and '2' means the slave.

R1: This requirement is for "atomicity" when the slave is not ready to execute the transaction and therefore it sends *no* to the coordinator. In this case it is essential that both of the nodes should eventually end in the same state and its formula is:

$[true * .sent\_by\_slave(no)]$
$\mu X.([!aborted(2)\&\&!aborted(1)]X \;\&\&\; < true > true)$
$\&\&$
$[true * .rcv\_by\_master(no)]$
$\mu X.([!aborted(2)\&\&!aborted(1)]X \;\&\&\; < true > true)$

R2: In the commit protocols, coordinator issues the *commit* command only if the participants are ready to complete the partially executed transaction of the first phase. In both 2PC and 3PC, second phase starts by issuing of this command. In other words, the *commit* command can only be issued after receiving *yes* from the participant(s). So its formula is:

$[!rcv\_by\_master(yes)*]$
$[committed(1) + committed(2)]false$

R3: Obtaining the final state for every node is an essential requirement. The transition systems given in Figures 3 and 4 show that the terminating states of both master and slave are "committed" and "aborted". So reaching any one of them is the requirement and it is presented by the following formula.

$\mu X.([!aborted(1) \;\&\&\; !committed(1)]X \;\&\&\; < true > \; true)$
$\&\&$
$[!failureM\_at\_start*]$
$\mu X.([!aborted(2) \;\&\&\; !committed(2)]X \;\&\&\; < true > true)$

R4: Atomic commit is also one of the basic requirements for all the commit protocols, i.e., the final states must be the same of all the nodes.

$[!failureM\_at\_start * .committed(2).true.*$
$aborted(1).true*]false$
$\&\&$
$[!failureM\_at\_start * .committed(1).true * .$
$aborted(2).true*]false$

In both R3 and R4, we have considered only the traces which are without $failureM\_at\_start$, i.e., the failure of master just before or after getting the $start\_xact$ because it doesn't cause any timeout for slave. But, as the slave receives the transaction then failure of any site will definitely effect the other as explained in section V-A2.

VI. Verification

A. Verification Techniques

THEY MODEL CHECKED

After modeling the commit protocols and writing their requirements in modal μ-calculus, we verify every formula with the help of the different tools like:

1) *mcrl22lps:* For simplification and easy manipulation, every process is linearized. Linearizing means converting systems of recursive equations into linear equations. It is a simple format used for generating transition system, simulation and checking certain properties.
Syntax: mcrl22lps [OPTION]... [Input_File [Output_File]]
2) *lps2pbes:* To verify modal μ-calculus formulae on transition systems, boolean equation systems are used. $lps2pbes$ is applied to convert the LPS (output of step 1) to a Parametrized Boolean Equation System (PBES) [14] regarding the μ-calculus state formula as given in section V-B1.
Syntax: lps2pbes [OPTION]... -f File [Input_File [Output_File]]
Here File contains μ-calculus formula.
3) *pbes2bool:* To check the validity of a pbes, this tool translates it into a Boolean Equation System (BES) and shows the output either "The pbes is valid" or "The pbes is not valid".
Syntax: pbes2bool Input_File
We can also get the counter example (using optional switches) to identify such traces where the formula becomes invalid.
4) *lps2lts:* This tool helps to generate the state space from the linear process specifications (LPS).
Syntax: lps2lts [OPTION]... [Input_File [Output_File]]
5) *ltsgraph:* This tool is used to view the graph of state space (as shown in Figure 5) by which it is easy to locate the trace(s) for counter example. For example, in the analysis of commit protocols, we found the inconsistency and its different paths with the help of this tool.
Syntax: ltsgraph [OPTION]... [Input_File]

B. Verification Results

	2PC			3PC	
	Sync	Async	Sync WF	Async	Sync WF
R1	T	T	T	T	T
R2	T	T	T	T	T
R3	T	T	T	T	T
R4	T	F	F	F	T

Here "Sync" stands for synchronous and "Asyc" is for asynchronous communication where the "WF" means "with failure" transitions.

- **R1:**
The formula R1 is satisfied by every model because if the slave sends *no* and afterwards there is no deadlock, then both the coordinator and the slave are supposed to end in the aborted state. The transitions systems of slave and master shown in Figure 3 and 4 illustrate the trace to aborted state in effect of *no*.
- **R2:**
The formula R2 is satisfied by every model because without the willingness of the slave, both the master and the slave cannot end in the committed state. This formula uses receiving *yes* instead of sending *yes* because master can send *commit* only after receiving *yes* message.
- **R3:**

This requirement is for non-faulty nodes, i.e., which are without failure transition, so R3 is satisfied in case of 2PC without failures. This also proves that 2PC works fine in the absence of failure(s) because all the requirements are fulfilled. The other models of 2PC and 3PC also satisfy this requirement, so every node reaches to its final state.

- **R4:**
  It is observed during the verification process that neither 2PC nor 3PC satisfy this property specifically in case of site(s) failure. While investigating, we found that in 2PC this happens when the participant fails after receiving the *commit* signal and shows aborted state, whereas the coordinator has moved to the committed state by sending *commit*. In 3PC, we observed the inconsistency only on

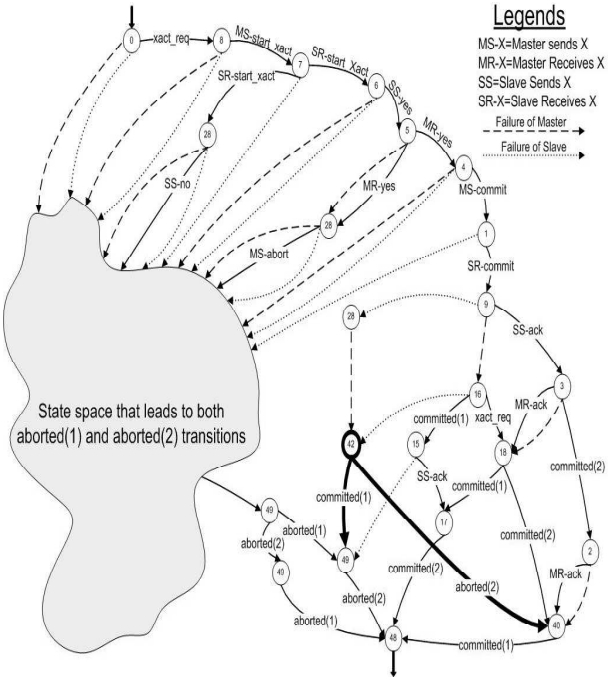

Figure 5. State Space of 3PC Asynchronized

the paths which contain failures of both master and slave as shown by bold lines in Figure 5. But R4 is satisfied in synchronized communication in 3PC because failure of one site is received as timeout message by the other, i.e., other site does not fail.

VII. Literature Review

Commit protocols (2pc and 3pc) are used in transaction processing, databases and computer networking [15], [16], [8]. Correctness of the distributed systems or protocols, can be verified with respect to their requirements by applying formal methods, i.e., model-based verification techniques . During the last two decades, research in applying formal methods has led to development of certain software tools and techniques [17], one example is mCRL2 [1]. Using mCRL2, first the protocols with their requirements are specified formally and then the automatic verification process is carried out. In this way, a requirement can either be satisfied or violated. In case of violation, the respective counter example helps to figure out the unexpected behaviour.

VIII. Conclusions

We observed during the modeling and verification that mCRL2 is well suited to evaluate the expected and unexpected behaviour of interactive systems. Experiments and their results show that tool-set for mCRL2 is capable enough for use in successful analysis of distributed algorithms. We have formalized and analyzed the Two-Phase Commit & Three-Phase Commit protocols in different settings with the help of mCRL2. We have also described the requirements and formulated them in modal μ-calculus for verification. We found and discussed the traces where a very important requirement, i.e., "consistency of final states" is violated. The critical task in this analysis was writing μ-calculus formulae because the expertise in it comes at a price. So adaptability of process algebra generally and μ-calculus particularly is one of the open problems.

References

[1] J. F. Groote, A. Mathijssen, M. van Weerdenburg, and Y. S. Usenko, "From μCRL to mCRL2: motivation and outline," *Electr. Notes Theor. Comput. Sci.*, vol. 162, pp. 191–196, 2006.

[2] M. Atif. mCRL2 code for two-phase and three-phase commit protocols. [Online]. Available: http://www.win.tue.nl/~atif/docs/2pc3pc.zip

[3] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," *IEEE Trans. Software Eng.*, vol. 9, no. 3, pp. 219–228, 1983.

[4] D. Skeen, "Nonblocking commit protocols," in *SIGMOD Conference*, Y. E. Lien, Ed. ACM Press, 1981, pp. 133–142.

[5] ——, "A quorum-based commit protocol," in *Berkeley Workshop*, 1982, pp. 69–80.

[6] T. Härder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, 1983.

[7] D. Kozen, "Results on the propositional mu-calculus," *Theor. Comput. Sci.*, vol. 27, pp. 333–354, 1983.

[8] J. Gray, "Notes on data base operating systems," in *Advanced Course: Operating Systems*, ser. Lecture Notes in Computer Science, M. J. Flynn, J. Gray, A. K. Jones, K. Lagally, H. Opderbeck, G. J. Popek, B. Randell, J. H. Saltzer, and H.-R. Wiehle, Eds., vol. 60. Springer, 1978, pp. 393–481.

[9] W. Fokkink, *Introduction to Process Algebra*. Secaucus, NJ, USA: Springer, 2000.

[10] W. Fokkink, J. F. Groote, J. Pang, B. Badban, and J. van de Pol, "Verifying a sliding window protocol in mCRL," in *AMAST*, ser. Lecture Notes in Computer Science, C. Rattray, S. Maharaj, and C. Shankland, Eds., vol. 3116. Springer, 2004, pp. 148–163.

[11] J. van de Pol and M. V. Espada, "Verification of javaspaces^{tm} parallel programs," in *ACSD*. IEEE Computer Society, 2003, pp. 196–205.

[12] J. F. Groote, J. Pang, and A. G. Wouters, "Analysis of a distributed system for lifting trucks," *J. Log. Algebr. Program.*, vol. 55, no. 1-2, pp. 21–56, 2003.

[13] S. Blom, W. Fokkink, J. F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol, "μCRL: A toolset for analysing algebraic specifications," in *CAV*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Springer, 2001, pp. 250–254.

[14] J. F. Groote and T. A. C. Willemse, "Parameterised boolean equation systems," *Theor. Comput. Sci.*, vol. 343, no. 3, pp. 332–369, 2005.

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

[16] G. Weikum and G. Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[17] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.