ID2203 –Distributed Systems, Advanced
Course Project – TLA+ and PlusCal
VT21 P3

TA – Harald Ng <hng@kth.se>

1 Organisation

This project focuses on writing specifications, properties and model checking using PlusCal, TLA+ and the TLC model checker. There are 3 tasks with increasing difficulty and it is therefore strongly recommended to do the tasks in order. The tasks are not directly related to each other but contain parts that could be reused later.
Hand in both the report (as .pdf) and the source code (as .zip or .tar.gz) of your project in Canvas. Do not package up the report with the code, but submit them as separate files! **Suffix the filename of your report with "_tla"**, so we can easily distinguish them.

Furthermore, it is strongly recommended to use GIT for managing your source code. You are free to use public Github or KTH's gits[1] repository. If you do use either, please provide a link to the repository in your report. (But still submit the code in Canvas anyway, to have a reference snapshot for grading.)

1.1 Dates

Preliminary Report 23 February in Canvas

Preliminary Report Peer Feedback Session 26 February Lab exercise

Final Report 12 March in Canvas

1.1.1 Preliminary Report

For the preliminary report, go through the template code and describe what you understand and not understand. During the peer-review session, you will have an opportunity to discuss and learn from each other. Furthermore, if you plan to do task 3, describe ideas of how you plan to implement it. This could be how to model message-passing, process failures etc.

Do not include any of your solutions in the preliminary report!

[1] https://gits-15.sys.kth.se/

1.2 Goals

The goal of the project is to implement distributed algorithms using PlusCal, define their properties with TLA+ and check that the implementation achieves the properties using the TLC model checker. To get going, you will be given parts of the code in PlusCal in the first two tasks.

1.3 Requirements

For this project you will need the TLA+ Toolbox[2]. All the skeleton code and other useful resources can be found at:

- https://gits-15.sys.kth.se/hng/id2203_tlaproject21

Please read the README before starting. It contains information of how to get started and learning PlusCal and TLA+.

1.4 Time

Be sure to plan enough time for the project. PlusCal and TLA+ are not very verbose but require a different mindset and syntax from programming languages. You are encouraged to look at examples to learn techniques and get more familiar with the mathematical notions.

2 Tasks

2.1 Fail-Stop Leader Election (10 Points)

In this task, a Leader Election algorithm in the Fail-Stop model is to be implemented. In every round, each process broadcasts their own id (`self`) and waits until all processes have completed this action. A process signals that it has completed the action by going to the next round. At this point, each process picks the highest id as the leader.

2.1.1 Getting Started

Create a new specification in TLA+ Toolbox called `LeaderElection` and paste the given skeleton code into `LeaderElection.tla`. Fill in the missing PlusCal code according to the comments and use the automatic translator in the ToolBox to translate the PlusCal code into TLA+.
The properties of our leader election algorithm are the following:

- **Agreement**: All processes elect the same leader.
- **Completeness**: Eventually, all processes elect some leader.

[2] https://lamport.azurewebsites.net/tla/toolbox.html

Define these properties in TLA+. Note that they must be placed after the automatically generated TLA+ code (i.e. after the comment `\* END TRANSLATION`).
Create a model and add the properties to it before running the model checker. Were your properties defined correctly? Try changing the algorithm to deliberately violate a property and let the model checker find it. Write about your findings in the report.

2.1.2 Crash Fault-tolerance

At this point, our leader election algorithm does not handle crashed processes. Create a new specification named `LeaderElectionCrash` and paste the given skeleton code. We introduce a macro `MaybeCrash` that can crash processes. With the possibility of processes crashing, our properties are changed to the following:

- **Agreement**: All **correct** processes elect the same **correct** leader.
- **Completeness**: Eventually, all **correct** processes elect some **correct** leader.

Fill in the missing code and change the TLA+ properties accordingly. You are allowed to reuse and modify parts of the solution from the previous task. Run the model checker with different constants and report your findings. Why do we not allow crashes at the last round? How does the properties of the previous task get violated when processes can crash?

2.2 Abortable Paxos (10 Points)

In this task, the Abortable Paxos[3] algorithm will be implemented. The template code consist of two set of processes; Proposers and Acceptors. The Acceptor processes also have the role of Learners. For simplicity, all messages are placed on a message board. For instance, a message to a certain proposer is placed on the message board for all proposers. By maintaining all of the messages that has ever been sent allows us to define macros for checking the state of a process without having to introduce a history variable for each process. The template code is based on this example that provides a more extensive explanation of the modelling.

Fill in the missing PlusCal code. You are free to add additional code as needed, as long as it is well-documented. Define the properties of Abortable Paxos in TLA+ and run the model checker. In the report, describe the most important parts of your code and how you reasoned about them.

2.3 Leader-based Sequence Paxos (20 Bonus Points)

Implement the Leader-based Sequence Paxos algorithm with all the optimisations[4] in PlusCal. It is allowed to assign processes to a fixed role as Proposer or Acceptor. The

[3] The algorithm and its properties can be found in Section 2 of https://arxiv.org/pdf/2008.13456.pdf
[4] Algorithm 7 in https://arxiv.org/pdf/2008.13456.pdf

implementation should be able to handle crashes for both Proposers and Acceptors. You are free to implement the algorithm using any approach as long as the properties of Sequence Consensus are satisfied.

- **Validity**: If process p decides CS then CS is a sequence of proposed commands.
- **Uniform Agreement**: If process p decides CS and process q decides CS' then one is a prefix of the other.
- **Integrity**: If process p decides CS and later decides CS' then CS is a strict prefix of CS'.
- **Termination**: If a command C is proposed infinitely often by a correct process, then eventually every correct process decides a sequence containing C infinitely often.

In the report, discuss your approach for implementing the algorithm and specify how the model checker was used (constants, parameters, TLC options etc.).