Proving a Non-Blocking Algorithm for Process Renaming with TLA$^{+\star}$

Aurélie Hurault and Philippe Quéinnec

IRIT – Université de Toulouse
{hurault,queinnec}@enseeiht.fr

Abstract. Shared-memory concurrent algorithms are well-known for being difficult to write, ill-adapted to test, and complex to prove. Wait-free concurrent objects are a subclass where a process is never prevented from progressing, whatever the other processes are doing (or not doing). Algorithms in this subclass are often non intuitive and among the most complex to prove. This paper presents the analysis and the proof of a wait-free concurrent algorithm that is used to rename processes. By its adaptive and non-blocking nature, the renaming algorithm resists to test, because of the cost of covering all its states and transitions even with a small input set. Thus, a proof has been conducted in TLA$^+$ and verified with TLAPS, the TLA$^+$ Proof System. This algorithm is itself based on the assembly of wait-free concurrent objects, the splitters, that separate processes. With just two shared variables and three assignments, a splitter seems a simple object but it is not linearizable. To avoid explicitly in-lining it and dealing with its internal state, the proof of the renaming algorithm relies on replacing the splitter with a sequential specification that is proved correct with TLAPS and verified complete by model-checking on finite instances.

Keywords: Formal verification · Non-blocking algorithm · TLA$^+$.

1 Introduction

Increase of computer performance is now obtained by multi-core multiprocessor units. Concurrent programming with locks or monitors is an old topic, and methodologies have been presented that allow developing software with few (or at least not too many) synchronization bugs. All these methodologies revolve around a *blocking* paradigm: identifying the condition for valid progress (e.g. exclusive access on a resource), and blocking while this condition is not true (e.g. waiting to acquire a lock). However, these approaches are inefficient under high load and are subject to deadlock when a process stops while holding a lock or an access. Non-blocking algorithms have been considered to alleviate these difficulties [MS98]. In a non-blocking algorithm, the delay or failure of a process does not prevent the progress of other processes. Subclasses of non-blocking include lock-free, where system progress is guaranteed, and wait-free, where process progress

* Supported by project PARDI ANR-16-CE25-0006.

is guaranteed [Her91]. Thus, wait-free algorithms are the most interesting to ensure individual progress in an adversary environment. Whatever the progress of the other processes, a process will be able to progress as long as the scheduler ensures that it has access to some CPU resources. Thus, wait-free algorithms are of great importance in highly concurrent environments.

Unfortunately, wait-free algorithms are hard to design and prone to bugs. They rely on low-level instructions, such as test-and-set, compare-and-swap, or shared data structures. Among these, atomic registers [Lam86,KS11] provide strong guarantees in presence of concurrency. An atomic register is a linearizable object with two operations, read and write. Progress is guaranteed even in presence of process failure. In a system where the only shared objects between processes are atomic registers, any concurrent execution (with true parallelism) is equivalent to a sequential execution on a single processor with interleaving. This nice property is instrumental in checking and verifying a wait-free algorithm as it spares us to consider true parallelism.

Nevertheless, writing a wait-free algorithm is error-prone. Correctness is often on paper only, and cannot be certified. Lastly, testing these wait-free algorithms is difficult as they generally exhibit a large reachable state space, and execution interleaving leads to a huge number of executions to consider. Bugs are often hidden in the deepest part of the algorithm and few schedulings can trigger them. Wait-free algorithms and data structures have started being used in standard libraries (for instance the `ConcurrentLinkedQueue` class in Java) but their first implementations were not always exempt of bugs [JDK08].

In this paper, we present the study of Moir and Anderson renaming algorithm [MA95]. The goal of this algorithm is to assign names (or identifiers) to processes such that the size of the namespace is small. This algorithm is wait-free, and is itself built upon wait-free objects, the splitters. This study has been conducted with the TLA$^+$ tools [Lam94] to provide a certified proof of the correctness of the renaming algorithm.

The contributions of this paper are:

- A mechanized proof of a wait-free renaming algorithm, based upon wait-free objects;
- An approach to handle the internal wait-free objects as black boxes;
- A combination of model-checking and formal proofs to verify correctness and completeness of the wait-free objects.

The *TAP Artifact Evaluation Committee* has reviewed the full artifact of our approach (available on [HQ19]). The four reviewers have found it consistent with the paper and have noted that the structure of the proofs in the artifact match their description in the paper. Regarding correctness, the four proof files have all been replayed without errors. As expected, checking the completeness is more intricate, and the reviewers have reached different upper bounds depending on their computing power.

2 TLA$^+$ Specification Language & Tools

2.1 Language

TLA$^+$ [Lam02] is a formal specification language based on untyped Zermelo-Fraenkel set theory for specifying data structures, and on the temporal logic of actions (TLA) for specifying dynamic behaviors. TLA$^+$ allows specifying symbolic transition systems with variables and *actions*. An action is a transition predicate between a state and a successor state. It is an arbitrary first-order predicate with quantifiers, set and arithmetic operators, and functions. In an action, x denotes the value of a variable x in the origin state, and x' denotes its value in the next state. Expressions rely on standard first-order logic, set operators, and several arithmetic modules. Functions are primitive objects in TLA$^+$. The application of function f to an expression e is written as $f[e]$. The set of functions whose domain is X and whose co-domain is a subset of Y is written as $[X \to Y]$. The expression DOMAIN f is the domain of the function f. The expression $[x \in X \mapsto e]$ denotes the function with domain X that maps any $x \in X$ to the expression e (which can include x). The notation $[f$ EXCEPT $![e_1] = e_2]$ is a function which is equal to the function f except at point e_1, where its value is e_2. A specification of a system is usually a disjunction of actions. Fairness, usually expressed as a conjunction of weak or strong fairness on actions, or more generally as an LTL property, ensures progression.

2.2 Tools

The TLA$^+$ toolbox contains the TLC model checker, the TLAPS proof assistant, and various tools such as a translator for the PlusCal Algorithm Language [Lam09] into a TLA$^+$ specification, and a pretty-printer that converts a textual TLA$^+$ specification into a LaTeX file.

PlusCal Pluscal is an algorithm language that looks like a programming language (assignment, loop, conditional) augmented with constructs for describing concurrency and non-determinism. PlusCal is actually more expressive than a traditional programming language as its expressions are any TLA$^+$ expressions.

TLC TLC, the TLA$^+$ Model Checker, is an enumerative explicit-state model-checker that can check safety and liveness properties. Its parallel implementation achieves a close to linear speedup for checking safety properties. To verify a TLA$^+$ specification, TLC requires all constants (e.g. number of processes) to be instantiated.

TLA$^+$ Proof System TLAPS, the TLA$^+$ Proof System, is a proof assistant for writing and checking TLA$^+$ proofs [CDLM10]. TLA$^+$ proofs are written in a hierarchical and declarative style with steps and substeps. A proof manager translates these steps in proof obligations, checks the trivial ones and uses backend

provers for the other ones. These backend provers include SMT provers (CVC3 is supplied, and Z3, CVC4, VeriT and Yices are supported), a TLA$^+$ theory in Isabelle, Zenon (an automated theorem prover for first-order logic based on the tableau method), or LS4 (to prove Propositional Temporal Logic).

3 The Renaming Problem

The renaming problem consists in renaming processes so they have a unique name on a small space. Initially, every process is assumed to have a unique identifier on an arbitrary large space (for instance, the IP address of the computer it is running on and the memory address of its control block). The only property of these identifiers is that they allow distinguishing processes, that is they can be compared for equality. A renaming algorithm assigns a distinct value, called a name, to each process, so that no two processes have the same name and the range of these names is small with regard to their initial identifiers. This algorithm is useful as it allows to efficiently refer to a set of processes, for instance by using an array indexed with their names.

In a shared-memory context, a trivial algorithm can solve the renaming problem using a counter and a lock. To get a name, a process acquires the lock, gets the current value of the counter and increments it, and releases the lock. Then the range of names for renaming N processes is optimal with size N. This algorithm has two shortcomings: it is inefficient under high load because of contention on the lock, and it is not delay-tolerant or fault-tolerant. If a process is delayed or stops while holding the lock, the other processes have to wait for it to complete its invocation, potentially infinitely. This can happen for instance if a process is paused or swapped by the process scheduler, or in an adversary environment, when an evil process does a denial-of-service attack by holding the lock.

Several wait-free algorithms for the renaming problem have been proposed [MA95,AM99,GR10,CRR11,RR11]. In these wait-free algorithms, the lack of progress of one process has no impact on the progress of the others. In this paper, we focus on the first wait-free algorithm for renaming that has been proposed, by Moir and Anderson [MA95]. This algorithm renames up to N processes from an arbitrary large namespace to a namespace of size $\frac{N(N+1)}{2}$ with time complexity $\Theta(N)$. We have actually considered a more demanding variation of the original algorithm: the proven algorithm is adaptive, in the sense that the size of the resulting namespace only depends on the actual number of participants (as opposed to the maximal number of participants for non-adaptive renaming).

3.1 Informal Description

Moir and Anderson algorithm is based on a building block, the splitter (Figure 1). A splitter is used to separate processes. When a process executes the code of a splitter, it eventually receives a value of *stop*, *right* or *down*. The main property is that at most one process ever receives *stop*, not all processes receive *right* and not all processes receive *down*. Simultaneous invocations of a splitter are

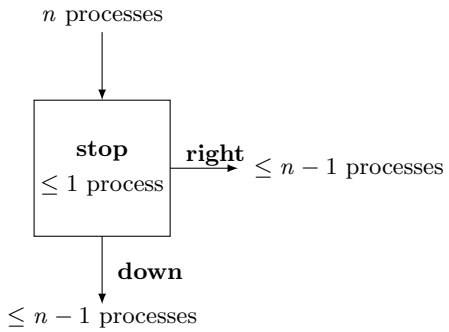

Fig. 1. Specification of a Splitter

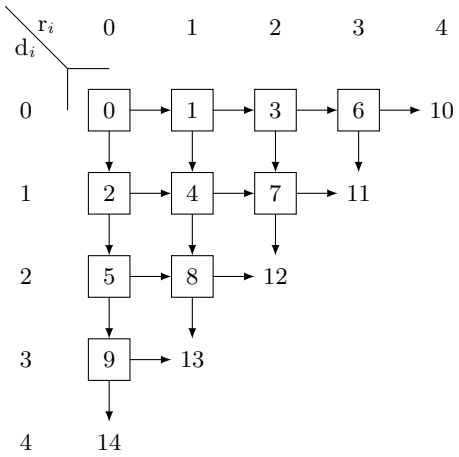

Fig. 2. Structure of Moir-Anderson algorithm for up to 5 Processes

possible, and its code is concurrently executed. Equipped with this splitter, the renaming algorithm consists in a half-grid of splitters (Figure 2).

Every process starts in the $(0, 0)$ box and follows the rule of the splitter it is on. If it receives *stop*, it gets the name on this splitter and ends. If it receives *right* (resp. *down*), it moves to the right (resp. down) splitter. For N processes, a half grid of $\frac{N(N+1)}{2}$ splitters ensures that every process eventually gets a unique name (on the diagonal in the worst case, which it needs N moves to reach, thus time complexity of $\Theta(N)$, splitters being $\Theta(1)$).

3.2 Algorithm

The renaming algorithm is given Figure 3. Each splitter at coordinate (d, r) is implemented with two atomic registers X and Y that are shared by all processes. These registers ensure that reads and writes are atomic under concurrent access.

```
1  initially ∀ d,r ∈ 0..N−1, d+r < N : Y[d][r] = FALSE ∧ X[d][r] = ?
2  rename(id)
3      while (d+r < N−1 ∧ ¬ stop ) do
4          X[d][r] := id;
5          if Y[d][r]
6          then r := r + 1  \∗ right
7          else
8              Y[d][r] := TRUE
9              if (X[d][r] = id)
10             then stop := TRUE \∗ stop
11             else d := d + 1       \∗ down
12             endif
13         endif
14     end while
15     return (1/2)(r + d)(r + d + 1) + d
```

Fig. 3. Code of the Renaming Algorithm

More precisely, an atomic register is a linearizable object that has two operations *read* and *write*. Informally, linearizability means that each operation call appears to take effect instantaneously at some point between its invocation and its response. More rigorously, linearizability means that the object behaves as if all processes agree on a sequential history of operations that is correct with regard to the sequential definition of the object (i.e. a read returns the most recent written value), and such that this history does not reorder completed operations. From a proof point of view, it means that the algorithm behaves as an interleaving of atomic process actions, where an action can read or write at most one shared variable.

Initially, all the Y are false, and the X can hold any value. The variables `d`, `r` and `stop` are local to each process. An arbitrary number of processes (up to N) concurrently call `rename` with their original ids on the large namespace. Each process individually progresses at its own speed. When it gets a *stop* or reaches the diagonal, it computes a return value from its coordinates in the grid, using Cantor pairing function (a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$).

4 Proving the Renaming Algorithm with in-line Splitters

4.1 A Focus on the Splitter

Let's consider one splitter in isolation (Figure 4). A splitter is inherently concurrent, and it cannot be specified as a sequential object with a single operation. This is shown by contradiction. Assume there is a sequential object with a single operation that behaves as a splitter. As it is sequential, there exists a first process to invoke its operation. As this process may be alone, it must obtain *stop* (*right* and *down* are incorrect if it is alone). The next processes can receive either *down* or *right*, without restriction. Such a sequential object has fewer behaviors than a splitter: with a splitter, a concurrent execution allows half the

```
1  initially X = false ∧ X = ?
2  direction(id)
3      X := id
4      if Y then dir := right
5      else Y := true
6          if (X = id) then dir := stop
7          else dir := down
8          endif
9      endif
10     return dir
```

Fig. 4. Implementation of a Splitter with Two Registers

processes to receive down and the other half to receive right, and none to receive stop. For instance, consider two processes that simultaneously enter the code in Figure 4. Process 1 writes register `X` with its id (line 3); process 2 overwrites it with its own id; process 1 tests `Y` and finds it false (line 4); it checks the value of `X` (line 6), does not found its id, and gets *down*; then process 2 checks `Y` (line 4), finds it true, and gets *right*. Thus, a splitter cannot be called as a function, and its code is in-line in the code of the renaming.

Correctness Properties The correctness properties of a splitter are:

Coherency processes are split in at least two categories, and at most one *stop*:

$$\Box \begin{pmatrix} \forall p, q \in Proc, dir_p = stop \wedge dir_q = stop \Rightarrow p = q \\ \wedge\ \exists p \in Proc, dir_p \neq right \\ \wedge\ \exists p \in Proc, dir_p \neq down \end{pmatrix} \quad (1)$$

Termination every correct entering process eventually gets a direction:

$$\forall p \in Proc, \Box(p \text{ enters the splitter} \Rightarrow \Diamond(dir_p \in \{stop, down, right\})) \quad (2)$$

State Space Another difficulty with the splitter (and more generally with wait-free objects) is that the reachable state space is a significant part of the possible valuations of the variables. The direction variable can hold four values (the undetermined value $\perp$, *stop*, *down*, *right*), thus the corresponding state space for N processes is 4^N. The number of valid assignments of directions that respect the property of coherency (1) is[1]:

number of processes	2	3	4	5	6	7	8
possible assignments	16	64	256	1024	4096	16384	65536
valid assignments	13	52	187	646	2185	7288	24055
percentage	81%	81%	73%	63%	53%	44%	38%

[1] The percentage decrease is explained by the progressive dominance of multiple *stop* in the invalid assignments.

NP	distinct states	depth	time
2	98	11	
3	1389	16	
4	17164	21	
5	193115	26	2 s
6	2041458	31	15 s
7	20675305	36	1 min 50 s
8	203055896	41	12 min 24 s
9	1948989879	46	16 h 08 min

NP	distinct states	depth	time
2	142	14	
3	21260	33	1 s
4	6381732	58	56 s
5	5183748425	90	51 h 19 min

Table 1. Model-Checking of a Splitter with registers (left) and of the Renaming Algorithm with registers (right). NP is the number of processes, distinct states is the number of distinct states found by TLC, and depth (or diameter) is the length of the longest execution (ignoring stuttering loops). Times are wall-clock time. Experiments were conducted on a 32 core 2,1 GHz computer with 16 GB.

4.2 An Attempt at Proving the Renaming Algorithm

The renaming algorithm has been specified in TLA$^+$, using PlusCal [Lam09]. The steps have been chosen such that there is at most one read or write of a shared register at each step. This leads to six atomic transitions. Even if the principle and the algorithm are quite simple (13 lines, 2 matrices of shared registers, 2 conditionals and 1 loop), the proof of the uniqueness of the names is not trivial.

Correctness Properties The renaming algorithm must verify:

Coherency processes obtain unique names:

$$\forall p, q \in Proc, p \neq q, \Box(name_p \neq \bot \wedge name_q \neq \bot \Rightarrow name_p \neq name_q) \quad (3)$$

Termination every correct entering process eventually gets a name:

$$\forall p \in Proc, \Box(p \text{ enters the renaming} \Rightarrow \Diamond(name_p \neq \bot)) \quad (4)$$

Model Checking To get an idea of the degree of interleaving, we have model-checked both a splitter and the renaming algorithm with TLC, the TLA$^+$ model checker (Table 1). Observe that model-checking was quickly overwhelmed for the renaming algorithm. Checking with four or five processes is insufficient to get confidence in the correctness of the algorithm. Scaling from four to five processes took a lot of time (more than two days compared with less than one minute) and resources (16 GB memory and up to 110 GB disk space were required).

Proof In the original paper, the correctness proof is given "on paper" and takes 4 pages and 12 invariants. Half of these invariants are considered trivial, and no proofs are given. Mechanically proving these *trivial* invariants was already challenging because of the intricate behavior of the splitters, and it was clear that a complete proof would require too much effort.

5 Proving the Renaming Algorithm with Linearizable Splitters

Seeing that the trivial parts of the proof were not that trivial, we decided to rethink the original algorithm:

- to reduce the interleaving, by reducing the number of step / transitions in the algorithm;
- to hide the internal parts of the splitters, as much as possible;
- to take advantage of the grid architecture (Figure 2) by finding a way to make direct call to the splitters and being able to use theorems on splitters.

5.1 Making the Splitter Linearizable

To reduce interleaving and being able to see the splitter as a black box, the natural idea is the notion of atomic or linearizable objects [HW90]: a linearizable object behaves as if it is accessed sequentially, even in presence of concurrent invocations. Unfortunately, as shown in Section 4.1, a splitter is inherently concurrent, and it cannot be expressed as a linearizable object with one operation. However, the results of [CHQR19] state that the splitter can be transformed in an object with a sequential specification composed of two operations *set* and *get*.

A set / get Splitter The specification of the splitter with two operations (*set* and *get*) is given in Figure 5. Each of the operations is divided in an enabling condition (*setenabled/getenabled*) and a construction of the new state (*set/get*). The *set* operation is enabled if the process is not already in the splitter and registers that the process is entering the splitter. The *get* operation corresponds to a process receiving a direction. The pre-condition of the *get* operation ensures that the answer is valid regarding the specification in Figure 1 and equation (1).

For a variable *spl* and a process $p \in Proc$, a well-formed usage of the module is a sequence of two TLA$^+$ actions: $setenabled(spl, p) \wedge spl' = set(spl, p)$, then $\exists dir \in \{Stop, Down, Right\} : getenabled(spl, p, dir) \wedge spl' = get(spl, p, dir)$.

Proofs of Correctness and Completeness To be useful, this version of the splitter needs to be correct and complete. The correctness is the coherency property (1) and the termination property (2). The completeness means that any correct output can be delivered by the splitter with *set/get*. Indeed, the goal of this version of the splitter is to be used as a black box in the renaming algorithm. The proof of the renaming algorithm will be done with the black box version, and the implementation will use a particular implementation of the splitter, for instance the one in Figure 4. If the black box version is not complete, the proof may omit some pathological cases.

```
MODULE Splitter
CONSTANT Proc  the calling processes
None ≜ "none"
Right ≜ "right"
Stop  ≜ "stop"
Down ≜ "down"
Direction ≜ {None, Right, Stop, Down}
Type ≜ [participants : SUBSET Proc,
        stop : SUBSET Proc, down : SUBSET Proc, right : SUBSET Proc]

new ≜ [participants ↦ {}, stop ↦ {}, down ↦ {}, right ↦ {}]

setenabled(s, pid) ≜ s ∉ s.participants     set not already called
set(s, pid) ≜ [s EXCEPT !.participants = s.participants ∪ {pid}]

getenabled(s, pid, ans) ≜
    ∧ pid ∈ s.participants ∧ pid ∉ (s.stop ∪ s.right ∪ s.down)  set done and get not done
    ∧ ∨ ans = Stop   ∧ s.stop = {}                              valid answers
      ∨ ans = Right  ∧ (s.right ∪ {pid}) ≠ s.participants
      ∨ ans = Down   ∧ (s.down ∪ {pid}) ≠ s.participants
get(s, pid, ans) ≜ CASE
      ans = Stop   → [s EXCEPT !.stop = {pid}]
  □   ans = Right  → [s EXCEPT !.right = s.right ∪ {pid}]
  □   ans = Down   → [s EXCEPT !.down = s.down ∪ {pid}]
```

Fig. 5. The TLA$^+$ *set/get* Specification of a Splitter

Correctness The correctness proof is done assuming that the splitter is well-used, i.e. that *set* is called before *get* and the enabling conditions of each operation are true. The proof of termination (2) is trivial: there is no loop and whatever were the return values to previous processes, a process can always get at least one valid value and thus cannot be blocked in the enabling condition *getenabled*. The proof of coherency (1) has been conducted with TLAPS for any number of processes (file `Splitter_correct_proof.tla` in [HQ19]).

Completeness Completeness of this specification has also been considered. It must be shown that all correct combinations of output values are possible. The predicate $CorrectDirection(Proc, direction)$ states that *direction* is a valid output array satisfying (1), and *dir* is the received values of the processes. Completeness is expressed as (where $\mathbf{EF}P$ is the CTL (computational tree logic) temporal operator stating that there exists a branch where P is eventually true):

$$Completeness \triangleq \forall direction \in [Proc \to Direction] : \\ CorrectDirection(Proc, direction) \Rightarrow \mathbf{EF}(dir = direction)$$

TLA$^+$ is based on LTL (linear temporal logic) and this CTL formula is not checkable. However, the negation of $\mathbf{EF}P$ is $\mathbf{AG}\neg P$, and as P is a state predicate, this corresponds to the LTL invariant $\Box\neg P$. Thus, completeness can be verified in the following way. First, TLC is used to enumerate all the valid arrays of direction. Then, each of them is stated as unreachable ($\Box\neg(\ldots)$), and this property is checked with TLC. A counter-example proves that the state is actually reachable. Optimisations based on symmetry have been introduced, and the completeness of the *set/get* splitter has been verified up to 10 processes (255877 distinct states to check for reachability, reduced to 119 with permutations, 1 h 42 min on a modern quad core laptop).

5.2 The Renaming Algorithm using Linearizable Splitters

The renaming algorithm has been rewritten using *set/get* splitters. The PlusCal version of the renaming problem using *set/get* splitters is given in Figure 6. The translated TLA$^+$ had to be slightly tweaked because the provers have difficulties handling EXCEPT with multi-dimensional arrays, whereas the equivalent form that defines an array is fine[2].

5.3 Proof Sketch of the Coherency Property

The full proof has been conducted with TLAPS and is available online [HQ19]. The line numbers below refer to the file `Renaming.tla` that holds the algorithm and its proof.

Overall Picture The correctness property (3) states that all the processes must have distinct names (`Uniqueness` property in the TLA$^+$ module – line 2027). This uniqueness is guaranteed if all the processes end with different coordinates (`I12` property – line 1955, whose proof (lines 1957–1979) is used to prove the `Uniqueness` property (lines 2030–2040)). A process ends either on the diagonal (condition $d + r < NP - 1$ violated, line 14 in Figure 6) or if it gets *Stop* in a splitter that is not on the diagonal (lines 15–17 in Figure 6). Consider two different processes that get a name (line 28):

- they both stop in a splitter that is not on the diagonal: as at most one process can stop in a splitter, they stop in different splitters and do not have the same coordinates (`StopDifferentProcessesDifferentCoordinates` property – defined line 429 and proved lines 353–434);
- one process stops in a splitter that is not on the diagonal, and the other one stops on the diagonal: they trivially do not have the same coordinates (`StopAndDoneDiffCoord` property – defined line 566 and proved lines 438–574);
- they both stop on the diagonal: this is not trivial and is explained in the following (`IDDX` property – defined line 1851 and proved lines 577–1906).

[2] The construct $[x \text{ EXCEPT } ![e_1] = e_2]$ is a shortcut for $[i \in \text{DOMAIN } x \mapsto \text{IF } i = e_1 \text{ THEN } e_2 \text{ ELSE } x[i]]$. For multi-dimensional arrays, provers work better with the latest.

```
1  ──────────────── MODULE renaming ────────────────
2  EXTENDS Naturals
3  CONSTANT NP                  – number of processes
4  INSTANCE Splitter WITH Proc ← 1 .. NP
5  Coord ≜ 0 .. NP − 1          – coordinate in the grid
7  --algorithm renaming
8  variables spl = [i ∈ Coord ↦ [j ∈ Coord ↦ new]] ;
9  fair process proc ∈ 1 .. NP
10 variables d = 0, r = 0, name = 0 ;
11 begin
12 l0: await setenabled(spl[d][r], self) ;
13     spl[d][r] := set(spl[d][r], self) ;
14 l1: if (d + r < NP − 1) then
15 l2:   either await getenabled(spl[d][r], self, Stop) ;
16               spl[d][r] := get(spl[d][r], self, Stop) ;
17               goto l8 ;
18       or await getenabled(spl[d][r], self, Right) ;
19          spl[d][r] := get(spl[d][r], self, Right) ;
20          r := r + 1 ;
21          goto l0 ;
22       or await getenabled(spl[d][r], self, Down) ;
23          spl[d][r] := get(spl[d][r], self, Down) ;
24          d := d + 1 ;
25          goto l0 ;
26       end either
27   end if ;
28 l8: name := ((r + d) ∗ (r + d + 1) ÷ 2) + d ;
29 end process
30 end algorithm
31 ──────────────────────────────────────────────────
```

Fig. 6. Renaming Algorithm in PlusCal using *set/get* Splitters

.

Number of Participants in a Splitter To prove this last case, the key inductive invariant is that (`NbParticipants` property – line 588):

$$\forall i, j \in \mathit{Coord} : \mathit{Cardinality}(\mathit{splitters}[j][i].\mathit{participants}) = 0 \\ \lor \mathit{Cardinality}(\mathit{splitters}[j][i].\mathit{participants}) \leq NP - (i + j)$$

The first disjunction handles the case where the coordinates are outside the half-grid ($i+j \geq NP$), and the second one handles the case where the coordinates are inside the half-grid ($i + j < NP$).

A simple induction using that processes enter a splitter (i, j) either from the splitter on top $(i - 1, j)$ or from the splitter on left $(i, j - 1)$, and that not all processes can go down or right, gives that there is at most $NP-(i-1+j)-1+NP-(i+j-1)-1$, i.e. $2*(NP-(i+j))$ processes in the splitter (i, j). This basic induction fails. As an example, consider the splitter $(1, 1)$. To receive $NP - 2$ processes from the splitter $(0, 1)$, there must be at least $NP - 1$ processes in the splitter $(0, 1)$. It means that there is at most 1 process in the splitter $(1, 0)$. This process, alone, stops in this splitter and no process comes from the splitter $(1, 0)$ to the splitter $(1, 1)$.

Since a simple induction fails, another invariant is needed (`NbParticipantsBis` property – line 795):

$$\forall i, j \in \mathit{Coord} : \mathit{Cardinality}(\{p \in \mathit{ProcSet} : d[p] \geq j \land r[p] \geq i\}) = 0 \\ \lor \mathit{Cardinality}(\{p \in \mathit{ProcSet} : d[p] \geq j \land r[p] \geq i\}) \leq NP - (i + j)$$

This invariant considers the triangle below and to the right of (i, j), i.e. the triangle with coordinates (i, j), $(NP-1-i, j)$ and $(i, NP-1-j)$. In the following, we refer to this triangle as *the* (i, j) *triangle*.

The proof is done by proving that the property is preserved by all the transitions. For the (i, j) triangle, the two non-trivial cases are when a process in a splitter in column $i - 1$ moves right, or a process in a splitter in line $j - 1$ moves down. In both cases, the number of processes in the (i, j) triangle increases. These two cases are symmetric, and only the first one is discussed.

The intuition behind the proof is shown in Figure 7.

- The induction hypothesis gives that in the $(i - 1, j)$ triangle (green in Figure 7), there are less than $NP - i - j + 1$ processes.
- The fact that *self* can move right ensures that there is at least another process in the splitters including or below the one of *self* (orange in Figure 7). This property (`2InColumnWhenRight` property – line 1380) is proved thanks to another invariant that states that if at one point there is a participant in a splitter, there will always be (at least) a process in the column of the splitter (`AlwaysOneInColumn` property – line 1248). This last one is proved thanks to the correctness of the splitter: not all processes can go right (`EnableRightExistsOtherNotRight` – line 199 in the `Splitter.tla` file).

This means that before *self* moves, there is at most $NP-i-j+1-2$ processes in the (i, j) triangle (blue in the figure 7). So after the transition, there are at most $NP - i - j + 1 - 2 + 1 = NP - i - j$ processes in the (i, j) triangle. QED.

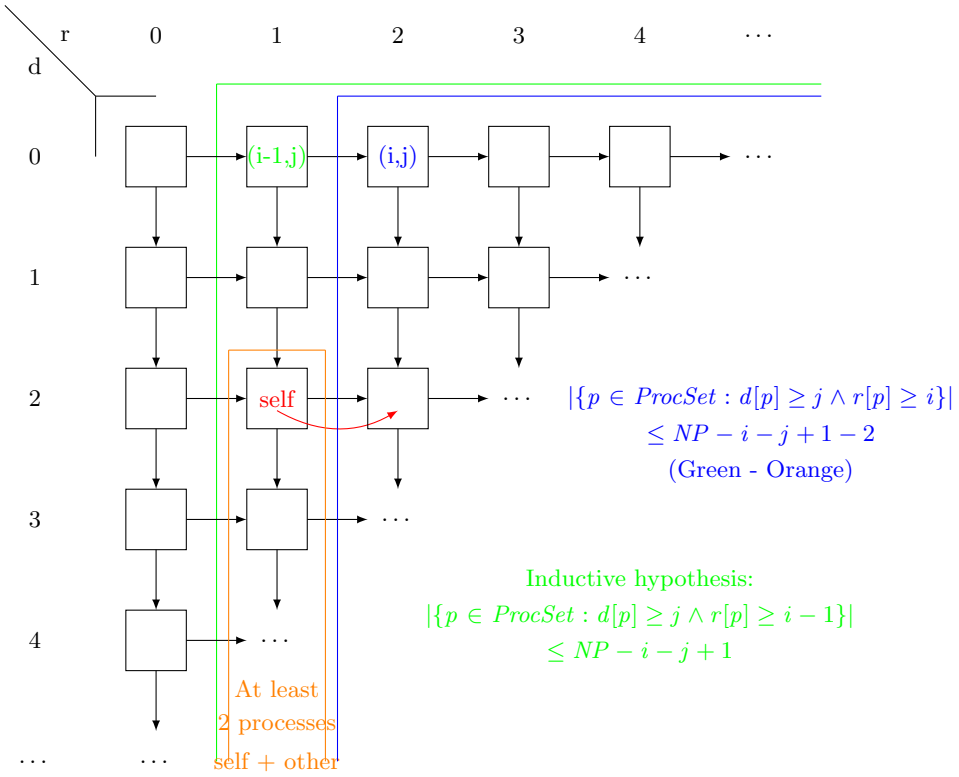

Fig. 7. Intuition behind the Proof of Moir-Anderson Algorithm

Metrics of the Proof The first version was 3000 lines, and after cleaning (factorization into lemma and removing steps not needed by TLAPS), it consists in 2000 lines of TLAPS for the renaming, and 200 lines for the splitter, with a total of 70 lemmas and theorems, and 963 proof steps.

The splitter proofs are composed of 93 proof obligations. Among them, 43 are obvious and discharged by tlapm (the TLA$^+$ proof manager). The other 50 are proved by SMT (we use CVC3, VeriT and Z3).

The renaming proof is composed of 1541 proof obligations. Among them, 914 are trivial and proved by tlapm. Among the 627 left, 108 are proved by Zenon (an automated theorem prover), 35 are proved by LS4 (temporal logic), 475 are proved by SMT and 9 are proved by Isabelle (properties on sets).

To check the proof, TLAPS takes 10 minutes on a quad core modern laptop.

5.4 Proof Sketch of the Termination Property

The second correctness property is the termination (4): every correct process eventually gets a name. To do this, we show by induction that, for each process, (d, r) lexicographically increases until either the sum reaches NP, or the process

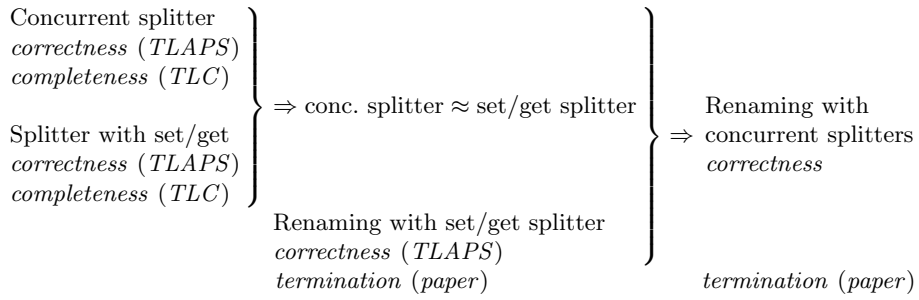

Fig. 8. Summary of the Approach. For each property, the verification method is given (proof with TLAPS, model-checking with TLC, proof on paper).

receives a *Stop* and terminates. Assume a process is at $l0$ (Figure 6, lines 12–13), that its current value of (d, r) is (i, j) and that (d, r) has lexicographically increased until that point, meaning that it is the first time it reaches (i, j). At $l0$, *setenabled* is true as the process has not previously called *set* on the splitter (i, j). The process reaches $l1$ (line 14). At $l1$, the process can go to $l8$ and terminate (QED), or continue to $l2$. At $l2$ (lines 15–27), *getenabled* is true as the process has previously called *setenabled* on this splitter, the process has not previously called *get* on this splitter (first occurrence of (i, j)), and at least one of the choices $\{Stop, Right, Down\}$ is enabled (the specification of a splitter (Figure 5) guarantees that the three sets *stop*, *down* and *right* are disjoint, thus at least one choice is enabled). If the process gets *Stop*, it goes to $l8$ and terminates (QED); if it gets *Down* or *Right*, d or r increases, thus (d, r) becomes $(i + 1, j)$ or $(i, j + 1)$, and so lexicographically increases. The condition in $l1$ eventually becomes false, and the process reaches $l8$ (QED).

TLAPS does not support checking liveness properties, and this proof has not been mechanically verified.

5.5 Back to the Original Algorithm

The last step to prove the original Moir-Anderson algorithm of Figure 3 consists in putting back the concurrent splitters implemented with registers in place of the linearizable splitters with *set/get*. Both versions of the splitter have been proved equivalent by proving that they satisfy the same correctness properties (properties (1) and (2)) and are both complete[3]. For the concurrent splitter, the proof of coherency (1) has been done in TLAPS (file `Splitter_register.tla` in [HQ19]). It consists in 8 lemmas and theorems that prove 17 elementary properties with 30 proof steps. The proof of termination (2) is trivial as it is

[3] Another approach would have been to show a bisimulation between their transition systems. Note that it requires to exhibit a *parameterized* bisimulation, as we have to consider any number of concurrent invocations. We had already proved the properties on the concurrent splitter during our initial attempt at proving the renaming algorithm (Section 4.2) and it seemed simpler to continue onward.

straight-line code. The proof of completeness has been done by model-checking just like for the linearizable splitter (Section 5.1).

Regarding termination of the renaming algorithm with concurrent splitters (4), a similar argument to 5.4 shows that, for each process, (d, r) lexicographically increases until either the sum reaches NP, or the process receives a *Stop* and terminates.

A summary of the approach we have used to prove the orignal Moir-Anderson algorithm is shown in Figure 8.

6 Related Work

Wait-free data structures were introduced by Herlihy [Her91], and then automatically derived from a sequential implementation using universal simulations [Her93]. However, the state of the art universal construction [CER10] is still far too slow compared to the hand-crafted lock-free or lock-based versions. In [TP14], the authors observe that wait-free implementations are notoriously hard to design and often inefficient. One approach consists in designing a lock-free data structure, and then in manually transforming it into a wait-free data structure. [TP14] presents a mechanical transformation of lock-free data structures to wait-free ones. The limits of this work is first, the existence of the lock-free implementation, which has to be written and proven, and secondly, that this implementation must have a normalized form to be convertible.

[GGH05] is a rare example of the certified proof of a lock-free algorithm. The proof of safety has been conducted in PVS, while the proof of progress is on paper. Building on this experience, [GH04] presents a reduction theorem that allows reasoning about a lock-free program to be designed on a higher level than the elementary synchronization primitives. Lamport's refinement mappings are used to prove that the lower-level specification correctly implements the higher-level one. The reduction theorem has been verified with PVS.

Wait-free implementation of tasks have been mechanically proven (e.g. [ORV+10,TSR14,DGH13]). However, to the best of our knowledge, no non-trivial wait-free algorithm built upon wait-free tasks has been mechanically proved. Our intuition for this situation is that proofs cannot be made modular and compositional when using bricks which are inherently concurrent if their internal structure must be visible to take into account this concurrency. Several complex and original algorithms can be found in the literature such as Moir and Anderson renaming algorithm [MA95] that we have considered in this paper, stacks implemented with elimination trees [ST97], lock-free queues with elimination [MNSS05]. In these papers, the correctness proofs are intricate as they must consider the algorithm as a whole, including the tricky part involving wait-free objects, and they have not been mechanically checked. Our approach which exposes a more simple and sequential specification (instead of a complex concurrent implementation) seeks to alleviate this limitation.

Other works have studied the replacement of the complex implementation of part of a system with a simpler one to ease the proof. For example in [GKR+15],

the authors use the notion of deep specification to verify a layer-based architecture. By taking into account the context, a given deep specification may have several implementations which all yield equivalent behaviors. Then, a proof of a whole program can be done using the deep specification of a subprogram (which is simpler than the implementations) and it will stay valid when replacing this subprogram with its implementation. Our approach is similar with a non-layered algorithm, and the set/get specification can be seen as a deep specification, adhering to the coherency property and compatible with the non-linearizable nature of the splitter.

7 Conclusion

Mechanically proving and certifying a wait-free algorithm built upon wait-free objects is a challenging task. An important step is the replacement of the internal wait-free objects with objects having a sequential specification. This allows a compositional approach to the proof, where theorems are proved on the internal objects and reused in the proof of the algorithm. Moreover, this black box view greatly reduces the number of control points in the algorithm, and thus the number of interleavings.

TLA$^+$ has been found well adapted for this task. It natively contains advanced data structures (sets, arrays and maps) with the usual mathematical foundations and notations. Concurrency is intrinsic, both in algorithm description and in reasoning. Tool support is adequate, allowing tests (possibly exhaustively on small models) and proofs. The hierarchical structure advocated by Lamport [Lam12] helps in incrementally building the proofs, by focusing on the important cases of a theorem and delaying the proofs of the obvious or marginal cases. Proving a realistic algorithm such as Moir-Anderson algorithm requires various theories: numbers (coordinates in the grid), sets (cardinality and partitioning), functions (arrays), temporal logic (invariants). The hierarchical structure allows decomposing the proof of a theorem that requires several theories in steps where only one is needed. This in turn allows using the most adequate prover for this theory and simplifies automation of the proof.

References

[AM99] Yehuda Afek and Michael Merritt. Fast, wait-free ($2k$)-renaming. In *18th Annual ACM Symposium on Principles of Distributed Computing*, pages 105–112, 1999.

[CDLM10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA+ proof system. In *5th International Joint Conference on Automated Reasoning, IJCAR 2010*, volume 6173 of *LNCS*, pages 142–148. Springer, 2010.

[CER10] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 335–344, 2010.

[CHQR19] Armando Castaneda, Aurélie Hurault, Philippe Queinnec, and Matthieu Roy. Modular machine-checked proofs of concurrent algorithms built from tasks. In *Submitted to DISC'2019*, 2019.

[CRR11] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. The renaming problem in shared memory systems: An introduction. *Computer Science Review*, 5(3):229 – 251, 2011.

[DGH13] Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *25th Int'l Conf. on Computer Aided Verification, CAV 2013*, volume 8044 of *LNCS*, pages 174–190. Springer, 2013.

[GGH05] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1):21–42, 2005.

[GH04] Hui Gao and Wim H. Hesselink. A formal reduction for lock-free parallel algorithms. In *16th International Conference on Computer Aided Verification*, volume 3114 of *LNCS*, pages 44–56. Springer, 2004.

[GKR+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *42nd ACM Symposium on Principles of Programming Languages*, pages 595–608, 2015.

[GR10] Eli Gafni and Sergio Rajsbaum. Recursion in distributed computing. In *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 6366 of *LNCS*, pages 362–376. Springer, 2010.

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

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[HQ19] Aurélie Hurault and Philippe Quéinnec. TLA+ proof of Moir-Anderson renaming algorithm. http://hurault.perso.enseeiht.fr/RenamingProof, 2019.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[JDK08] Bug JDK-6785442 : ConcurrentLinkedQueue.remove() and poll() can both remove the same element, 2008.

[KS11] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*, chapter Distributed shared memory. Cambridge University Press, March 2011.

[Lam86] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Lam02] Leslie Lamport. *Specifying Systems*. Addison Wesley, 2002.

[Lam09] Leslie Lamport. The PlusCal algorithm language. In *Theoretical Aspects of Computing*, volume 5684 of *LNCS*, pages 36–60. Springer, 2009.

[Lam12] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, 2012.

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

[MNSS05] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free FIFO queues. In *17th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–262, 2005.

[MS98] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[ORV+10] Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *29th ACM Symposium on Principles of Distributed Computing*, pages 85–94, 2010.

[RR11] Sergio Rajsbaum and Michel Raynal. A theory-oriented introduction to wait-free synchronization based on the adaptive renaming problem. In *25th IEEE Int'l Conf. on Advanced Information Networking and Applications*, pages 356–363, 2011.

[ST97] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30(6):645–670, 1997.

[TP14] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '14*, pages 357–368, 2014.

[TSR14] Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. A compositional proof method for linearizability applied to a wait-free multiset. In *11th Int'l Conf. on Integrated Formal Methods, IFM 2014*, volume 8739 of *LNCS*, pages 357–372. Springer, 2014.

A Description of the Artifact

This artifact [HQ19] allows to replicate all the results that we claim in the paper:

- the proof of correctness of the splitter with registers,
- the proof of correctness of the splitter with set/get,
- the way to check the completeness of the splitter with set/get,
- the way to check the completeness of the splitter with registers,
- the proof of correctness of the renaming algorithm using the set/get splitter.

Combining these results implies that the renaming algorithm is correct when used with the original splitter with registers.

A.1 Requirements

Running the artifact relies on the TLA+ Toolbox and the TLA+ Proof System. Both are included in the distributed files, and a script is supplied to install them. The verification of the correctness proofs is achievable on a standard laptop, checking the completeness requires much more resources (at least a quad core with 8 GB of memory, and a 32 core computer with 16 GB was used for the largest numbers of processes).

A.2 Proofs of correctness

The proof of correctness of Moir-Anderson renaming algorithm is built upon the four following files:

- `Splitter.tla`: the set/get splitter with some lemma needed for the correctness proof of the renaming;
- `Splitter_correct_proof.tla`: the proof of correctness of the set/get splitter;
- `Renaming.tla`: the Renaming algorithm and its proof of correctness;
- `Splitter_register.tla`: the splitter based on register and its proof of correctness.

One just has to load them in the toolbox and launch the verification, as explained in the artifact distribution.

A.3 Proofs of completeness

The goal is to verify the completeness of an implementation w.r.t a property P, i.e. that all the valid (w.r.t to P) assignments of the variables are reachable in the implementation.

Our approach consists in two steps: first generate all the valid assignments; then, for each of these assignments, verify that it is reachable. As we rely on TLC, the TLA$^+$ model checker, this approach works for a given number of processes.

1. Choose a number of processes;
2. `generate-states nbproc Splitter_config.tla`
   The module Splitter_config.tla is used to generate all the return values that satisfy the correctness property of the splitter, for `nbproc` processes.
3. `check-reachability vars XXX.tla`
   For each valuation (read from stdin), a module is built with an invariant

   $$\text{INVARIANT } \mathit{Unreachable} \triangleq \mathit{vars} \neq \mathit{valuation}$$

   and it is model-checked with the specification in XXX. It should end with a violation, which proves that this state is reachable.
4. Once all states have been checked, this proves the completeness of XXX for this number of processes.

A script `runall` enumerates the verification of both splitter specifications for 1, 2, 3 etc processes. One can also check a specific size.