Case Studies
Parameters in Parameterized Distributed Systems

PARDI Project

March 16, 2018

Abstract

This report presents several parameterized algorithms and systems. They are intended as representative samples, first to determine parameters, and secondly to identify limitations of current tools. An analysis of these case studies is also provided.

Contents

1 Introduction **4**

2 Case Studies **4**
2.1 Distributed Algorithms 4
2.1.1 Two Phase Commit 4
2.1.2 Consensus 5
2.1.3 Chandy-Misra Algorithm for Mutual Exclusion 9
2.1.4 Naimi-Tréhel Algorithm for Mutual Exclusion 10
2.1.5 Misra's Algorithm for Termination Detection 11
2.1.6 Ricart-Agrawala Algorithm for Mutual Exclusion . . . 11
2.1.7 Token-based Algorithm for Mutual Exclusion 12
2.2 Workflows 13
2.2.1 Examination Management System 13
2.2.2 Shift Worker Scheduling 14
2.2.3 A Conference Reviewing Process 15
2.3 Shared Memory Algorithms 15
2.3.1 Splitter 16
2.3.2 Renaming 17
2.3.3 Wait-Free Linked List 17
2.4 Summary 17

3 Numerical Parameters **19**
3.1 Processes 19
3.2 Number of Failures 20
3.3 Exchanged Values 20
3.4 Additional Parameters 21

4 Functional Parameters **21**
4.1 Network Topology 21
4.2 Process Failures 21
4.3 Communication Parameters 21

5 Limitations of Current Tools **23**
5.1 Cubicle 23
5.1.1 General Comments 23
5.1.2 Case studies 24
5.2 TLC 25
5.3 TLAPS 26

5.4 Synthesis 26

1 Introduction

This document is a companion to the case studies available in PARDI's repository. Section 2 describe these case studies. Sections 3 and 4 identify the numerical and functional parameters that the case studies exhibit. Section 5 presents the limitations of the current tools that we have encountered while building the case studies.

2 Case Studies

All case studies are available in PARDI's repository (`https://gforge.inria.fr/projects/anr-pardi/`)

2.1 Distributed Algorithms

2.1.1 Two Phase Commit

This holds two models of the two-phase commit protocol. Every agent individually decides whether they want to commit or abort. The coordinator then decides on commit or abort, where commit is possible only if all agents are ready to commit. In the abstract specification, the coordinator and the agents directly access each other memory, avoiding explicit communication. In the lower-level specification, agents and coordinator communicate explicitly by message passing. To model communication, the coordinator and each agent has an input set of pending messages (an agent receive only one message that comes from the coordinator, and the coordinator receives a message from each agent).

Models are given in PlusCal and Cubicle. The models are parameterized by the number of agents.

The PlusCal models are easily verified by TLC (the TLA$^+$ model checker) for a fixed and small number of agents: for seven agents, the abstract model has 72097 distinct states (depth 18) and the concrete model has 1663597 distinct states (depth 22). Of course, they cannot be verified for an arbitrary number of agents.

The Cubicle model of the abstract model is close the TLA$^+$ version. However, the Cubicle model of the concrete model is more different. The sets of input messages are modeled by two arrays: one for the coordinator to hold a message from each agent, and one for where each agent can receive one message. Moreover, the Cubicle model requires a ghost variable that

counts the aborts. A direct translation between TLA$^+$ and Cubicle is not feasible here.

2.1.2 Consensus

Consensus is one of the fundamental algorithms in the theory of distributed algorithms. It is a generic tool for fault tolerance, as it allows to build replicated state machine [Lam84, Sch90]. From a theoretical point of view, consensus (together with registers) is sufficient to implement any object with a sequential specification [Her91].

The consensus problem requires that N processes (or sites) agree on k common values (in the basic case, $k = 1$). Each process proposes one value, and the agreed value must be one of them. When dealing with faults, only correct processes are expected to decide. More precisely, consensus is defined by the four properties:

Agreement At most k distinct values are decided for the correct processes.

Integrity A process decides at most once (it doesn't change its decision value).

Validity The decision value is one of the proposed values.

Termination A correct process eventually decides.

Formally, consensus can be specified as follows in TLA$^+$:

MODULE *Consensus*

```
* The definition of what is consensus.
* Sites, proposed and decided values, and failures are explicit.
* This allows to precisely specify consensus with the usual four properties:
* Agreement, Integrity, Validity, Termination.

EXTENDS Naturals, FiniteSets

CONSTANT N,        number of sites
         VALUE     the set of possible proposed values

ASSUME N_IN_NAT ≜ N ∈ Nat ∧ N ≥ 1
ASSUME VALUE_NOT_EMPTY ≜ VALUE ≠ {}

SITE ≜ 1 .. N

VARIABLES
  proposed,      for each site, its proposed value
  hasdecided,    the set of sites that have decided on a value
```

$decision,$	for each site, the decided value
$hasfailed$	the set of sites that have failed

$$TypeInvariant \triangleq \begin{array}{l} \land\ proposed \in [SITE \rightarrow VALUE] \\ \land\ hasdecided \in \text{SUBSET } SITE \\ \land\ hasfailed \in \text{SUBSET } SITE \\ \land\ decision \in [SITE \rightarrow VALUE] \end{array}$$

—————— *Consensus* properties ——————

Agreement: At most k distinct values are decided for the correct processes.

$$Agreement(k) \triangleq \Box(Cardinality(\{decision[i] : i \in hasdecided \setminus hasfailed\}) \leq k)$$
$$Agreement1 \triangleq Agreement(1)$$
$$Agreement1b \triangleq \Box(\forall\, i, j \in hasdecided \setminus hasfailed : decision[i] = decision[j])$$

Uniform agreement: At most k distinct values are decided for the processes which have decided, be they failed or not.

$$UniformAgreement(k) \triangleq \Box(Cardinality(\{decision[i] : i \in hasdecided\}) \leq k)$$
$$UniformAgreement1 \triangleq UniformAgreement(1)$$

Integrity: a process decides at most once (it doesn't change its decision value)

$$Integrity1 \triangleq \Box[\forall\, i \in SITE : i \in hasdecided \Rightarrow decision[i]' = decision[i]]_{\langle proposed,\, hasdecided,\, decision,\, hasfailed\rangle}$$
$$Integrity2 \triangleq \forall\, i \in SITE,\, k \in VALUE : \Box(i \in hasdecided \land (decision[i] = k) \Rightarrow \Box(decision[i] = k))$$

Validity: the decision value is one of the proposed values.

$$Validity1 \triangleq \Box(\{decision[i] : i \in hasdecided\} \subseteq \{proposed[i] : i \in SITE\})$$
$$Validity2 \triangleq \Box(\forall\, i \in hasdecided : \exists\, j \in SITE : decision[i] = proposed[j])$$

Termination: A correct process eventually decides.

$$Termination1 \triangleq \forall\, i \in SITE : (\Box(i \notin hasfailed)) \Rightarrow (\Diamond(i \in hasdecided))$$
$$Termination2 \triangleq \forall\, i \in SITE : (\Diamond(i \in hasfailed)) \lor (\Diamond(i \in hasdecided))$$

Failure is stable. Not mandatory.

$$StableFailure \triangleq \forall\, i \in SITE : \Box(i \in hasfailed \Rightarrow \Box(i \in hasfailed))$$

The following implementations of consensus are available in the repository.

`abstract_async.tla` and `abstract_k_async.tla`: An abstract specification of consensus. Processes decide individually (asynchronously).

`abstract_sync.tla`: An abstract specification of consensus. Processes decide simultaneously (synchronously).

`async_broadcast.tla`: A simple algorithm for consensus in an asynchronous world: Each process non-atomically broadcasts its value, waits for all the values of the processes and chooses the min. It tolerates no process failure, and has an explicit dependency on the number of processes.

`async_broadcast_crash.tla`: A simple algorithm for K-consensus in an asynchronous world with crashes. Each process non-atomically broadcasts its value, waits for $N - K + 1$ values (where N is the number of processes) and chooses the min. The algorithm tolerates at most $K - 1$ permanent process crashes.

`async_broadcast_crash2.tla`: Idem. These two specifications differ in the way faults are modeled. In the first version, a broadcast action sends a message to all the processes; when a process crashes, the messages it has sent are removed from net. In this other version, when a process crashes during broadcast, a subset of the processes gets the message.

`sync_crash1.tla`: An algorithm for consensus in a synchronous world with permanent crash failure [Ray10, pp 16]. At each round, each process waits for broadcasted values, computes min and broadcasts its value if it has changed. $F + 1$ rounds tolerates F crashes. This specification is more synchronous than `sync_crash2`: all processes make their step simultaneously.

`sync_crash2.tla`: In this specification, processes independently progress in the same round. When all the processes have done their step, a new round is initiated.

`sync_omission2.tla`: An algorithm for consensus in a synchronous world with crash failure and omission failure [Ray10, pp 103]. At each round, each process waits for broadcasted values, computes min and broadcasts its value if it has changed. One last round exists to decide on the value with a majority of votes. $F + 2$ rounds tolerates F faults, where $F < N$ (the lowest bound is $F + 1$ rounds, but that algorithm is significantly more complex). In this specification, processes independently progress.

Additionally, the following refinements are defined :

`refinement_async_broadcast.tla`: async_broadcast is a refinement of abstract_async.

`refinement_async_broadcast_crash.tla`: async_broadcast_crash is a refinement of abstract_k_async.

`refinement_async_broadcast_crash2.tla`: async_broadcast_crash2 is a refinement of abstract_k_async.

`refinement_sync_crash1.tla`: sync_crash1 is a refinement of abstract_sync.

`refinement_sync_crash2.tla`: sync_crash2 is a refinement of abstract_sync.

`refinement_sync_omission2.tla`: sync_omission2 is a refinement of abstract_sync.

All specification have been verified by model checking for small numbers of processes and faults (up to 4). Moreover, TLAPS was used to prove some properties in `abstract_sync.tla`, namely TypeInvariant, Validity1, Validity2, Agreement1b and Integrity1.

Traps

- Broadcast is not atomic with regard to crash. This is important as the algorithms would be a lot simpler (and almost trivial) with atomic broadcast. This non-atomicity can be specified in several ways: splitting the step of a site in a sequence of send; using non determinism to send a subset of the expected messages on crash; removing some of the sent messages on crash.

  The second way is the clearest, and the closest to the usual description of crashes. An advantage of the third way is that its description is simpler with less actions and transitions, but one must be careful to not make visible impossible states. The first way is awkward and leads to state explosion.

- Bags: `async_broadcast` and `async_broadcast_crash` use a bag to collect the received values. A set plus a counter (to count how many values have been received) is enough: the multiplicity of each value is not used (the algorithm only needs the set of values and the cardinality of the bag).

Cubicle Cubicle is currently unable to verify the properties of consensus, except for agreement:

- Agreement is an invariant with universal quantification and it can be specified as unsafe states.
- Integrity is a stable property $\Box(p \Rightarrow \Box p)$ ou $\Box(p \Rightarrow p')$.
- Validity is an invariant which requires existential quantification or set inclusion.
- Termination is a liveness property $\Diamond p$.

2.1.3 Chandy-Misra Algorithm for Mutual Exclusion

These models describe Chandy-Misra bounded and adaptative algorithm for mutual exclusion [CM84]. This algorithm is permission based: sites give their permission to other nodes. It is bounded (in state space), as it uses a boolean priority between couple of sites to avoid starvation, and no timestamps are necessary. Lastly, it is adaptative, in the sense that if a site is not interested, it will eventually stops receiving request messages.

The TLA$^+$ model is a close translation of the description found in [Ray13, pp 259]. The Cubicle model is a translation of the TLA$^+$ model, removing priorities. Cubicle does not check liveness properties and priorities are only there for liveness. This is somewhat infortunate as the priority mechanism was one of the innovations of the algorithm.

Communication is fully asynchronous and modeled with a set (as there is no duplicated message, a bag is not required). The algorithm does not need FIFO 1-1 communication. Especially, a request message can pass over a previously sent permission message, without invalidating safety or liveness.

Results

- TLA$^+$ model with N = 3 sites: 1398 distinct states, depth = 27
- TLA$^+$ model with N = 4 sites: 542472 distinct states, depth = 50
- TLA$^+$ model with N = 5 sites: interrupted after 259333847 distinct states, depth = 41
- Cubicle : ok. It needs only 2 processes to conclude.
  Using option `-brab 2` found some interesting invariants, for instance the unsafe state:

```
Request[#1, #2] = False && Request[#2, #1] = False
```
(in any couple of sites, at least one must request entrance from the other before entering mutual exclusion)

2.1.4 Naimi-Tréhel Algorithm for Mutual Exclusion

Several models are given for the mutual exclusion algorithm by Naimi - Tréhel [NTA96]. The expected properties are those of an algorithm for exclusion mutual:

Exclusion No two distinct processes hold both the exclusive access:
$\Box(\forall i, j \in \mathit{Sites} : \mathit{token}[i] \land \mathit{token}[j] \Rightarrow i = j)$

Strong Liveness or absence of famine or individual progress: every requesting process eventually gets access:
$\forall i \in \mathit{Sites} : \mathit{requesting}[i] \leadsto \mathit{token}[i]$

Weak Liveness or absence of deadlock or global progress: if there are requests, there will eventually be (some) answers:
$(\exists i \in \mathit{Sites} : \mathit{requesting}[i]) \leadsto (\exists i \in \mathit{Sites} : \mathit{token}[i])$

The models are inspired by `http://www-master.ufr-info-p6.jussieu.fr/2007/Ajouts/Master_esj20_2007_2008/IMG/pdf/Naimi-Trehel.pdf`. They mainly differ in the communication ordering. See [CHQ16] for a description of the communication models.

`naimi_trehel_async.tla`: A TLA$^+$ model where communication is asynchronous and is modeled with a global set. Exclusion and WeakLiveness are verified for N = 5 sites (158020 states). As expected, StrongLiveness is NOT verified: a counter-example exists with 3 sites. Intuitively, the token or another request can incessantly pass over a request that is propagated along the chain of requesters.

`naimi_trehel_fifo11.tla`: A TLA$^+$ model where communication is FIFO 1-1 and is modeled with a sequence for each couple of sites. Exclusion and WeakLiveness are verified for N = 5 sites (440120 distinct states). StrongLiveness is not verified (a counter-example exists with 3 sites). This is counter-intuitive as the algorithm claims to work with this communication model.

`naimi_trehel_fifon1.tla`: A TLA$^+$ model where communication is FIFO n-1 and is modeled with an array of sequences as input buffers. Exclusion, WeakLiveness and StrongLiveness are verified for N = 5 sites (223995 distinct states).

naimi_trehel_fifonn.tla: A TLA$^+$ model where communication is FIFO n-n and is modeled with a global sequence. Exclusion, WeakLiveness and StrongLiveness are verified for N = 5 sites (509355 distinct states).

naimitrehel2.cub: A Cubicle model, with asynchronous communication. There can be at mosy one message in transit from x to y. A new message replaces the previous one. This certainly invalidates liveness (lost requests) but cannot invalidate safety. Unfortunately, Cubicle died with out of memory after 234 min.

2.1.5 Misra's Algorithm for Termination Detection

Variations around Misra's algorithm to detect termination with a token on a ring [Mis83]. The two expected properties of an algorithm to detect termination are:

Correction If termination is detected, processes are actually inactive:
$\Box(\mathit{detection} \Rightarrow \forall i \in \mathit{Process} : \mathit{inactive}[i])$

Completeness If all processes are inactive, termination will be detected:
$(\forall i \in \mathit{Process} : \mathit{inactive}[i]) \land \mathit{network} = \mathit{EmptyBag} \leadsto \mathit{detection}$

The following TLA$^+$ models are proposed. No effort was done to write a Cubicle model as the expected properties cannot be verified by Cubicle.

misra1.tla: Communication is imperfectly modeled with an input buffer per process, implemented with sequences. Thus, it is analogous to FIFO n-1 communication, where as the algorithm only needs FIFO 1-1 communication.

misra3.tla: Synchronous (instantaneous) communication of messages and token.

mirsa_async.tla: Arbitrary (asynchronous) communication implemented with a bag. As expected, the algorithm does not work: it is known that it requires at least FIFO 1-1 communication, else the token can pass over an applicative message.

2.1.6 Ricart-Agrawala Algorithm for Mutual Exclusion

An algorithm for mutual exclusion [RA81]. This algorithm is based on permission. A requesting process asks permission from all the other. To ensure liveness, requests are ordered by date using Lamport's clocks. A version that

only ensures safety can use static priorities. The algorithm does not tolerate any failure (with crash failure, safety is nevertheless verified as the other processes will block waiting for an answer from the faulty site). Communication can be fully asynchronous. As the algorithm uses Lamport's clocks to order the requests, its state space is infinite.

2.1.7 Token-based Algorithm for Mutual Exclusion

This presents several variations of algorithms for mutual exclusion based on a token. Only the process that holds the token can enter mutual exclusion. To ensure starvation freedom, a topology where the token infinitely often visits any requesting node is required. The simplest case uses an endlessly circulating token on a logical ring. Actual implementations of this algorithm use request messages which are propagated to the holder of the token (e.g. [SK85] which uses broadcast to ask for the token, or [Ray89] which uses a spanning tree). This avoids movements of the token when no node is interested.

`token1a.tla` and `token1b.tla`: The token is modeled by a unique shared variable holding the owner, and the transmission of the token is atomic. Liveness is ensured with strong fairness (`token1a.tla`) or applicative actions that force the token to move (`token1b.tla`).

`token2.tla`: The variable token is split in an array of boolean. The transmission of the token is atomic (synchronous communication).

`token3.tla`: The variable token is split in an array of boolean. The transmission of the token is asynchronous, implemented via channels. As there can be at most one message in transit in the whole system, ordering properties have no effect.

`token*.cub` Five variations which differ in the token representation (one unique shared variable or an array of boolean) and the communication (synchronous or asynchronous).

The TLA$^+$ models are quickly model-checked for a small number of sites (e.g. one or two seconds for 6 nodes). The Cubicle models are also quickly checked and proved correct.

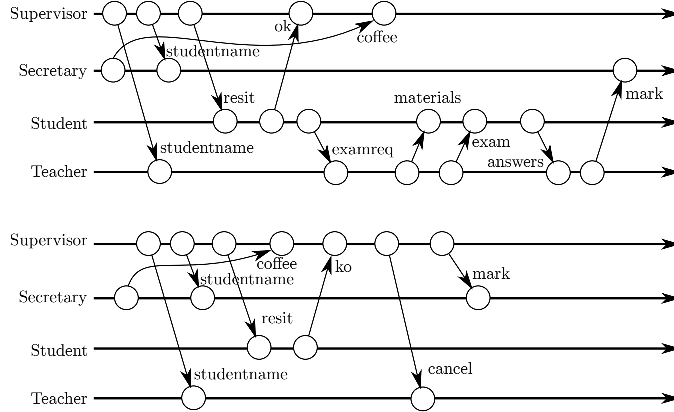

Figure 1: Examples of Expected Executions

2.2 Workflows

2.2.1 Examination Management System

This example is taken from [CHQ16]. Let us consider an examination management system composed of a student, a supervisor, a secretary, and a teacher. When the supervisor notices that a student has failed, he/she sends the name of the student to the teacher and the secretary, and the resit information to the student. If the student chooses to resit, he/she answers ok and asks the teacher for the exam. The teacher sends the needed materials and then the exam, after which the student sends back his/her answers, then the teacher sends a mark to the secretary. If the student declines to resit, he/she informs the supervisor who sends a cancel message to the teacher and the former mark to the secretary. An unrelated exchange also occurs between the secretary and the supervisor who would like to meet during the coffee break. The secretary sends a message to inform the supervisor that coffee is ready. The supervisor is ready to join after he/she has sent work-related messages: just before, after, or during the time he deals with the student's choice. Sample executions are depicted in Figure 1 and the system is specified in Figure 2.

This example is available in different forms:

- A TLA$^+$ specification, parameterized with the communication model. Causal communication is required between the channels $\{\mathit{studentname}, \mathit{resit}, \mathit{examreq}, \mathit{cancel}, \mathit{mark}\}$, FIFO 1-1 is required between $\{\mathit{materials}, \mathit{exam}\}$, and no ordering constraint is necessary for

$$
\begin{array}{rcl}
\textbf{Supervisor} & \triangleq & \mathit{studentname}! \cdot \mathit{studentname}! \cdot \mathit{resit}! \\
& & \quad \cdot ((\mathit{ok}? \cdot 0 + \mathit{ko}? \cdot \mathit{cancel}! \cdot \mathit{mark}! \cdot 0) \| (\mathit{coffee}? \cdot 0)) \\
\textbf{Secretary} & \triangleq & \mathit{coffee}! \cdot \mathit{studentname}? \cdot \mathit{mark}? \cdot 0 \\
\textbf{Student} & \triangleq & \mathit{resit}? \cdot (\tau \cdot \mathit{ko}! \cdot 0 + \tau \cdot \mathit{StudentOK}) \\
\mathit{StudentOK} & \triangleq & \mathit{ok}! \cdot \mathit{examreq}! \cdot \mathit{materials}? \cdot \mathit{exam}? \cdot \mathit{answers}! \cdot 0 \\
\textbf{Teacher} & \triangleq & \mathit{studentname}? \cdot (\mathit{cancel}? \cdot 0 + \mathit{examreq}? \cdot \mathit{TeacherExam}) \\
\mathit{TeacherExam} & \triangleq & \mathit{materials}! \cdot \mathit{exam}! \cdot \mathit{answers}? \cdot \mathit{mark}! \cdot 0
\end{array}
$$

Figure 2: Examination Management Specification. x! denotes the asynchronous sendi of a message on x, and x? denotes the blocking reception. $+$ is a non-deterministic choice. $\|$ denotes interleaving of the two processes.

$\{\mathit{ok}, \mathit{ko}, \mathit{answers}, \mathit{coffee}\}$.

- A cubicle model (`university1.cub`) that uses asynchronous communication, described with channels which count the number of messages inside them. As expected, this model is incorrect and Cubicle finds a counter-example where the teacher receives a cancel before it has received the studentname.

- A Cubicle model (`university2.cub`) that uses RSC communication, where at most one channel can hold a message at a given time. After 20 *min*, Cubicle finds an error trace with two students. The first student receives a resit message, answers with ok and sends an examreq. The teacher answers with the materials. The second student gets these materials without having previously received a resit message.

- A Cubicle model (`university3.cub`) that uses RSC communication and imposes only one student. Cubicle has been interrupted after 5,5 hours, without conclusion.

- A BPMN collaboration (`https://github.com/pascalpoizat/bpmn-models/blob/master/pictures/e0001v0001.png`)

- A BPMN choreography (`https://github.com/pascalpoizat/bpmn-models/blob/master/pictures/e0001v0002.png`).

2.2.2 Shift Worker Scheduling

This example has been taken and slightly modified from [BW10].

A BPMN collaboration is available (`https://github.com/pascalpoizat/bpmn-models/blob/master/pictures/e0002v0001.png`).

2.2.3 A Conference Reviewing Process

Let's consider a conference reviewing system. The peers are the authors, the presidents of the program committee and the reviewers. Authors send their papers to all the presidents of the program committee (multicast to the presidents). Each president attributes a paper number and takes responsibility for a part of the papers, based on this number. In order that all presidents attribute the same number to a given paper, and without internal coordination between the presidents, the authors use a totally ordered multicast so that the papers are delivered in the same order to all the presidents (fifo n-1 ordering model). After the deadline has passed, the presidents reject new submissions (point-to-point communication to the author). After the deadline, each president independently sends its papers to some of the reviewers (bounded multicast to the reviewers), waits for the reviews (multicast from the reviewers to the presidents), and sends the acceptance result to the author (point-to-point). The system must ensure that it does not deadlock and that every author eventually receives an answer (either rejection for a late paper, or acceptance result if reviewed). This system exposes both strict ordering constraints (submissions sent to the presidents), and high interleaving (each reviewer is independently handling the papers it has received).

The system has been described in the PlusCal Agorithm Language, which is translated by the TLA$^+$ tools to a TLA$^+$ specification, which can then be checked with the TLC model checker. Results have shown that the message cap on the number of messages in transit is instrumental to avoid state explosion as it ensures that messages are not delayed for too long. During the development of the system, several bugs were found. For instance, the logic to split the papers amongst the presidents was faulty with an odd number of presidents (e.g. one...) and some authors were never receiving their acceptance result; in some cases, the same paper was sent twice to the reviewers, and an unfortunate (but legal) interleaving in the reception of the reviews led to two acceptance messages to the same author. This system, albeit simple, already experiences enough communication interactions to warrant formal verification. Moreover it is heavily parameterized: on the number of authors, on the number of presidents, on the number of reviewers, on the number of needed reviews.

2.3 Shared Memory Algorithms

Shared memory algorithms are not the main focus of PARDI. However, they present interesting challenges and abstract specifications of distributed algo-

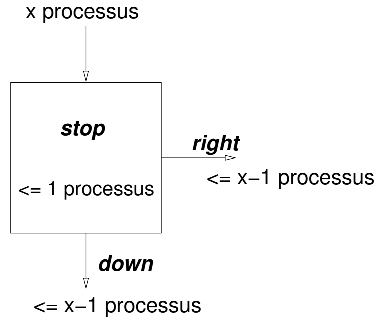

Figure 3: Splitter Object

rithms and shared memory algorithms are close enough to warrant study.

2.3.1 Splitter

The splitter is an concurrent object used to distinguish callers (Figure 3). Each process that calls the splitter gets a decision value among *stop*, *left*, *right*. The decision values respect the following rules:

- x processes concurrently enter the splitter. The value x is unknown.
- At most one process ends with *stop*.
- if $x = 1$, the process necessarily ends with *stop*.
- At most $(x - 1)$ processes end with *right*.
- At most $(x - 1)$ processes end with *down*.

An algorithm for the splitter uses two shared registers: X (initially any value) and Y (initially false). Each process has a unique identifier id_i.

direction(id_i)
  $X \leftarrow id_i$;
  if Y **then** $dir_i \leftarrow right$;
  else $Y \leftarrow true$;
    if $(X = id_i)$ **then** $dir_i \leftarrow stop$;
    else $dir_i \leftarrow down$; **endif**
  endif
  return dir_i;

This algorithm has been translated in a PlusCal model (`splitter.tla`) and has been model checked for up to 7 processes (188718572 distinct states

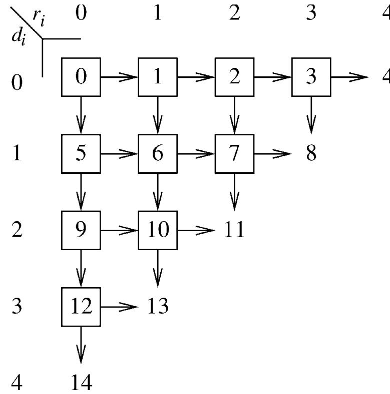

Figure 4: Renaming with an array of splitter

/ depth 50). The Cubicle descriptions are currently non-conclusive: Cubicle runs forever ($> 15\ h$) without conclusion.

2.3.2 Renaming

Using splitters, a problem known as renaming can be solved [MA95]. Let's n process with distinct identities $id_1, \ldots, id_n \in [0..N]$ where $N \gg n$ (where $\gg$ means much greater than). The goal is to give the processes new distinct identities in $[0..M]$ where $M \ll N$.

A wait-free solution uses an array of splitters and solves the renaming problem with $M = \frac{n(n+1)}{2}$ (Figures 4 and 5). Verification of `renaming.tla` with model checking has been performed up to 4 processes (6381732 distinct states, depth = 58). No Cubicle model is available as arithmetic is not possible on process identifiers.

2.3.3 Wait-Free Linked List

This presents a wait-free linked list, with concurrent enqueues and dequeues and no lock [MS96]. The TLA$^+$ model has two parameters: the number of processes (NP) and the number of cells (NC, the maximal length of the list). The current TLA$^+$ model (in PlusCal) is close to the algorithmic code but is rather naive with regard to symmetry. This leads to early state explosion.

2.4 Summary

This table summarizes the number of variants of each problem with each specification language.

$d_i \leftarrow 0; r_i \leftarrow 0; term_i \leftarrow false;$
while $(\neg term_i)$ **do**

> $X[d_i, r_i] \leftarrow id_i;$
> **if** $Y[d_i, r_i]$ **then** $r_i \leftarrow r_i + 1;$ % *right*
> **else** $Y[d_i, r_i] \leftarrow true;$
>   **if** $(X[d_i, r_i] = id_i)$ **then** $term_i \leftarrow true;$ % *stop*
>   **else** $d_i \leftarrow d_i + 1;$ % *down*
>   **endif**
> **endif**

endwhile
return $(n * d_i + r_i - (d_i(d_i - 1)/2))$
% *name = cell* d_i, r_i

Figure 5: Algorithm for renaming based on an array of splitter

Problem	TLA$^+$	Cubicle	Workflow
Two Phase Commit	2	3	
Consensus	10	–	
Chandy-Misra Mutual Exclusion	1	1	
Naimi-Tréhel Mutual Exclusion	4	2	
Misra Termination	3	–	
Token-based Mutual Exclusion	5	5	
Examination Management Sys.	1	3	2
Shift Worker Scheduling	–	–	1
Reviewing	1	–	–
Splitter	1	3	
Renaming	1	–	
Linked List	1	–	

3 Numerical Parameters

These parameters are often represented as naturals. We will see that numerical operations are sometimes unnecessary, and a simple set of distinct values is often enough.

3.1 Processes

The most usual parameter of a distributed algorithm is the number of processes / sites. For instance, this is apparent in the specification of Naimi Tréhel exclusion algorithm, of Misra termination algorithm, of the consensus problem. Actually, some algorithms do not need to count the processes, just to know the set of processes, or even just to distinguish between two processes.

Unused value Several algorithms do not need to know the value of this parameter. For instance, Naimi-Tréhel algorithm for mutual exclusion and Chandy-Misra algorithm for mutual exclusion don't need to know neither the number of processes, nor the identity of all the process (as long as they have distinct identities). The splitter also do not use the number of processes in its implementation, but it is used in its specification to express "at least one" and "not all".

Explicit usage Other algorithms need to know the set of processes. The number of processes is used in Misra's termination detection algorithm to build a ring. This algorithm needs the number of processes for two different roles: first to compute the next neighbor in the ring ($next(self) = (self + 1) \bmod N$); secondly, to count the number of passive nodes and concludes if all sites are passive when N consecutive passive (and white) nodes are found.

In the asynchronous implementation of consensus based on broadcast, the set of all processes is needed so that a process knows it has received a message from all the processes.

In Ricart-Agrawala mutual exclusion algorithm, a process request permission from all the other, and waits for an answer from each of them. A set of pending request is needed in the algorithm.

Common operations on processes

- No numeric operations. Specification properties may use $\forall site$ and $\exists site$, and the algorithm may need $\exists j \neq self$.

Examples: splitter (2.3.1), Naimi-Tréhel (2.1.4).

- Set operations. For instance the set of all the processes, subset of processes (SUBSET *Sites*), membership and inclusion tests ($\in$, $\subseteq$). Cardinality is rarely used or can often be replaced by set comparison.

  Examples: Two Phase Commit (2.1.1), Ricart-Agrawala (2.1.6), Chandy-Misra (2.1.3), Naimi-Tréhel (2.1.4), specification of the consensus problem (2.1.2), asynchronous consensus.

- Structural properties. For instance, processes are organized on a ring, and the algorithm uses $next(self)$ and $previous(self)$.

  Examples: Misra's termination algorithm (2.1.5), token-based mutual exclusion (2.1.7).

- Explicit numerical count. $+1$ or -1, and test $= N$ or $< N$.

  Examples: Misra's termination algorithm (2.1.5) to detect a complete cycle, synchronous consensus with crash or omission failure to count the number of rounds (2.1.2), renaming using splitters (2.3.2).

Multiple classes of processes In some cases, especially in workflows, there are several classes of processes, each one is itself parameterized. For instance, the conference reviewing workflow has three classes of processes (authors, presidents of the committee, reviewers) whose sizes are independent; a client-server system has usually two classes of processes.

3.2 Number of Failures

In the consensus problem, the number of site failures is a second parameter. This parameter is linked to the number of processes. With synchronous communication, the consensus is solvable with the following constraints:

site failure	constraint
none	ok
crash	f failures $< n$ processes
omission	$f < n/2$
byzantine	$f \leq \lfloor (n-1)/3 \rfloor$

3.3 Exchanged Values

Some algorithms use a set of exchanged values between the sites. For instance, the description of the consensus problem states that each process

proposes a value among a set of possible values. In the consensus problem, it is known that two distinct values are enough to reason. This parameter is simple as it is generally sufficient to know that at least 1 (or 2 or...) distinct values exist. It is often explicitly instanciated to prove an algorithm.

3.4 Additional Parameters

The k-consensus problem introduces a third parameter: the maximal number of decided values ($k = 1$ is the standard consensus). There exist algorithms to solve the k-consensus problem with asynchronous communication and crash failure, as long as $f < k < n$, where f is the number of failures, k the maximal number of decided values, n the number of sites.

4 Functional Parameters

4.1 Network Topology

The network topology is a parameter which can influence the correctness of an algorithm. For instance, in the token-based algorithm for mutual exclusion 2.1.7, the token must visit infinitely often all the nodes to ensure freedom of starvation. The simplest solution is to define a static logical ring with all the nodes. However, the algorithm does not depend on this ring structure, only that the token visits all nodes. Other structures are allowed, static as well as dynamic. Note that this structure only influences starvation, and the safety (the mutual exclusion property) does not depend on the topology.

4.2 Process Failures

Obviously, the kind of failure which can occur is an important parameter. The vast majority of fault-tolerant algorithms consider one of the four classes of process failure: no failure, crash failure (permanent halt of a process), omission failure (missing reception or send of a message), byzantine failure (arbitrary failure). An algorithm is generally dedicated to one class of failure and this parameter is thus instanciated.

Additional work in PARDI is expected, especially with regard to failure detectors [CT96].

4.3 Communication Parameters

Interaction in distributed systems is plagued with multiple properties: point-to-point versus multicast vs broadcast, synchronous vs asynchronous, failures

(loss, duplication, corruption), ordering... This makes it difficult to specify and verify distributed algorithms.

Asynchronous communication is often viewed as a single entity, the counterpart of synchronous communication. Although the basic concept of asynchronous communication is the decoupling of send and receive events, there is actually room for a variety of additional specification of the communication, for instance in terms of ordering. Yet, these different asynchronous communications are used interchangeably and seldom distinguished.

However, in distributed algorithms research, it has long been known that the properties of the communication, and especially the order of message delivery, is essential to the correctness of the system. For instance, the Chandy-Lamport snapshot algorithm [CL85] requires that the communication between two processes is FIFO 1-1, and Misra's algorithm for termination detection (2.1.5, [Mis83]) works with a ring containing each *node* if the communication ensures causal delivery, but requires a cycle visiting all network *edges* if communication is only FIFO 1-1.

The communication model is often merged in the model of the algorithm, and one has to guess which properties it has. For instance, a set may be used, with transitions to add and remove from this set. One guesses that asynchronous communication is used. However, messages can carry additional information, such as a timestamp or a logical clock, which determine which messages are available for reception. In this case, the communication may be FIFO-ordered or causally-ordered or... Additionally, communication failures may also be merged, with transitions to non-deterministically remove (or not adding) messages.

Work has been conducted to isolate the communication model from the TLA$^+$ models of the algorithm. We have built a framework where one can use specific modules to build one's communication model. Modules are available to specify seven ordering (FIFO 1-1, causal, FIFO n-n...), two modalities (point to point, multicast), constraints (e.g. number of messages in transit). Current work aims at specifying failures. This allows to easily check a given algorithm with different communication models.

5 Limitations of Current Tools

Cubicle, a model checker for array-based systems, and TLAPS, a proof assistant well-adapted to study distributed algorithms, are the basis of this project. Here are some limitations which have prevented the verification of some of the presented algorithms. The listed limitations are not intended to denigrate these tools, which have already shown their power. These are directions for improvement.

5.1 Cubicle

All comments apply to Cubicle 1.0.2. The newer version of january 2018 (1.1.2) is currently under consideration.

5.1.1 General Comments

- Cubicle checks state invariant properties only. Stability can be checked by introducing ghost variables. Liveness properties are of course out of scope of Cubicle. State Reachability can be checked by invalidating an unsafe property with these states.

- Asymmetric initialisation is cumbersome. For instance, the token-based algorithm 2.1.7 needs to start with the token on some node. In this case, the solution is to start without a token and to have a transition that creates a token if none exists.

- Asymmetric behavior of processes is cumbersome to describe and leads to models (seemingly) impossible to check. Cubicle is targeted to systems where all processes have the same behavior. Workflows are typically constructed of several processes with completely different behaviors (see 2.2.1 for example). This example (`university1.cub` and `university2.cub`) uses acrobatic transitions to initialize the processes in different states.

- In the Splitter, we were unable to express a “not all” property (not all processes end with right, not all processes end with down) in a way which was checkable by Cubicle. This is directly linked to the fault model of Cubicle, where processes may stop and disappear. Unsuccessful attempts were made with additional transitions or with ghost variables.

- No existential quantification. However, existential quantification on process is available in the transitions by adding a parameter to the transitions.
- No support for high-level data structures, such as sets or lists. However, sets of processes are available, by coding their characteristic function as a boolean array.
- No arithmetic operations on process identifiers. Process identifiers are opaque values which can only be compared for equality and ordering ($<$).

5.1.2 Case studies

Here is the advancement of the Cubicle models of the case studies.

Two Phase Commit OK for the abstract model. The concrete model needs a ghost variable to be verified by Cubicle.

Consensus Little success.

Difficulties regarding the expected properties:

- Agreement (an invariant with universal quantification) can be specified as unsafe states.
- Integrity is a stable property $\Box(p => \Box p)$ ou $\Box(p \Rightarrow p')$.
- Validity is an invariant which requires existential quantification or set inclusion.
- Termination is a liveness property $\Diamond p$.

Difficulties regarding the modeling of algorithms:

- In TLA$^+$, ad-hoc modeling of communication and failures, with heavy use of sets and bags.
- In some versions, synchronous actions where all peers make a step.

Chandy-Misra Algorithm for Mutual Exclusion Almost OK, the Cubicle model is a close translation of the TLA$^+$ specification. The limits are the removal of priorities (required for liveness only) and an over-strict initialisation.

Naimi-Tréhel Algorithm for Mutual Exclusion Almost OK, the Cubicle model is a close translation of the TLA$^+$ specification. The restriction is that there can be at most one message in transit from x to y, whereas the TLA$^+$ specification allows for several messages. This can actually happen, but only liveness is invalidated by dropping messages.

Misra's Algorithm for Termination Detection No Cubicle model. The interesting properties are a global safety property with all the processes and which imposes no failure, and a liveness property. None can be checked by Cubicle.

Token-based Algorithm for Mutual Exclusion OK, various models close to the TLA$^+$ specifications.

Examination Management System Semi-success. The Cubicle model is a close translation of the TLA$^+$ specifications (except for the communication). The invalid models (asynchronous communication, or two students with RSC communication) are correctly detected by Cubicle. The (expected) correct model runs infinitely.

Shift Worker Scheduling No TLA$^+$ specification, no cubicle model.

Conference Reviewing No cubicle model, several parameters.

Splitter The Cubicle model, a direct translation of the TLA$^+$ specification, exists. It has been augmented with ghost variables, but runs for hours without conclusion.

Renaming KO. It needs basic arithmetics and arrays indexed by integer.

Linked list KO. A direct translation of the TLA$^+$ specification requires two independent parameters. Not much effort was given to this example.

5.2 TLC

- TLC is an enumerative model checker. It needs instantiating all the parameters with small values.

5.3 TLAPS

- Using TLAPS to create a proof needs great expertise, both in the proved algorithm and in proof methods.
- There is little feedback when a step is invalid. Is it really false? Or just unsuccessful? What is missing? Often, the verification of a step fails because of tiny things, such as a missing type property or a forgotten expansion of a definition.

5.4 Synthesis

Problem	TLA$^+$ (N max)	Cubicle	Workflow
Two Phase Commit	N=7	∼OK	
Consensus	N=2 to 4	no model	
Chandy-Misra Mutual Exclusion	N=4	OK	
Naimi-Tréhel Mutual Exclusion	N=5	∼OK	
Misra Termination	N=5	no model	
Token-based Mutual Exclusion	N=6	OK	
Examination Management Sys.	fixed	KO	collab. & chor.
Shift Worker Scheduling	no model	no model	collab.
Conference reviewing	2 of each	no model	
Splitter	N=6	KO	
Renaming	N=3	no model	
Linked List	NP+NC=7	no model	

The N max value in the TLA$^+$ column is the maximal number of processes for which the model was verified with TLC without much effort: no parallelism and under 1 hour. With more effort, one can expect to gain one or two processes.

In the Cubicle column, KO means that a model has been written but Cubicle was unable to check it (generally, it runs for ever). "No model" means that we do not have a Cubicle model, generally because of limited expressiveness.

References

[BW10] Jeremy W. Bryans and Wei Wei. Formal analysis of BPMN models using event-b. In Stefan Kowalewski and Marco Roveri, editors, *15th International Workshop on Formal Methods for Industrial Critical Systems*, volume 6371 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2010.

[CHQ16] Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. On the diversity of asynchronous communication. *Formal Aspects of Computing*, 28(5):847–879, September 2016.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[Lam84] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, April 1984.

[MA95] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, 1995.

[Mis83] Jayadev Misra. Detecting termination of distributed computation using markers. In *2nd ACM Symposium on Principles of Distributed Computing*, pages 290–294, August 1983.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275. ACM, 1996.

[NTA96] Mohamed Naimi, Michel Trehel, and André Arnold. A log(N) distributed mutual exclusion algorithm based on path revers al. *Journal of Parallel and Distributed Computing*, 34(1):1–13, April 1996.

[RA81] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.

[Ray89] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, January 1989.

[Ray10] Michel Raynal. *Fault-Tolerant Agreement in Synchronous Message-Passing Systems*. Morgan & Claypool Publishers, 2010.

[Ray13] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SK85] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, November 1985.