ID2203 Distributed System, Advanced Course Course Project - TLA+ and PlusCal Report

Weiran Wang
ICT Innovation - Cloud and Network Infrastructures
KTH Royal Institute of Technology
weiranw@kth.se

I. Project Introduction

There are two main tasks, i.e. Fail-Stop Leader Election + Crash Fault-tolerance and Abortable Paxos, in this project by focusing on specifications, properties and model checking using PlusCal, TLA+ and the TLC model checker.

In general, PlusCal is the main algorithm language in this project, which gives a pseudo code-like interface to ease users' understanding. After finishing, saving and translating the PlusCal, it can be automatically translated to TLA+, which is a formal specification language and can define the properties of what users want instead of how to find it.

Finally, the TLC Model Checker is a program for finding errors in TLA+ specification and explore all possible states, which help users debug during the programming.

II. Leader Election

A. Fail-Stop Leader Election

Based on the given skeleton code in LeaderElection.tla, only three steps, corresponding to the three labels in the skeleton code, are needed in this task.

1) **P:** A process's value with self can be got in this label.
2) **Bcast:** Each process broadcasts their own id (self) and waits until al processes have completed this action.
3) **Sync:** Each process picks the highest id as the leader. Specifically speaking, there exists the possibility that some processes are implemented faster than others. In this case, processes should wait each other till all of them select a leader so that they can go to the next round together. Furthermore, considering there is no crashes in this section, a selected leader will exist forever. Consequently, there is no need to wait till the last round of the algorithm. In this way, when all processes have selected a leader, we can then terminate the Leader Election algorithm.

Two properties of Fail-Stop Leader Election should be satisfied, which are:

1) **Agreement:** All processes elect the same leader. With the help of sequence "elected", it can be defined that a process, e.g. pid1 or pid2 has selected a leader and then compare whether these two processes select the same leader. This property is put as an invariant.
2) **Completeness:** Eventually, all processes elect some leader. "<>" reflects the keyword "eventually" and should be put as a property in Model Overview as introduced in the project tutorial video. In this way, it is easy to understand as long as all the elements in consequence "leader" are not 0.

In TLA+, the code is written in this way:

```
\* Agreement: All processes elect the same leader.
Agreement == \A pid1, pid2 \in Processes: (elected[pid1] = TRUE /\ elected[pid2] = TRUE)
          => leader[pid1] = leader[pid2]
\* Completeness: Eventually, all processes elect some leader.
Completeness == <> (\A pid \in Processes: leader[pid] # 0)
```

Fig. 1. Fail-Stop Leader Election Properties.

In the Model Overview, the values of declared constants are specified as: $N \leftarrow 3$, $Stop \leftarrow 5$) and the results are shown in figure 2 and 3.

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:00:07	94	101,939	24,628	0
00:00:01	0	1	1	1

Fig. 2. Fail-Stop Leader Election State Space Progress.

Sub-actions of next-state (at 00:00:07)

Module	Action	Location	States Found	Distinct States
LeaderElection	Terminating	line 127, col 1 to line 127, col 11	1	0
LeaderElection	Init	line 84, col 1 to line 84, col 4	1	1
LeaderElection	Sync	line 113, col 1 to line 113, col 10	6,246	2,059
LeaderElection	P	line 93, col 1 to line 93, col 7	7,367	3,021
LeaderElection	Bcast	line 101, col 1 to line 101, col 11	88,334	19,547

Fig. 3. Fail-Stop Leader Election Sub-actions of next state

B. Crash Fault-tolerance

In this task, $f = F$ - the number of processes left to be crashed, and $alive = [p \backslash in Processes| \rightarrow TRUE]$ - whether

a process is alive are added to indicate the crash process. Two properties should be satisfied as well here.

1) **Agreement:** All **correct** processes elect the same correct leader. Similar to the previous one, if two processes have a elected leader, their leader should be the same. In addition, both of these processes should be alive as the prerequisite. Furthermore, according to the hint given by Harald, the property can be changed to ”eventually” to fulfill the requirement.
2) **Completeness:** Eventually, all **correct** processes elect some correct leader, which indicates if a process is correct, its leader should also be correct.

In the PlusCal code, they are implemented like this.

```
\* Agreement: All correct processes elect the same correct leader.
Agreement == <> (\A pid1, pid2 \in Processes: (alive[pid1] /\ alive[pid2]) /\ (elected[pid1] /\ elected[pid2])
             => leader[pid1] = leader[pid2])

\* Completeness: Eventually, all correct processes elect some correct leader
Completeness == <>(\A pid \in Processes: leader[pid] # 0 /\ alive[pid] => (alive[leader[pid]] = TRUE))
```

Fig. 4. Crash Fault-tolerance Properties

1) **Question:** Why do we not allow crashes at the last round?
   Solution: Suppose one possibility that a process which has a potential to be a leader, but, unfortunately, crashes at the last round after it has been elected as a leader. In this case, the leader election becomes invalid. In this case, both Agreement and Completeness are violated. Specifically speaking, the leader which is selected is not correct.
2) **Question:** How does the properties of the previous task get violated when processes can crash?
   Solution: When a process crash before it selects a leader, it will violate property **Completeness** that eventually, all processes elect some leader, since this crashed process hasn’t elected a leader yet and will not elect a leader in the future in this fail-stop model. Moreover, it violates the property **Agreement** as well, since **Agreement** asks to all processes elect the same leader. However this process doesn’t have a leader.

The running result is shown as figure 5 and 6.

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:01:13	76	7,199,357	2,028,912	0
00:01:04	50	6,604,816	1,879,963	41,517
00:00:04	11	62,394	17,932	6,438
00:00:01	0	1	1	1

Fig. 5. Crash Fault-tolerance State space progress

Sub-actions of next-state (at 00:01:13)

Module	Action	Location	States Found	Distinct States
LeaderElectionCrash	Terminating	line 129, col 1 to line 129, col 11	79	0
LeaderElectionCrash	Init	line 76, col 1 to line 76, col 4	1	1
LeaderElectionCrash	P	line 87, col 1 to line 87, col 7	1,479,321	337,514
LeaderElectionCrash	Sync	line 115, col 1 to line 115, col 10	1,467,804	501,696
LeaderElectionCrash	Bcast	line 95, col 1 to line 95, col 11	4,252,152	1,189,701

Fig. 6. Crash Fault-tolerance Sub-actions of next state

III. Abortable Paxos

There are three roles in Paxoes, i.e., Processor, Acceptor and Learner. However, according to the project guideline, the Acceptor processes also have the role of Learners.

1) **Processor:** Wants a particular proposed value to be decided. In the typical majority quorum setup, a proposer tries to get a majority of acceptors to accept its proposal.
2) **Acceptor:** Acknowledges acceptance of proposed values.
3) **Learner:** Decides based on acceptance of values. If a proposal has a majority of acceptors, then it is called chosen and the learners will decide it.

The algorithm designing mainly focuses on three states in the algorithms as well, i.e., Propose, Promise and Learn.

1) **Propose State (P1L):** Proposers will generate the ballot send the Prepare request to acceptors. Then acceptors will send the promising message (ReplyPrepare(ba) in A) according to the Prepare message they receive. It is worth mentioning in this phase that when acceptors receive the Prepare request, they will make the following two promises: 1. They will not receive the ballot which is smaller than the current Prepare request’s. 2. Without violating the promises made before, acceptors will reply to the value and ballot of the proposal with the largest ballot that has been accepted.
2) **Accept State (CP1L & P2L):** After the proposer has received a majority of promise messages from the acceptors’ side, the proposer will send Accept request to acceptors. After that, acceptors will deal with the Accept messages (ReplyAccept(ba) in A).
3) **Learn State (CP2L & P3L):** After receiving a majority number of ReplyAccept messages from the acceptors, the proposer will send the formed ballot and value to all accptors / learners, which indicates the success of this ballot. Consequently, all acceptors / learners will receive the decision (RcvDecide(ba) in A).

The following properties hold:

1) **UC1 (Validity):** Only proposed values may be decided. It can be checked in Acceptor’s messages. If there is a message type ”decided”, there must exists a message type ”accept” with the same decided value.

2) **UC2 (Uniform Agreement):** No two processes decide different values. In other words, the Uniform Agreement property will be violated is the acceptors have two different value at different time slot. Based on this understanding, the elements in decided sequence can be checked and the decided sequence should not be empty as the prerequisite.
3) **UC3 (Integrity):** Each process can decide a value at most once. Considering the number should be one, the keyword "Cardinality" in TLA+ can play its role here. As long as Cardinality(decided[x][y]) = 1, then this property is fulfilled.
4) **UC4 (Termination):** Every correct process eventually decides a value. Similar with the "Completeness" property in Leader Election algorithm, "<>" can be used here to describe the keyword "eventually". Considering the initialization of sequence "decided" is set to empty, the final state of "decided" not being empty will fulfill this property.

In the code, they are written in this way (The termination is put in properties while others are put in invariants):

```
\* UC1 (Validity) Only proposed values may be decided
Validity == \A i \in AccMsg: i.type = "decide"
    => \E msg \in AccMsg: msg.type = "accept" /\ i.slot = msg.slot /\ i.dcd = msg.val[3]
\* UC2 (Uniform Agreement) No two processes decide different values
Uniform_Agreement == \A pid1, pid2 \in Acceptor: \A i \in Slots:
                     (decided[pid1][i] # {} /\ decided[pid2][i] # {})
                     => decided[pid1][i] = decided[pid2][i]
\* UC3 (Integrity) Each process can decide a value at most once
Integrity == \A pid1, pid2 \in Acceptor:
             \A i \in Slots: (decided[pid1][i] # {} /\ decided[pid2][i] # {})
             => Cardinality(decided[pid1][i]) = 1 /\ Cardinality(decided[pid2][i]) = 1
\* UC4 (Termination) Every correct process eventually decides a value
Termination == <>[] \A pid \in Acceptor: \A i \in Slots: decided[pid][i] # {}
```

Fig. 7. Abortable Paxos Properties

According to the comment in the skeleton code, I set $M \leftarrow 2$, $N \leftarrow 3$, $MAXB \leftarrow 2$ and $STOP \leftarrow 2$. Consequently, the running results are shown as followed.

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:00:13	33	740,051	57,096	0
00:00:05	24	135,075	18,144	6,456
00:00:02	0	1	1	1

Fig. 8. Abortable Paxos State space progress

Sub-actions of next-state (at 00:00:13)

Module	Action	Location	States Found	Distinct States
AbortablePaxos	Init	line 171, col 1 to line 171, col 4	1	1
AbortablePaxos	CP1L	line 230, col 1 to line 230, col 10	16	6
AbortablePaxos	P2L	line 242, col 1 to line 242, col 9	528	309
AbortablePaxos	CP2L	line 250, col 1 to line 250, col 10	4,752	906
AbortablePaxos	P3L	line 263, col 1 to line 263, col 9	2,112	1,261
AbortablePaxos	A	line 188, col 1 to line 188, col 7	656,004	8,580
AbortablePaxos	P1L	line 217, col 1 to line 217, col 9	14,806	11,097
AbortablePaxos	L	line 210, col 1 to line 210, col 7	31,174	14,646
AbortablePaxos	END	line 273, col 1 to line 273, col 9	30,658	20,290

Fig. 9. Abortable Paxos Sub-actions of the next state