Tasks in modular proofs of concurrent algorithms

Armando Castañeda, Aurélie Hurault, Philippe Quéinnec, Matthieu Roy

Tasks in Modular Proofs of Concurrent Algorithms[1]

Armando Castañeda[a], Aurélie Hurault[b], Philippe Quéinnec[b], Matthieu Roy[c]

[a] *Instituto de Matemáticas, UNAM, Mexico.*
[b] *IRIT – Université de Toulouse, France.*
[c] *LAAS, CNRS, Toulouse, France.*

Abstract

Proving correctness of distributed or concurrent algorithms is a mind-challenging and complex process. Slight errors in the reasoning are difficult to find, calling for computer-checked proof systems. In order to build computer-checked proofs with usual tools, such as Coq or TLA$^+$, having sequential specifications of all base objects that are used as building blocks in a given algorithm is a requisite to provide a modular proof built by composition. Alas, many concurrent objects do not have a sequential specification.

This article describes a systematic method to transform any *task*, a specification method that captures concurrent one-shot distributed problems, into a sequential specification involving two calls, `set` and `get`. This transformation allows system designers to *compose* proofs, thus providing a framework for modular computer-checked proofs of algorithms designed using tasks and sequential objects as building blocks.

Moir & Anderson implementation of *renaming* using *splitters* is an iconic example of such algorithms designed by composition, although the actual algorithm is not modular. Moir & Anderson algorithm is adaptive and non-blocking, and, being the assembly of wait-free concurrent objects, the splitters, it resists testing, because of the cost of covering all its states and transitions even with a small input set. Using the `get/set` transformation, a modular description of the algorithm can be obtained. A proof of the modular algorithm has been conducted in TLA$^+$ and verified with TLAPS, the TLA$^+$ Proof System. As far as we know, this is the first time this algorithm is mechanically verified.

Keywords: Formal methods · Verification · Concurrent algorithms · Renaming · Splitter · Linearizability · Distributed tasks · TLA$^+$.

1. Introduction

Context. Fault-tolerant distributed and concurrent algorithms are extensively used in critical systems that require strict guarantees of correctness [25]; consequently, verifying such algorithms is becoming more important nowadays. Yet,

[1] A preliminary version of the paper was published in [9, 23].

proving distributed and concurrent algorithms is a difficult and error-prone task, due to the complex interleavings that may occur in an execution. Therefore, it is crucial to develop frameworks that help assessing the correctness of such systems.

A major breakthrough in the direction of systematic proofs of concurrent algorithms is the notion of *atomic* or *linearizable* objects [21]: a linearizable object behaves as if it is accessed sequentially, even in presence of concurrent invocations, the canonical example being the atomic register. Atomicity lets us model a concurrent algorithm as a transition system in which each transition corresponds to an atomic step performed by a process on a base object. Human beings naturally reason on sequences of events happening one after the other; concurrency and interleavings seem to be more difficult to deal with.

However, it is well understood now that several natural one-shot base objects used in concurrent algorithms cannot be expressed as sequential objects [8, 17, 35] providing a single operation. Such objects are inherently concurrent and any attempt to describe them as sequential objects providing a single operation is incomplete. One-shot means that each process can invoke the object at most once.

An iconic example is the *splitter* abstraction [33], which is the basis of the classical Moir & Anderson renaming algorithm [33]. Intuitively, a splitter is a concurrent one-shot problem that splits calling processes as follows: whenever p processes access a splitter, at most one process obtains stop, at most $p-1$ obtain right and at most $p-1$ obtain down. Moir & Anderson used their splitter abstraction to solve the well-known *renaming* concurrent one-shot distributed problem [4] in which processes are required to pick new distinct names from a given space. Moir & Anderson renaming algorithm uses splitters arranged in a half grid to scatter processes and provide new names to processes. It is worth to mention that, since its introduction almost thirty years ago, the renaming problem has become a paradigm for studying symmetry-breaking in concurrent systems (see, for example, [1, 7]).

A second example is the *exchanger* object provided in Java, which has been used for implementing efficient linearizable elimination stacks [17, 38, 40]. Roughly speaking, an exchanger is a meeting point where pairs of processes can exchange values, with the constraint that an exchange can happen only if the two processes run concurrently.

Splitters and exchangers are instances of one-shot concurrent objects known in the literature as *tasks*. Tasks have played a fundamental role in understanding the computability power of several models, providing a topological view of concurrent and distributed computing [19]. Intuitively, a task is an object providing a single one-shot operation (namely, each process invokes the operation at most once), formally specified through an input domain, an output domain and an input/output relation describing the valid output configurations when a set of processes run concurrently, starting from a given input configuration. Tasks can be equivalently specified by mappings between topological objects: an input simplicial complex (i.e., a discretization of a continuous topological space) modeling all possible input assignments, an output simplicial complex

modeling all possible output assignments, and a carrier map relating inputs and outputs.

Contributions. On the theoretical side, our main contribution is a generic transformation of any task T (with a single operation) into a sequential object S providing two operations, set and get. The behavior of S "mimics" the one of T by splitting each invocation of a process to T into two invocations to S, first set and then get. Intuitively, the set operation records the processes that are participating to the execution of the task. A process actually calls the task and obtains a return value by invoking get. Each of the operations is atomic; however, set and get invocations of a given process may be interleaved with similar invocations from other processes.

We show that these two operations are sufficient for any task, no matter how complicated it may be; since a task is a mapping between simplicial complexes, it can specify very complex concurrent behaviors, sometimes with obscure associated operational semantics.

A main benefit of our transformation is that one can replace an object solving a task T by its associated sequential object S, and reason as if all steps happen sequentially. This allows us to obtain simpler models of concurrent algorithms using solutions to tasks and sequential objects as building blocks, leading to modular correctness proofs.

On the practical side, our main contribution is an application of our transformation to obtain a simple transition system of Moir & Anderson renaming algorithm, which helps to reason about it. Our model is used to derive a full and modular TLA$^+$ proof of the algorithm, the first available mechanized formal proof of it. The proof uses a combination of model-checking and formal proofs to verify correctness and completeness

Organization. Section 2 introduces TLA$^+$ specification language and its associated tools. In Sections 3 to 5, we explain the ideas in Moir & Anderson renaming algorithm that motivated our general transformation. More precisely, Section 3 describes Moir & Anderson algorithm and its expected properties, Section 4 presents a rewording of Moir & Anderson algorithm with sequential objects, and Section 5 details its formal proof with TLA$^+$. Section 6 presents the formal foundations for tasks and sequential objects, and Section 7 describes the general transformation of any task into a sequential object with two operations, as was illustrated with Moir & Anderson algorithm. Lastly, we examine related work in Section 8.

2. TLA$^+$ Specification Language & Tools

2.1. Language

TLA$^+$ [27] 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. Stuttering is enabled in all states, and fairness, usually expressed as a conjunction of weak or strong fairness on actions, or more generally as an LTL property, ensures progress, i.e. that some continuously or infinitely often enabled actions do occur.

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 [28] 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 [10]. 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. Verifying Moir & Anderson Renaming

We consider a concurrent system with n *asynchronous* processes, meaning that each process can experience arbitrarily long delays during an execution. Moreover, processes may *crash* at any time, i.e., permanently stopping taking steps. Before crashing, a crashed process follows its local algorithm (i.e. we do not consider malicious Byzantine failures). Each process is associated with a unique id $\in \mathbb{N}$. The processes can access *base* objects like simple *atomic* read/write registers, or more complex objects.

The original Moir & Anderson renaming algorithm [33] is designed and explained with splitters. Their seminal work first introduces the splitter algorithm based on atomic read/write registers and discusses its properties. Then, they describe a renaming algorithm that uses a grid of splitters. The actual implementation inlines splitters into the code of the renaming algorithm, and their proof is performed on the resulting program that uses solely read/write registers as base objects.

3.1. The Splitter Abstraction

A *splitter* [33] is a one-shot concurrent task in which each process starts with its unique id $\in \mathbb{N}$ and has to return a value satisfying the following properties:

1. Validity. The returned value is right, down, or stop.
2. Splitting. If $p \geq 1$ processes participate in an execution of the splitter, then at most $p - 1$ processes obtain the value right, at most $p - 1$ processes obtain the value down, at most one process obtains the value stop.
3. Termination. Every non-crashed process returns a value.

Notice that if a process runs solo, i.e., $p = 1$, it must obtain stop, since the splitting property holds for any $p \geq 1$.

Figure 1 contains the simple and elegant splitter implementation based on atomic read/write registers from [33]. We refer the reader to [33] for a detailed correctness proof of the algorithm. The fact that the implementation is based on *atomic* registers allows us to obtain a transition system of it in which each transition corresponds to an atomic operation on an object. The benefit of this modelization is that every execution of the implementation is simply described as a sequence of steps, as concurrent and distributed systems are usually modeled (see, for example, [20, 37])).

3.2. Verifying the Splitter Algorithm

Our TLA$^+$ model of the splitter algorithm in Figure 1 and its TLAPS proof are available online (`Splitter_register` in [24]). The correctness properties of the splitter are defined as follows:

Validity:

$$\forall p \in Proc, \Box(dir_p \in \{\bot, stop, down, right\}) \tag{1}$$

```
Shared atomic read/write registers:
      Y = false ∧ X = ⊥
Local (possibly non-atomic) variable per process:
      dir = ⊥

operation splitter():
(01)  X ← id;
(02)  if Y then dir ← right
(03)  else
(04)     Y ← true
(05)     if X = id then dir ← stop
(06)     else dir ← down
(07)     end if
(08)  end if
(09)  return dir
end operation
```

Figure 1: Implementation of a Splitter [33].

Splitting:

$$\Box \left(\begin{array}{l} \quad \forall p, q \in \mathit{Proc}, \mathit{dir}_p = \mathit{stop} \wedge \mathit{dir}_q = \mathit{stop} \Rightarrow p = q \\ \wedge\ \exists p \in \mathit{Proc}, \mathit{dir}_p \neq \mathit{right} \\ \wedge\ \exists p \in \mathit{Proc}, \mathit{dir}_p \neq \mathit{down} \end{array} \right) \tag{2}$$

Termination:

$$\begin{array}{l} \forall p \in \mathit{Proc}, \Box(p \text{ enters the splitter and does not crash} \\ \qquad\qquad \Rightarrow \Diamond(\mathit{dir}_p \in \{\mathit{stop}, \mathit{down}, \mathit{right}\})) \end{array} \tag{3}$$

Although the splitter implementation is very short and simple, its TLA$^+$ proof is long and rather complex — particularly when considering that it uses a boolean register and a plain register only —. Another difficulty with the splitter (and more generally with concurrent algorithms) is that the reachable state space is a significant part of the possible valuations of the variables. This defeats explicit model-checking approaches even for a small number of processes.

3.3. The Renaming Algorithm of Moir & Anderson

M-renaming is a one-shot coordination task [4] in which each process starts with its unique id $\in \mathbb{N}$, and processes are required to return an output name satisfying the following properties:

1. Validity. The output name of a process belongs to $[0, \ldots, M-1]$.
2. Uniqueness. No two processes obtain the same output name.
3. Termination. Every non-crashed process returns an output name.

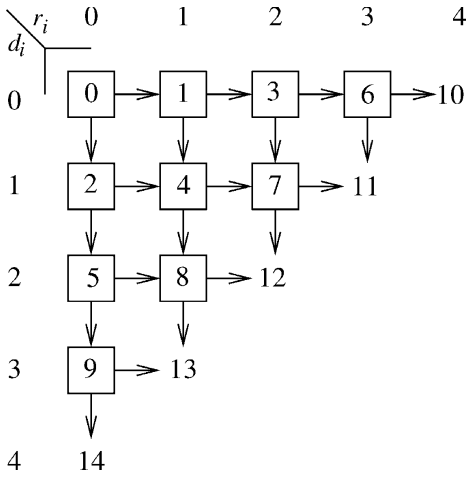

Figure 2: Renaming using Splitters.

Moir and Anderson propose in [33] a read/write renaming algorithm designed using the splitter abstraction. The algorithm is conceptually simple: for up to n processes, a set of $n(n+1)/2$ splitters are placed in a half-grid, each with a unique name, as shown in Figure 2 for $n = 5$. Each process starts invoking the splitter at the top-left corner, following the directions obtained at each splitter. When a splitter invocation returns stop, the process returns the name associated with the splitter. When a process reaches the diagonal, the process gets the name associated with this position.

Although Moir & Anderson renaming algorithm is easily described in a modular way, the actual algorithm is not modular as each splitter in the conceptual grid is replaced by an independent copy of the splitter implementation. Figure 3 depicts the algorithm where each splitter in the grid is inlined. This lack of modularity provokes the correctness proof in [33] to deal with all possible interleavings that can occur, considering all read/write splitter implementations in the grid, throwing away the correctness proof of the splitter implementation.

3.4. A First Attempt to Verify Moir & Anderson Algorithm

The renaming algorithm has been specified in TLA$^+$, using PlusCal. 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 new names is not trivial. The TLA$^+$ model and the TLAPS proofs can be found in [24].

Moir & Anderson renaming algorithm must verify:

Validity:

$$\forall p \in Proc, \Box(name_p \in \{\bot, 0, 1, \ldots, \frac{1}{2}n(n+1) - 1\}) \tag{4}$$

```
initially ∀d, r ∈ {0, 1, ..., n − 1}, Y[d][r] = false ∧ X[d][r] = ⊥
operation rename():
(01)  stop ← false
(02)  d, r ← 0, 0
(03)  while d + r < n − 1 ∧ ¬stop do
(04)      X[d][r] ← id
(05)      if Y[d][r] then r ← r + 1        %% Right
(06)      else
(07)          Y[d][r] ← true
(08)          if X[d][r] = id then stop ← true %% Stop
(09)          else d ← d + 1        %% Down
(10)          end if
(11)      end if
(12)  end while
(13)  return ½(r + d)(r + d + 1) + d
end operation
```

Figure 3: Moir and Anderson Renaming Algorithm [33]. This is actually a slight variation of the original algorithm: it uses here the Cantor pairing function to compute the returned name, to make the algorithm be adaptive.

Uniqueness:

$$\forall p, q \in \mathit{Proc}, p \neq q, \Box(\mathit{name}_p \neq \bot \wedge \mathit{name}_q \neq \bot \Rightarrow \mathit{name}_p \neq \mathit{name}_q) \quad (5)$$

Termination:

$$\forall p \in \mathit{Proc}, \Box(p \text{ enters the renaming and does not crash} \Rightarrow \Diamond(\mathit{name}_p \neq \bot)) \quad (6)$$

To get an idea of the degree of interleaving in Moir & Anderson algorithm, 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. Checking with four or five processes is insufficient to get confidence in the correctness of the algorithm. Scaling from four to five processes took an unreasonable amount of time (going from less than one minute to more than two days) and resources (16 GB memory and up to 110 GB disk space were required).

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. Proving these *trivial* invariants mechanically was already challenging because of the intricate behavior of the splitters, and it was clear that a complete proof would require too much effort.

4. Towards a Modular Moir & Anderson Renaming

In the light of the simple splitter-based conceptual description of Moir & Anderson algorithm, we would like to have a transition system describing the

Model-checking of a Splitter

n	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

Model-checking of M&A Renaming

n	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 (top) and of the Renaming Algorithm with registers (bottom). n is the number of processes, distinct states are 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 of memory space.

algorithm based on splitters as building blocks, in which each step corresponds to a splitter invocation. Such a description would be very beneficial as it would allow us to obtain a modular correctness proof showing that the algorithm is correct as long as the building blocks are splitters, hence the correctness is independent of any particular splitter implementation, hiding the intricate behavior inside a splitter implementation.

4.1. Specifying a Splitter as a Sequential Object

As it is formally proved in Section 7, it is impossible to obtain such a transition system. The obstacle is that a splitter is inherently concurrent and cannot be specified as a sequential object with a single operation. The intuition of the impossibility is the following. By contradiction, suppose that there is a sequential object corresponding to a splitter. Since the object is sequential, in every execution, the object behaves as if it is accessed sequentially (even in presence of concurrent invocation). Then, there is always a process that invokes the splitter object first, which, as noted above, must obtain stop. The rest of the processes can obtain either down or right, without any restriction (the value obtained by the first process precludes that all obtain right or all down). However, such an object is allowing strictly fewer behaviors: in the original splitter definition it

State: Sets *Participants*, *Stop*, *Down*, *Right*, all initialized to $\emptyset$

Function set(id)
  Pre-condition: id $\notin$ *Participants*
  Post-condition: *Participants*$'$ $\leftarrow$ *Participants* $\cup$ {id}
  Output: void
endFunction

Function get(id)
  Pre-condition: id $\in$ *Participants* $\wedge$ id $\notin$ *Stop*, *Down*, *Right*
  Post-condition:
    $D \leftarrow \emptyset$
    if $|Stop| = 0$ **then** $D \leftarrow D \cup \{\mathsf{stop}\}$
    if $|Down| < |Participants| - 1$ **then** $D \leftarrow D \cup \{\mathsf{down}\}$
    if $|Right| < |Participants| - 1$ **then** $D \leftarrow D \cup \{\mathsf{right}\}$
    Let *dec* be any value in D
    if *dec* = stop **then** $Stop \leftarrow Stop \cup \{\text{id}\}$
    if *dec* = down **then** $Down \leftarrow Down \cup \{\text{id}\}$
    if *dec* = right **then** $Right \leftarrow Right \cup \{\text{id}\}$
  Output: *dec*
endFunction

Figure 4: A get/set Sequential Specification of the Splitter.

is perfectly possible that all processes run concurrently and half of them obtain right and the other half obtain down, while none obtains stop.

One can circumvent the impossibility described above by splitting the single method provided by a splitter into two *atomic* operations of a sequential object. Figure 4 presents a sequential specification of a splitter with two operations, set and get, using a standard pre/post-condition specification style. Each process invoking the splitter, first invokes set and then get (always in that order). The idea is that the set operation first records in the state of the object the processes that are participating in the splitter, so far, and then the get operation non-deterministically produces an output to a process, considering the rules of the splitter. Once we have this sequential specification, we can replace each splitter in the original Moir & Anderson splitter-based algorithm with two consecutive calls to an instance of this sequential object, hence obtaining a transition system.

4.2. Verifying get/set *Sequential Splitters*

A sequential specification of a splitter in the TLA$^+$ module using a get/set scheme 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 with regard to the specification of the splitter and equation (2).

```
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}]
```

Figure 5: The TLA$^+$ get/set Specification of a Splitter

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

To be useful, this version of the splitter needs to be *correct* and *complete*. The correctness is the validity, splitting and termination properties in (1), (2) and (3). 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 Moir & Anderson 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 1. If the black box version is not complete, the proof may omit some pathological cases.

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 (3) is trivial: there is no loop and whatever were the values given to previous processes, a process can always get at least one valid value and thus cannot be blocked in the enabling condition getenabled. The proofs of validity (1) and splitting (2) have been conducted with TLAPS

for any number of processes (file `Splitter_correct_proof.tla` in [24]).

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 (2), 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):

$$\begin{aligned} \mathit{Completeness} \triangleq \forall \mathit{direction} \in [\mathit{Proc} \to \mathit{Direction}] : \\ \mathit{CorrectDirection}(\mathit{Proc}, \mathit{direction}) \implies \mathbf{EF}(\mathit{dir} = \mathit{direction}) \end{aligned}$$

TLA$^+$ is based on LTL (linear temporal logic) and this CTL formula with the existential branching operator $\mathbf{E}$ is not checkable. However, the negation of $\mathbf{EF}P$ is $\mathbf{AG}\neg P$, and when 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. Optimizations based on symmetry have been introduced. As Proc set is unordered and process identifiers are not used to select a direction, checking the reachability of a direction array ensures that all its permutations are reachable. 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. Proving Moir & Anderson Renaming with Sequential Splitters

Using the sequential specification of a splitter in Section 4, we can easily obtain a *modular* description of the original Moir & Anderson splitter-based algorithm: each splitter object is replaced with an equivalent sequential version of it, and every process accessing a splitter object *asynchronously* invokes first set and then get, which returns a direction to the process. The resulting algorithm does not rely on any particular splitter implementation, and uses only atomic objects, which allows us to obtain a transition system of it. This is the algorithm that is verified in TLA$^+$. The equivalence between the concurrent splitter specification and the sequential set/get specification implies that our proof also proves the original Moir & Anderson splitter-based algorithm.

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].

[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 latter.

The full proof has been conducted with TLAPS and is available online [24]. The line numbers below refer to the file `Renaming.tla` that holds the algorithm and its proof.

5.1. Proof Sketch of the Validity Property

The correctness property (4) constrains the name to be in the space $\{0, 1, \ldots, \frac{1}{2}n(n+1)-1\}$. An auxiliary invariant that bounds $d+r$ is defined : $\forall p \in Proc : \Box(0 \leq d[p]+r[p] \leq n-1)$ (`I5` property, line 171). It is easily proved by induction considering the condition at line 14 in Figure 6 (lines 294–311). With `I5`, the `Validity` property is proved with an arithmetic argument (lines 2044–2104).

5.2. Proof Sketch of the Uniqueness Property

Overall Picture. The correctness property (5) states that all the processes get 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 $d + r < n - 1$ 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):

- both processes 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);
- both processes stop on the diagonal: this is not trivial and is explained in the following (`IDDX` property – defined line 1851 and proved lines 577–1906).

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

$$\begin{aligned}\forall i, j \in Coord : \; & Cardinality(splitters[j][i].participants) = 0 \\ & \vee \; Cardinality(splitters[j][i].participants) \leq n - (i + j)\end{aligned}$$

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

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 $n - (i - 1 + j) - 1 +$

```
MODULE renaming
EXTENDS Naturals
CONSTANT n                    – number of processes
INSTANCE Splitter WITH Proc ← 1 .. n
Coord ≜ 0 .. n − 1            – coordinate in the grid
--algorithm renaming
variables spl = [i ∈ Coord ↦ [j ∈ Coord ↦ new]] ;     – shared variables
fair process proc ∈ 1 .. n
variables d = 0, r = 0, name = 0 ;          – local variables of each process
begin
l0: await setenabled(spl[d][r], self) ;
    spl[d][r] := set(spl[d][r], self) ;
l1: if (d + r < n − 1) then
l2:   either await getenabled(spl[d][r], self, Stop) ;
                spl[d][r] := get(spl[d][r], self, Stop) ;
                goto l8 ;
      or await getenabled(spl[d][r], self, Right) ;
          spl[d][r] := get(spl[d][r], self, Right) ;
          r := r + 1 ;
          goto l0 ;
      or await getenabled(spl[d][r], self, Down) ;
          spl[d][r] := get(spl[d][r], self, Down) ;
          d := d + 1 ;
          goto l0 ;
      end either
  end if ;
l8: name := ((r + d) ∗ (r + d + 1) ÷ 2) + d ;
end process
end algorithm
```

Figure 6: Renaming Algorithm in PlusCal using set/get Splitters

.

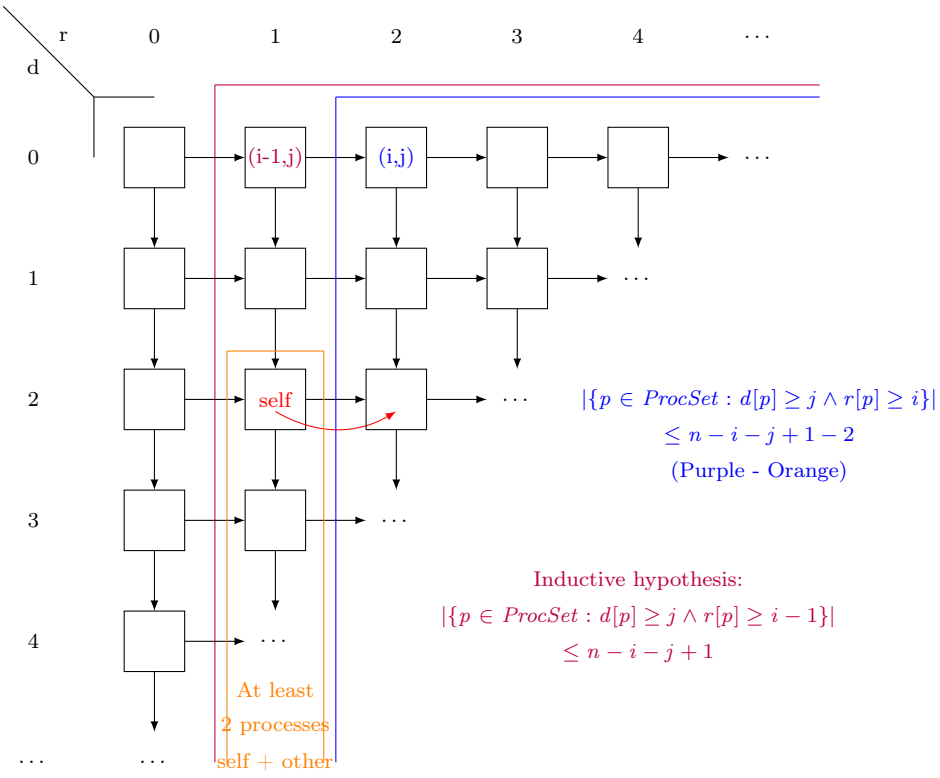

Figure 7: Intuition behind the Proof of Moir & Anderson Algorithm

$n-(i+j-1)-1$, i.e. $2*(n-(i+j))$ processes in the splitter (i,j). This basic induction fails. As an example, consider the splitter $(1,1)$. To receive $n-2$ processes from the splitter $(0,1)$, there must be at least $n-1$ 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):

$$\forall i,j \in Coord : Cardinality(\{p \in ProcSet : d[p] \geq j \wedge r[p] \geq i\}) = 0 \\ \vee\ Cardinality(\{p \in ProcSet : d[p] \geq j \wedge r[p] \geq i\}) \leq n - (i+j)$$

This invariant considers the triangle below and to the right of (i,j), i.e. the triangle with coordinates (i,j), $(n-1-i,j)$ and $(i,n-1-j)$. 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 (purple in Figure 7), there are less than $n - i - j + 1$ 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 $n-i-j+1-2$ processes in the (i, j) triangle (blue in the figure 7). So after the transition, there are at most $n - i - j + 1 - 2 + 1 = n - i - j$ processes in the (i, j) triangle. This concludes the proof.

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.

5.3. Proof Sketch of the Termination Property

The third correctness property is termination (6): every non-crashed 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 n, or the process 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

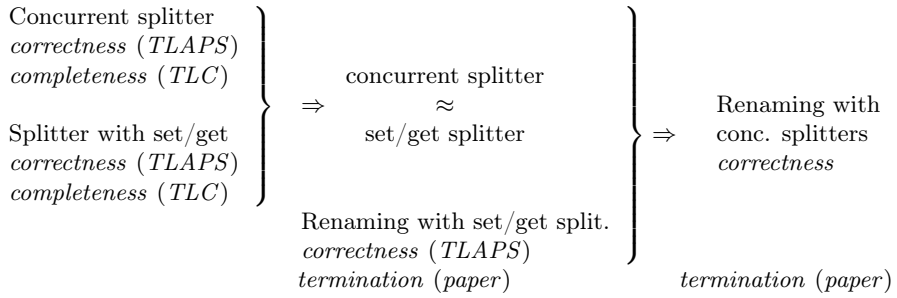

Figure 8: Summary of the Approach. For each property, the verification method is given (mechanized proof with TLAPS, model-checking with TLC, proof on paper).

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.

5.4. 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), (2), and (3)) and are both complete[3]. For the concurrent splitter, the proof of the validity (1) and splitting (2) properties has been done in TLAPS (file `Splitter_register.tla` in [24]). It consists in 8 lemmas and theorems that prove 17 elementary properties with 30 proof steps. The proof of termination (3) is trivial as it is straight-line code. The proof of completeness has been done by model-checking like for the linearizable splitter, up to 7 processes (Section 4.2).

Regarding termination of the renaming algorithm with concurrent splitters (6), a similar argument to 5.3 shows that, for each process, (d, r) lexicographically increases until either the sum reaches n, or the process receives a *Stop* and terminates.

A summary of our approach to prove the original Moir & Anderson algorithm is shown in Figure 8.

[3]Another approach would have been to show a bisimulation between their transition systems [32]. 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 3) and it seemed simpler to continue onward.

6. Tasks and Sequential Objects

In the next two sections, we show that the transformation in Section 4 of the splitter task into a sequential object with two operations, get and set, is not a trick but rather a general methodology to deal with tasks without a sequential specification. Our get/set solution proposed here is reminiscent to the *request-follow-up* transformation in [39] that allows to transform a *partial* method of a sequential object (e.g. a queue with a blocking dequeue method when the queue is empty) into two *total* methods: a total request method registering that a process wants to obtain an output, and a total follow-up method obtaining the output value, or *false* if the conditions for obtaining a value are not yet satisfied (the process invokes the follow-up method until it gets an output). We stress that the *request-follow-up* transformation [39] considers only objects with a sequential specification and is not shown to be general as it is only used for queues and stacks.

6.1. Model of Computation in Detail

We consider a standard concurrent system with n *asynchronous* processes, denoted $p_1, \ldots, p_n$, which may *crash* at any time during an execution of the system, i.e., stopping taking steps (for more detail see for example [20, 37]). Before crashing, a crashed process follows its local algorithm (i.e. we do not consider malicious Byzantine failures). Processes communicate with each other by invoking operations on shared, concurrent *base objects*. A base object can provide Read/Write operations (also called *register*), more powerful operations, such as Test&Set, Fetch&Add, Swap or Compare&Swap, or solve a concurrent distributed problem, for example, Splitter, Renaming or Set_Agreement.

Each process follows a local state machines $A_1, \ldots, A_n$, where A_i specifies which operations on base objects p_i executes in order to return a response when it invokes a high-level operation (e.g. push or pop operations). Each of these base-objects operation invocations is a *step*. An *execution* is a possibly infinite sequence of steps and invocations and responses of high-level operations, with the following properties:

1. Each process first invokes a high-level operation, and only when it has a corresponding response, it can invoke another high-level operation, i.e., executions are *well-formed*.
2. For any invocation $inv(\langle \mathsf{opType}, p_i, input \rangle)$ of a process p_i, the steps of p_i between that invocation and its corresponding response (if there is one), are steps that are specified by A_i when p_i invokes the high-level operation $\langle \mathsf{opType}, p_i, input \rangle$.

A high-level operation in an execution is *complete* if both its invocation and response appear in the execution. An operation is *pending* if only its invocation appears in the execution. A process is *non-crashed* in an execution if it takes infinitely many steps.

6.2. The Linearizability Formalism

A central paradigm for specifying distributed problems is that of a shared object that processes may access concurrently [20, 37], but the object is defined in terms of a sequential specification, i.e., an automaton describing the outputs the object produces when it is accessed sequentially.

A *sequential object* X is specified by a (not necessarily finite and possibly non-deterministic) Mealy state machine (Q, Inv, Res, δ), where Inv is the set with all possible invocations to the object and Res is the set with all possible responses from the object. The responses are determined both by its current state $s \in Q$ and the current input $in \in Inv$. If X is in state q and it receives as input an invocation $in \in Inv$ by process p, then, if $(q', r) \in \delta(q, in)$, the meaning is that X may return the response r to the invocation in by process p, and move to state q'. Notice that there may be several possible responses (if the object is non-deterministic), however, it is convenient to assume that the next state q' is uniquely determined by the response r, namely, if $(q', r), (q'', r) \in \delta(q, in)$, then we have $q' = q''$. Also, it is convenient to require that the object X is *total*, meaning that for any state q, $\delta(q, in) \neq \emptyset$, for all $in \in Inv$.

For any sequence of invocations $in_0, \ldots, in_m$, a *sequential execution of* X starting in q_0 is

$$q_0, in_0, r_0, q_1, in_1, r_1, \ldots, q_m, in_m, r_m$$

where q_0 is an initial state of X, and $(q_{i+1}, in_{i+1}) \in \delta(q_i, in_i)$. However, given that we require that the object's response at a state uniquely determines the new state, we may denote the execution by

$$in_0, r_0, in_1, r_1, \ldots, in_m, r_m,$$

because the sequence of states $q_1, \ldots, q_m$ is uniquely determined by q_0, and by the sequences of invocations and responses. Without loss of generality we only consider sequential automata with a single initial state for each object.

The *sequential specification* of an object X, $SSpec(X)$, is the set of all its sequential executions. Notice that $SSpec(X)$ is *prefix-closed*: if an execution is in $SSpec(X)$, so is the execution obtained by removing the last invocation and its response.

Figure 9 presents a sequential specification of the well-known Test&Set object, which has been used in a large number of concurrent algorithms (see for example [20, 37]); the specification is presented in the usual pre/post-condition specification style. Intuitively, the object is initialized to 0 and the first invocation obtains 0 (the winner) and the rest obtain 1 (the losers).

Once we have a sequential specification, there are various ways of defining what it means for an execution to *satisfy* an object, namely, that it respects the sequential specification. *Linearizability* [21] is the standard notion used to identify correct executions of implementations of sequential objects. Intuitively, an execution is linearizable if its operations can be ordered sequentially, without reordering non-overlapping operations, so that their responses satisfy the specification of the implemented object. To formalize this notion we define a partial order on the completed operations of an execution E: $\mathsf{op} <_E \mathsf{op}'$ if and

State: Integer X initialized to 0

Function Test&Set()
  Pre-condition: none
  Post-condition:
    $temp \leftarrow X$
    $X' \leftarrow 1$
  Output:
    $temp$
endFunction

Figure 9: Sequential Specification of Test&Set.

only if $res(\mathsf{op})$ precedes $inv(\mathsf{op}')$ in E. Two operations are *concurrent* if they are incomparable by $<_E$. The execution is *sequential* if $<_E$ is a total order.

Definition 1. *An execution E is* linearizable *with respect to X if there is a sequential execution S of X (i.e., $S \in SSpec(X)$) such that*

1. *S contains every completed operation of E and might contain some pending operations. Inputs and outputs of invocations and responses in S agree with inputs and outputs in E.*
2. *For every two completed operations* op *and* op′ *in E, if* op $<_E$ op′*, then* op *appears before* op′ *in S.*

Using the linearizability correctness criteria for sequential objects we can define the set of *valid* executions for X, denoted $VE(X)$. Arguably, the set $VE(X)$ contains the behavior one might expect from a *building-block* (e.g. an algorithm) that implements X (i.e. all its executions are linearizable w.r.t. X).

$VE(X) = \{E | E \text{ has only invocations and responses and is linearizable w.r.t. } X\}$

6.3. The Task Formalism

6.3.1. Definition of a Task

A task is the basic distributed equivalent of a function in sequential computing, defined by a set of inputs to the processes and for each (distributed) input to the processes, a set of legal (distributed) outputs of the processes, e.g., [19]. In an algorithm designed to solve a task, each process starts with a private input value and has to eventually decide irrevocably on an output value. A process p_i is initially not aware of the inputs of other processes. Consider an execution where only a subset of $k \leq n$ processes participate; the others crash without taking any steps. A set of pairs $\sigma = \{(\mathrm{id}_1, x_1), \ldots, (\mathrm{id}_k, x_k)\}$ is used to denote the input values, or output values, in the execution, where x_i denotes the value of the process with identity id_i, either an input value or an output value.

A set σ as above is called a *simplex*, and if the values are input values, it is an *input simplex*, if they are output values, it is an *output simplex*. The elements of σ are called *vertices*. An *input vertex* $v = (\mathrm{id}_i, x_i)$ represents the initial state of process id_i, while an *output vertex* represents its decision. The *dimension* of

a simplex σ, denoted $dim(\sigma)$, is $|\sigma| - 1$, and it is *full* if it contains n vertices, one for each process. A subset of a simplex, which is a simplex as well, is called a *face*. Since any number of processes may crash, simplexes of all dimensions are of interest, for taking into account executions where only processes in the simplex participate. Therefore, the set of possible input simplexes forms a *complex* because its sets are closed under containment. Similarly, the set of possible output simplexes also form a complex.

More generally, a *complex* $\mathcal{K}$ is made of a set of vertices $V(\mathcal{K})$, and a set of simplexes (i.e. a set of sets), each simplex being a finite, nonempty subsets of $V(\mathcal{K})$, satisfying: (1) if $v \in V(\mathcal{K})$ then $\{v\}$ is a simplex of $\mathcal{K}$, and (2) if σ is a simplex of $\mathcal{K}$, so is every nonempty subset of σ. The dimension of $\mathcal{K}$ is the largest dimension of its simplexes, and $\mathcal{K}$ is *pure* of dimension k if each of its simplexes is a face of a k-dimensional simplex. In distributed computing, the simplexes (and complexes) are often *chromatic*, since each vertex v of a simplex is labeled with a distinct process identity. The set of processes identities in an input or output simplex σ is denoted $\mathrm{id}(\sigma)$.

Definition 2 (Task)**.** *A* task *T for n processes is a triple $(\mathcal{I}, \mathcal{O}, \Delta)$ where $\mathcal{I}$ and $\mathcal{O}$ are pure chromatic $(n-1)$-dimensional complexes, and Δ maps each simplex σ from $\mathcal{I}$ to a subcomplex $\Delta(\sigma)$ of $\mathcal{O}$, satisfying:*

1. *$\Delta(\sigma)$ is pure of dimension $dim(\sigma)$,*
2. *For every τ in $\Delta(\sigma)$ of dimension $dim(\sigma)$, $\mathrm{id}(\tau) = \mathrm{id}(\sigma)$,*
3. *If σ, σ' are two simplexes in $\mathcal{I}$ with $\sigma' \subset \sigma$ then $\Delta(\sigma') \subset \Delta(\sigma)$.*

A task has only one operation, let us call it $\mathsf{task}()$, which process id_i may call with value x_i only if (id_i, x_i) is a vertex of $\mathcal{I}$. The operation $\mathsf{task}(x_i)$ may return y_i to the process only if (id_i, y_i) is a vertex of $\mathcal{O}$. A task is a very compact way of specifying a distributed problem, and indeed typically it is hard to understand what exactly is the problem being specified. Intuitively, Δ specifies, for every simplex $\sigma \in \mathcal{I}$, the valid outputs $\Delta(\sigma)$ for the processes in $\mathrm{id}(\sigma)$ assuming they run to completion, and the other processes crash initially, and do not take any steps.

As with other frameworks for specifying concurrent objects (e.g. linearizability for sequential specifications), tasks have their own correctness criteria that defines the executions satisfying a given task. Let E be an execution where each process invokes a task $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ once. Then, σ_E is the input simplex defined as follows: (id_i, x_i) is in σ_E iff in E there is an invocation of $\mathsf{task}(x_i)$ by process id_i. The output simplex τ_E is defined similarly: (id_i, y_i) is in τ_E iff there is a response y_i to a process id_i in E. We say that E *satisfies* $(\mathcal{I}, \mathcal{O}, \Delta)$ if for every prefix E' of E, it holds that $\tau_{E'} \in \Delta(\sigma_{E'})$. Note that it might be the case that $dim(\tau_{E'}) \leq dim(\sigma_{E'})$.

The prefix requirement prevents executions that globally seem correct, but in a prefix a process predicts future invocations. This requirement has been implicitly considered in the past by stating that an algorithm solves a task if any of its executions agree with the task specification.

Using the satisfiability notion of tasks we can now consider the set of valid executions, $VE(T)$, for a given task $T = (\mathcal{I}, \mathcal{O}, \Delta)$. Arguably, the set $VE(T)$ contains the behavior one might expect from a *building-block* (e.g. an algorithm) that implements T.

$$VE(T) = \{E | E \text{ has only invocations and responses and satisfies } T\}$$

6.3.2. The Splitter Task

As an example consider the *splitter* task [33] defined informally as follows. Each process invokes splitter with its id as input and outputs stop, down or right. For every $0 < k \leq n$, it is required that if k processes invoke the splitter (note necessarily concurrently), at most one process outputs stop, at most $k-1$ output down and at most $k-1$ output right. Formally, the splitter task $T_{\mathsf{spl}} = (\mathcal{I}_{\mathsf{spl}}, \mathcal{O}_{\mathsf{spl}}, \Delta_{\mathsf{spl}})$ is defined as:

1. The vertices of the input complex $\mathcal{I}_{\mathsf{spl}}$ are all pairs of the form $(\mathrm{id}_i, \mathrm{id}_i)$, for every ID process id_i.
2. $\mathcal{I}_{\mathsf{spl}}$ is the complex made of the $(n-1)$-dimensional simplex $\{(\mathrm{id}_1, \mathrm{id}_1), \ldots, (\mathrm{id}_n, \mathrm{id}_n)\}$ (and all its faces), with all distinct id processes $\mathrm{id}_1, \ldots, \mathrm{id}_n$.
3. The vertices of the output complex $\mathcal{O}_{\mathsf{spl}}$ are all pairs of the form $(\mathrm{id}_i, \mathsf{stop})$, $(\mathrm{id}_i, \mathsf{down})$ and $(\mathrm{id}_i, \mathsf{right})$ for every ID process id_i.
4. Given a simplex $\tau = \{(\mathrm{id}_1, y_1), \ldots, (\mathrm{id}_m, y_m)\}$ with vertices in $\mathcal{O}_{\mathsf{spl}}$ and an integer k, let $SP(\tau, k)$ be the *splitter predicate* that holds only if
   (a) all id_is are distinct,
   (b) $|Stop| \leq 1$, $|Down| \leq k-1$ and $|Right| \leq k-1$, where $Stop = \{\mathrm{id}_i | y_i = \mathsf{stop}\}$, $Down = \{\mathrm{id}_i | y_i = \mathsf{down}\}$ and $Right = \{\mathrm{id}_i | y_i = \mathsf{right}\}$.
5. $\mathcal{O}_{\mathsf{spl}}$ contains every $(n-1)$-dimensional simplex τ (and all its faces), such that $SP(\tau, n)$ holds.
6. For every $(k-1)$-dimensional input simplex σ, $\Delta_{\mathsf{spl}}(\sigma)$ contains every $(k-1)$-dimensional output simplex τ (and all its faces) such that $\mathrm{id}(\tau) = \mathrm{id}(\sigma)$ and $SP(\tau, k)$ holds.

Figure 10 shows a graphic description of the splitter task for three processes with IDs 1, 2 and 3. Since we have only three processes, the input and output complexes are of dimension two, namely, they are made of vertices, edges and triangles. The input complex, shown at the left, consists of a triangle and all its faces (i.e. vertices and edges); the complex has a single input vertex for a process because each process starts with its ID in every execution. The output complex, at the right, contains all possible valid output simplexes, namely, in each of its simplexes at most one vertex has stop and not all vertices in the simplex have down/right (hence, for example, the triangle with all right outputs is not in the complex). The Δ function maps the input vertex with ID 1 to the output vertex $(1, \mathsf{stop})$ specifying that if process with ID 1 runs *solo*, it must return stop. Δ maps the input edge with IDs 1 and 2 to the cycle made of the solid and dashed bold edges of the output complex (formally, a subcomplex

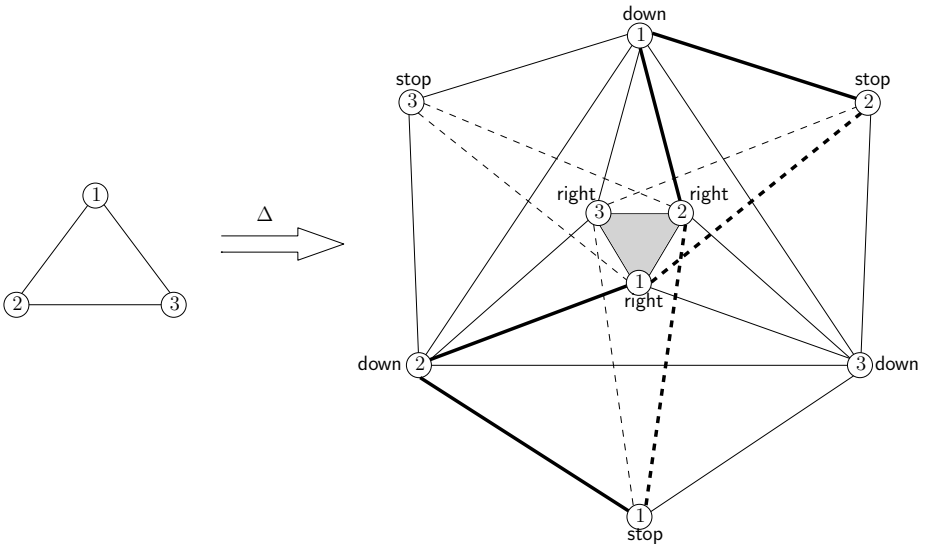

Figure 10: The Splitter Task for Three Processes.

of the output complex), which specifies that if processes with IDs 1 and 2 run *concurrently*, they can obtain the outputs in any of these edges. The input triangle is mapped to the whole output complex. The rest of Δ is defined symmetrically.

6.3.3. The Exchanger Task

A second interesting example is the Java *exchanger* object which is informally defined as follows in the Java documentation:

> *A synchronization point at which threads can pair and swap elements within pairs. Each thread presents some object on entry to the exchange method, matches with a partner thread, and receives its partner's object on return.*

Clearly the object is informally specified in terms of concurrent executions, very much in the style of the task formalism.

Exchangers have been used in [18] to implement a concurrent stack, and the lack of a sequential specification of exchangers makes the proof in that paper intricate. They have also been used in a number of concurrent implementations, e.g. [38, 40]. More precisely, in [40], Shavit and Touitou present the implementation of pools and stacks with *elimination trees*, a form of *diffracting trees* [41] which achieves high efficiency at high contention levels. A simplified version of their algorithm is the following. There are two kinds of opposite requests: enqueue and dequeue for a stack. The structure is constructed from *elimination balancers* that are connected to one another to form a balanced binary tree. Each leaf of the tree holds a standard concurrent stack implementation (e.g.

with locks). Each internal node of the tree holds a *prism* and an *exchanger*. The prism has an internal state (0 or 1) and two outputs labeled 0 and 1. It routes a request according to this state: an enqueue request goes on the output labeled as the internal state, a dequeue request goes on the output labeled as the inverse of the internal state. The internal state is flipped after each request. This allows the requests to spread on the tree while ensuring that a dequeue follows the same path as the most recent enqueue. To speed things up and to avoid contention of the internal state, two mechanisms are added. First, two concurrent requests of the same kind are directly routed on both output without changing the internal state. Secondly, an *exchanger* is used to pair opposite requests: when both an enqueue and a dequeue are present, they are matched, they swap their values and they directly exit the tree without being further propagated (observe that this version of the exchanger is slightly different than the one above as processes exchange opposite requests). The actual implementation uses an array of prisms to avoid the bottleneck of the root and first-levels balancers, however this does not change the overall specification of the algorithm.

Although there is no sequential specification of the exchanger in the literature (a proof such as the one for Lemma 1 shows that there does not exist such a specification), one can succinctly define it as a task. Intuitively, for processes to exchange values, an exchanger matches pairs of processes, with the possibility that some processes are unmatched (marked as matched with a default value denoted $\perp$). The exchanger task $T_{\mathsf{exc}} = (\mathcal{I}_{\mathsf{exc}}, \mathcal{O}_{\mathsf{exc}}, \Delta_{\mathsf{exc}})$ is defined as follows.

1. The vertices of the input complex $\mathcal{I}_{\mathsf{exc}}$ are all pairs of the form $(\mathrm{id}_i, \mathrm{id}_i)$, for every ID process id_i.
2. $\mathcal{I}_{\mathsf{exc}}$ is the complex made of the n-dimensional simplex $\{(\mathrm{id}_1, \mathrm{id}_1), \ldots, (\mathrm{id}_n, \mathrm{id}_n)\}$ (and all its faces), with all distinct id processes $\mathrm{id}_1, \ldots, \mathrm{id}_n$.
3. The vertices of the output complex $\mathcal{O}_{\mathsf{exc}}$ are all pairs $(\mathrm{id}_i, \mathrm{id}_j)$ and $(\mathrm{id}_i, \perp)$, where id_i and id_j are distinct process IDs.
4. Given a simplex $\tau = \{(\mathrm{id}_1, y_1), \ldots, (\mathrm{id}_m, y_m)\}$ with vertices in $\mathcal{O}_{\mathsf{exc}}$, let $EX(\tau)$ be the *exchanger predicate* that holds only if
   (a) all id_i's are distinct,
   (b) id_i is matched with a different process or not matched at all: $y_i \in \{\mathrm{id}_1, \ldots, \widehat{\mathrm{id}}_i, \ldots, \mathrm{id}_m, \perp\}$, where circumflex ($\widehat{\ }$) denotes omission,
   (c) id_i is matched with at most one process, namely, it appears in a second entry at most once,
   (d) matches are consistent, i.e., if $y_i = \mathrm{id}_j$ then $y_j = \mathrm{id}_i$.
5. $\mathcal{O}_{\mathsf{exc}}$ contains every n-dimensional simplex $\tau = \{(\mathrm{id}_1, y_1), \ldots, (\mathrm{id}_n, y_n)\}$ (and all its faces) such that $EX(\tau)$ holds.
6. For every $(k-1)$-dimensional input simplex σ, $\Delta_{\mathsf{exc}}(\sigma)$ contains every $(k-1)$-dimensional output simplex τ (and all its faces) such that $\mathrm{id}(\tau) = \mathrm{id}(\sigma)$ and $EX(\tau)$ holds.

The exchanger task for three processes with IDs 1, 2 and 3 is depicted in Figure 11. Δ maps the input vertex with ID i to the output vertex $(i, \perp)$

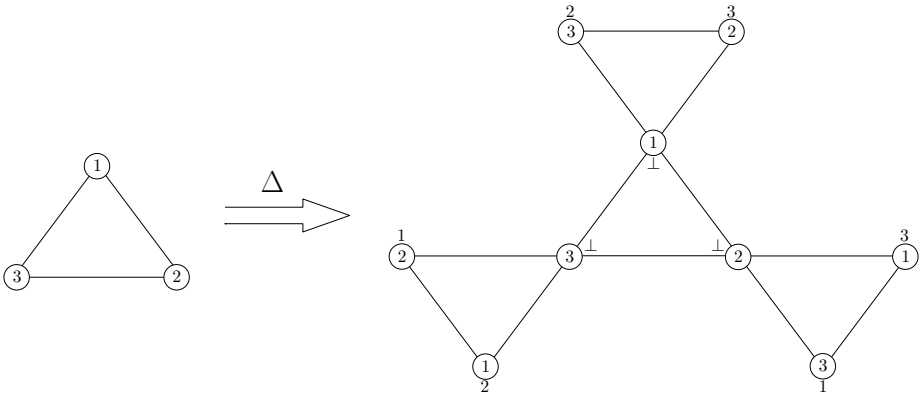

Figure 11: The Exchanger Task for Three Processes.

indicating that if process with ID i runs *solo*, it is unmatched. The edge of the input complex with IDs i and j are mapped to the subcomplex made of the two output edges $\{(i,\perp),(j,\perp)\}$ and $\{(i,j),(j,i)\}$, which specifies that if the two processes run *concurrently*, they can either be matched or unmatched. The input triangle is mapped to the whole output complex.

7. Dealing with Tasks without a Sequential Specification

Intuitively, tasks and sequential specifications are inherently different paradigms for specifying distributed problems: while a task specifies what a set of processes might output when running concurrently, a sequential specification specifies the behavior of a concurrent object when accessed sequential (and linearizability tells when a concurrent execution "behaves" like a sequential execution of the object).

7.1. Impossibility of Modeling a Task as a Sequential Object with a Single Operation

A natural question is if any task can be modeled as a sequential object with a single operation, namely, the object defines the same set of valid executions. A well-known example for which this is possible is the consensus distributed coordination problem that can be equivalently defined as a task or as a sequential object (see for example [20] where it is defined as an object[4] and [19] where it is defined as a task). Another interesting example is the Test&Set atomic operation that is typically specified through a sequential object, however it can also be specified as a task. Figure 12 depicts the Test&Set task for three processes (the specification in Figure 9 is not one-shot but it can be easily made one-shot

[4]Sometimes the object is defined with two operations (in the style of the Theorem 2), however, consensus can be equivalently defined with one operation.

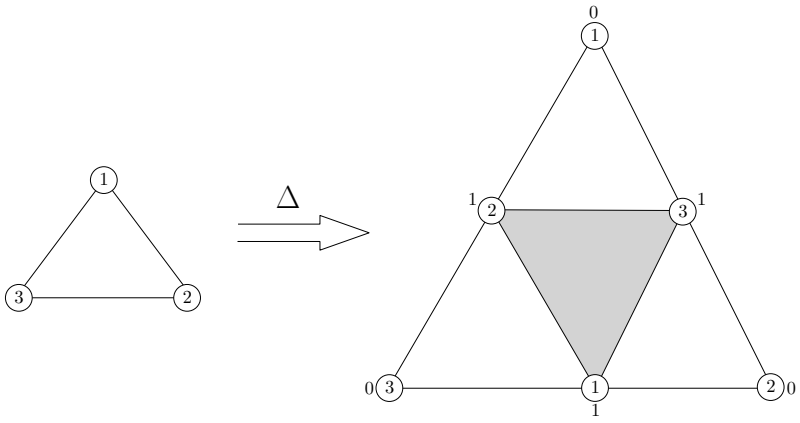

Figure 12: The one-shot Test&Set Object for Three Processes Modeled as Task.

by adding that restriction in the pre-condition). In general, this is not the case, as the following result shows.

Lemma 1. *Consider the splitter task* $T_{\mathsf{spl}} = (\mathcal{I}_{\mathsf{spl}}, \mathcal{O}_{\mathsf{spl}}, \Delta_{\mathsf{spl}})$. *There is no sequential object* X_{spl} *with a single operation satisfying:*

$$VE(T_{\mathsf{spl}}) = VE(X_{\mathsf{spl}}).$$

Proof. Suppose by contradiction that there is such an object X_{spl} and consider the following fully concurrent execution for three processes:

$$E = \begin{array}{l} inv(p_1, p_1); inv(p_2, p_2); inv(p_3, p_3); resp(p_1) : \mathsf{down}; resp(p_2) : \mathsf{down}; \\ resp(p_3) : \mathsf{right}. \end{array}$$

For any prefix E' of E, one can verify that $\tau_{E'} \in \Delta_{\mathsf{spl}}(\sigma_{E'})$; for example, consider $\sigma_E = \{(p_1, p_1), (p_2, p_2), (p_3, p_3)\}$, $\tau_E = \{(p_1, \mathsf{down}), (p_2, \mathsf{down}), (p_3, \mathsf{right})\}$ and $\tau_E \in \Delta_{\mathsf{spl}}(\sigma_E)$. Then, E satisfies T_{spl}, from which follows that $E \in VE(T_{\mathsf{spl}})$.

Now, our assumption implies that $E \in VE(X_{\mathsf{spl}})$, thus E is linearizable with respect to X_{spl}. Without loss of generality suppose that there is a linearization S of E in which $inv(p_1, p_1); resp(p_1) : \mathsf{down}$ is the first linearized operation. Thus, S is a sequential execution of X_{spl}, namely, $S \in SSpec(X_{\mathsf{spl}})$. Since $SSpec(X_{\mathsf{spl}})$ is prefix-closed and $F = inv(p_1, p_1); resp(p_1) : \mathsf{down}$ is a prefix of S, we have that $F \in SSpec(X_{\mathsf{spl}})$. This is a contradiction because F is indeed an execution which is linearizable with respect to X_{spl} (F is a linearization of itself), hence $F \in VE(X_{\mathsf{spl}})$, but F does not satisfy T_{spl} (clearly $\tau_F \notin \Delta_{\mathsf{spl}}(\sigma_F)$), and thus $F \notin VE(T_{\mathsf{spl}})$, which is a contradiction. □

In a very similar way one can prove that the exchanger task defined above and the following known tasks cannot be specified as sequential objects with a single operation:

1. Adaptive renaming [4]. Processes start with distinct inputs names taken from the space $[1, \ldots, N]$ and decide distinct outputs names from the space $[1, \ldots, M]$, with $N >> M$. It is required that if $k \leq n$ processes run concurrently, the output names belong to $[1, \ldots, f(k)]$, for some function $f : 1, \ldots, n \rightarrow \{1, \ldots, N\}$, i.e., the output space is on function on the number of participating processes.
2. Set agreement [11]. It is a generalization of the well-known consensus where processes propose values and have to agree on at most k proposals.
3. Immediate snapshot [5]. It is a task which plays an important role in distributed computability [19]. A process can write a value to the shared memory using this operation, and gets back a snapshot of the shared memory, such that the snapshot occurs immediately after the write.
4. Adopt-commit [6, 14] is a concurrent object which proved to be useful to simulate round-based protocols for set-agreement and consensus. Given an input u to the object, the result is an output of the form (commit, v) or (adopt, v), where $\mathit{commit}/\mathit{adopt}$ is a decision that indicates whether the process should decide value v immediately or adopt it as its preferred value in later rounds of the protocol.
5. Conflict detection [3] is a task that has been shown to be equivalent to the adopt-commit. Roughly, if at least two different values are proposed concurrently, at least one process outputs true.

7.2. Modeling a Task as a Sequential Object with Two Operations

To circumvent the impossibility result in the previous lemma, we model any given task T through a sequential object S with two operations, set and get, that each process access in a specific way: it first invokes set with its input to the task T (receiving no output) and later invokes get in order to get an output value from T. Intuitively, decoupling the single operation of T into two (atomic) operations allows us to model concurrent behaviors that a single (atomic) operation cannot specify. In what follows, let $\mathit{SSpec}(S)$ be the set with all sequential executions of S in which each process invokes at most two operations, first set and then get, in that order.

Theorem 2. *For every task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ there is a sequential object S with two operations, $\mathsf{set}(\mathrm{id}_i, x_i)$ and $\mathsf{get}(\mathrm{id}_i) : y_i$, such that there is a bijection α between $VE(T)$ and $\mathit{SSpec}(S)$ satisfying that*

1. *each invocation or response of process id_i is mapped to an operation of process id_i,*
2. *each invocation inv (response resp) with input (output) x is mapped to a completed* set *(*get*) operation with input (output) x.*

Proof. We define S as follows. The sets of invocation and responses, *Inv* and *Res*, of S contain $\mathit{inv}(\mathsf{set}, \mathrm{id}_i, x_i)$ and $\mathit{res}(\mathsf{set}, \mathrm{id}_i, x_i) : \mathsf{void}$, respectively, for each input vertex $(\mathrm{id}_i, x_i) \in \mathcal{I}$. Similarly, for each output vertex $(\mathrm{id}_i, y_i) \in \mathcal{O}$, *Inv* and *Res* contain $\mathit{inv}(\mathsf{get}, \mathrm{id}_i)$ and $\mathit{res}(\mathsf{get}, \mathrm{id}_i) : y_i$.

For every execution $E \in VE(T)$, S has a state s_E and the initial state of S is s_ξ, where ξ denotes the empty string. We define the transition function δ of S inductively as:

1. For every execution $E \in VE(T)$ consisting of only one invocation $inv(\mathrm{id}_i, x_i)$ (i.e. $E = inv(\mathrm{id}_i, x_i)$), we define

$$\delta(s_\xi, inv(\mathsf{set}, \mathrm{id}_i, x_i)) = \{(s_E, res(\mathsf{set}, \mathrm{id}_i, x_i) : \mathsf{void})\}.$$

2. For every execution $E \in VE(T)$ with the form $E = E' \cdot e$, for some non-empty E' prefix, δ is defined as:
   (a) If $e = inv(\mathrm{id}_i, x_i)$, then

$$\delta(s_{E'}, inv(\mathsf{set}, \mathrm{id}_i, x_i)) = \{(s_{E' \cdot e}, res(\mathsf{set}, \mathrm{id}_i, x_i) : \mathsf{void})\}.$$

   (b) If $e = res(\mathrm{id}_i) : y_i$, then

$$\delta(s_{E'}, inv(\mathsf{get}, \mathrm{id}_i)) = \{(s_{E' \cdot e}, res(\mathsf{get}, \mathrm{id}_i) : y_i)\}.$$

Observe that S is a deterministic automaton whose sequential executions are precisely the executions in $VE(T)$ (one can think that S is an automaton that recognizes the language $VE(T)$). Moreover, each invocation (id_i, x_i) in an execution in $VE(T)$ induces a transition in S with an invocation to $\mathsf{set}(\mathrm{id}_i, x_i)$ and, similarly, each response (id_i, y_i) in an execution in $VE(T)$ induces a transition in S with an invocation to $\mathsf{get}(\mathrm{id}_i)$ whose response value is y_i. Thus, the desired bijection α in $VE(T) \rightarrow SSeq(S)$ is precisely obtained from the definition of S. □

An implication of Theorem 2 is that if one is analyzing an algorithm that uses a building-block (subroutine, algorithm, etc.) B that solves a task T, one can safely replace B with the sequential object S related to T described in the theorem (each invocation to the operation B is replaced with an (atomic) invocation to set and then an (atomic) invocation to get), and then analyze the algorithm considering the atomic operations of S. The advantage of this transformation is that (1) if all operations in an algorithm are atomic, we can think that each process takes a step at a time in an execution, hence obtaining a transition system with atomic events, (2) at all times we have a concrete state of S in an execution (which is not clear in a task specification) and (3) given a state of S, an output for a get operation can be easily computed using the sequential object S (something that is typically complicated for B as it might be accessed concurrently). The construction used (for simplicity) in the proof of Theorem 2 might be too coarse to be helpful for performing (1), (2) and (3) when analyzing an algorithm. Thus, we present below a construction producing an equivalent sequential automaton modeling the task in a simpler way.

Consider the sequential object in Figure 13 obtained from any given task $T = (\mathcal{I}, \mathcal{O}, \Delta)$, which is described in a classic pre/post-condition form. Intuitively, the meaning of a state (σ, τ) is the following: σ contains the processes that have invoked the task so far (this represents the *participating set* of the

current execution) while τ contains the outputs that have been produced so far. The main invariant of the specification is that $\tau \in \Delta(\sigma)$. It directly follows from the properties of the task: when a process invokes $\mathsf{set}(\mathrm{id}_i, x_i)$, we have that $\tau \in \Delta(\sigma \cup \{(\mathrm{id}_i, x_i)\})$ because $\Delta(\sigma) \subset \Delta(\sigma \cup \{(\mathrm{id}_i, x_i)\})$, and when a process invokes $\mathsf{get}(\mathrm{id}_i)$, it holds that $\tau \cup \{(\mathrm{id}_i, y_i)\} \in \Delta(\sigma)$ because $\Delta(\sigma)$ is pure of dimension $dim(\sigma)$ and thus there must exist a simplex in $\Delta(\sigma)$ (properly) containing τ and with an output for id_i. One can formally prove that this sequential object and the one in the proof Theorem 2 define the same set of sequential executions.

State: a pair (σ, τ) of input/output simplexes, initialized to $(\emptyset, \emptyset)$

Function $\mathsf{set}(\mathrm{id}_i, x_i)$
  Pre-condition:
    $\mathrm{id}_i \in \mathrm{id} \wedge \mathrm{id}_i \notin \mathrm{id}(\sigma)$
  Post-condition:
    $\sigma' \leftarrow \sigma \cup \{(\mathrm{id}_i, x_i)\}$
  Output:
    void
endFunction

Function $\mathsf{get}(\mathrm{id}_i)$
  Pre-condition:
    $\mathrm{id}_i \in \mathrm{id} \wedge \mathrm{id}_i \notin \mathrm{id}(\tau)$
  Post-condition:
    Let y_i be any output value such that $\tau \cup \{(\mathrm{id}_i, y_i)\} \in \Delta(\sigma)$
    $\tau' \leftarrow \tau \cup \{(\mathrm{id}_i, y_i)\}$
  Output:
    y_i
endFunction

Figure 13: A Generic Sequential Specification of a Task $T = (\mathcal{I}, \mathcal{O}, \Delta)$.

The formal definition of the sequential object in Figure 13 is the following.

1. For every $\sigma \in \mathcal{I}$, and for every $\tau \in \mathcal{O}$, $q_{(\sigma,\tau)}$ is a state in Q. The initial state is $q_{(\emptyset,\emptyset)}$.
2. For every input vertex $(\mathrm{id}_i, x_i) \in \mathcal{I}$, $inv(\mathsf{set}, \mathrm{id}_i, x_i) \in Inv$ and $res(\mathsf{set}, \mathrm{id}_i, x_i) : \mathsf{void} \in Res$.
3. For each output vertex $(\mathrm{id}_i, y_i) \in \mathcal{O}$, $inv(\mathsf{get}, \mathrm{id}_i) \in Inv$ and $res(\mathsf{get}, \mathrm{id}_i) : y_i \in Res$.
4. For every state $q_{(\sigma,\tau)}$,
   (a) for every (id_i, x_i) such that $\mathrm{id}_i \notin \mathrm{id}(\sigma)$ and $\sigma \cup \{(id_i, x_i)\} \in \mathcal{I}$,

$$\delta(q_{(\sigma,\tau)}, inv(\mathsf{set}, \mathrm{id}_i, x_i)) = \{(q_{(\sigma \cup \{(id_i, x_i)\}, \tau)}, res(\mathsf{set}, \mathrm{id}_i, x_i) : \mathsf{void})\},$$

   (b) for every (id_i, y_i) such that $\mathrm{id}_i \in \mathrm{id}(\sigma)$, $\mathrm{id}_i \notin \mathrm{id}(\tau)$ and $\tau \cup \{(id_i, y_i)\} \in \Delta(\sigma)$,

$$(q_{(\sigma, \tau \cup \{(id_i, y_i)\})}, res(\mathsf{get}, \mathrm{id}_i) : y_i) \in \delta(q_{(\sigma,\tau)}, inv(\mathsf{get}, \mathrm{id}_i)).$$

Finally, one can obtain simpler and equivalent specifications for specific tasks, like we did for the splitter in Section 3. Figure 4 presents such a specification where σ is represented with the set *Participants*, τ with the sets *Stop*, *Down* and *Right* and the splitter predicate in the task is literally expressed in the get operation. An ad hoc sequential specification of the exchanger is depicted in Figure 14 (a slight variation gives the exchanger used in [40]).

State: Sets *Participants*, *Matching*, both initialized to $\emptyset$

Function set(id)
  Pre-condition: id $\notin$ *Participants*
  Post-condition: *Participants*$'$ $\leftarrow$ *Participants* $\cup$ {id}
  Output: void
endFunction

Function get(id)
  Pre-condition: id $\in$ *Participants* $\wedge$ $\{\text{id}, \cdot\} \notin$ *Matching*
  Post-condition:
    Matched $\leftarrow$ $\{\text{id}^* | \{\text{id}^*, \cdot\} \in$ *Matching*$\}$
    Free $\leftarrow$ *Participants* $\setminus$ *Matched*
    if id $\in$ *Matched* **then**
      Let id^* be the value in Matched such that $\{\text{id}, \text{id}^*\} \in$ *Matched*
    else
      Let id^* be any value in *Free* $\cup \{\perp\}$
      Matching$'$ $\leftarrow$ *Matching* $\cup \{\{\text{id}, \text{id}^*\}\}$
  Output: id^*
endFunction

Figure 14: An *ad hoc* Specification of the Exchanger.

7.3. Two Additional Observations

Correctness and Completeness. In the light of the ad hoc sequential specifications in Figures 4 and 14, consider the following question: how can we know if a given sequential specification X with get and set operations corresponds to a task T, namely, it actually models T? That is to say, we consider the direction opposite to Theorem 2, from sequential objects to tasks. One way to obtain such a result is to show that there is an isomorphism between X and the sequential automaton, say S_T, obtained from the generic construction of Figure 13, instantiated with T. A second equivalent approach is to verify that X is *correct*, i.e., it satisfies the input/output relation of T, and *complete*, namely, it specifies all possible executions in $VE(T)$. Satisfying these two properties implies that X and S_T are isomorphic.

Formally, X is *correct w.r.t.* T if, for each of its executions $E \in SSpec(X)$, $\tau_E \in \Delta(\sigma_E)$, where σ_E is the simplex containing an invocation to T for each (complete) set operation of X in E, with same process and input value, and, similarly, τ_E is the simplex containing a response from T for each (complete) get operation of X in E, with same process output value.

We say that X is *complete w.r.t.* T if for each execution of $E \in VE(T)$, $S_E \in SSpec(X)$, where S_E is the sequential execution obtained from E by replacing each invocation to T in E with a complete set operation of X, with same process and input value, and each response from T in E with a complete get operation of X, with same process and output value.

On Adaptiveness. An interesting property of the splitter and Test&Set sequential objects in Figures 4 and 9 is that they do not take into account the number of processes in the system, namely, the specification is the same for any number of processes. This property is known as *adaptiveness* and can be formalized as follows in our framework.

Consider an infinite set of processes $\Pi = \{p_1, p_2, \ldots\}$. Consider a distributed problem that is specified through an infinite family of sequential objects: for every finite set $S \subset \Pi$, let X_S be a sequential object for processes in S. The family of objects is *adaptive* if for every two sets $S \subset S'$, $SSpec(X_S) = SSpec(X_{S'}, S)$, where $SSpec(X_{S'}, S)$ is the subset of $SSpec(X_{S'})$ with operations of processes in S.

The notion of adaptiveness for tasks is defined similarly. Consider a distributed problem that is specified through an infinite family of tasks: for every finite set $S \subset \Pi$, let $T_S = (\mathcal{I}_S, \mathcal{O}_S, \Delta_S)$ be a sequential object for processes in S. The family of tasks is *adaptive* if for every two sets $S \subset S'$, $\mathcal{I}_S \subset \mathcal{I}_{S'}$ and for every $\sigma \in \mathcal{I}_S$, $\Delta_S(\sigma) = \Delta_{S'}(\sigma)$.

8. Related Work

Linearizability Criteria. Neiger observed for the first time that some fundamental tasks, like *set agreement* [11] and *immediate snapshot* [5], cannot be modeled as sequential objects [35] (with a single operation). Motivated by the need of a unified framework for tasks and objects, he proposed *set-linearizability* [35]. Roughly speaking, a set sequential object is generalization of a sequential object in which transitions between states involve more than one operation (formally, a set of operations), meaning that these operations are allowed to occur concurrently, and their results can be *concurrency-dependent*. Set linearizability is the consistency condition for set-sequential objects, where one needs to find linearizability points (same as in linearizability) and several operations can be linearized at the same point (different from linearizability).

Later on, it was again observed that for some concurrent objects it is impossible to provide a sequential specification, and *concurrency-aware* linearizability was defined [17]. Set linearizability and concurrency-aware linearizability are very closely related, both based on the same principle: sets of operations can occur concurrently. Also, a non-automatic verification technique for reasoning about concurrency-aware objects is presented in [17].

Recently it was observed in [8] that some natural tasks specify concurrency dependencies that are beyond the set-linearizability and concurrency-aware formalisms, hence that paper proposed *interval linearizability*. In an interval-sequential object not only sets of operations can occur concurrently but some

of these operations might be pending and then overlap operations in the next transition; thus each operation corresponds to an interval instead of a single point. Interval linearizability is the related consistency condition in which, for each operation, one needs to find an interval in which the operation happens. It is shown in [8] that interval-linearizability is *complete* for tasks in the sense that it is possible to specify *any* task as an interval-sequential object (with a single operation).

Although interval-sequential specifications can model any task, this approach does not seem to be useful when one is searching for machine-checked proofs of concurrent algorithms. The main reason is that by replacing a task with its equivalent interval-sequential object, we obtain a transition system in which one still needs to think in concurrent behaviors, which is usually hard to deal with. In contrast, our proposed get-set transformation allows to "decouple" the inherent concurrency in tasks in a way that in the resulting transition system all events are atomic, namely, they happen one after the other.

Mechanized Verification of Distributed Algorithms. Mechanized (or machine-assisted) verification of distributed and concurrent algorithms is usually done with model checking or theorem proving or a combination of both. Enumerative model-checking is the oldest fully automatic method with tools like Spin [22] or TLC, the TLA$^+$ model checker [27]. To avoid the well-known problem of state explosion, various optimizations such as symmetry or reduction have been introduced, and recent work is on going on parameterized model checking, for instance with MCMT (Model Checking Modulo Theory) [15], Cubicle [12] or ByMC [26]. Nevertheless, automatic verification of a distributed/concurrent algorithm is still restricted to small finite instances of the algorithm or imposes significant constraints on its description, due to the limited expressiveness of the specification language.

Fully automatic theorem proving is based on a proof decision procedure. For useful logics, it is often semi-decidable at best and heavily depends on heuristics to achieve good performance. Recent work on SMT has made a substantial leap forward checking complex formulae combining first-order reasoning with decision procedures for theory such as arithmetic, equality, arrays. Nonetheless, the overall proof of a distributed algorithm is still largely manual and, when seeking confidence in this proof, an interactive proof assistant is the current approach. Several examples of verification of complex distributed algorithms exist: Chord with Alloy [43], Pastry with TLA$^+$ [31, 30], Paxos also with TLA$^+$ [29], snapshot algorithms in Event-B [2], just to cite a few.

Several wait-free implementations of tasks have been mechanically proven (e.g. [36, 42, 13]). However, to the best of our knowledge, no non-trivial algorithm built upon concurrent 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 [33] that we have considered in this paper, stacks implemented with

elimination trees [40], lock-free queues with elimination [34]. 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.

9. Final Remarks and Future Work

In this paper, we showed a technique to circumvent the known impossibility of specifying a task as a sequential object. Our technique consists in modeling the single operation of the task with two atomic operations, set and get. This transformation leads to a framework for developing transitional models of concurrent algorithms using tasks and sequential objects as building blocks. As a proof of concept, we developed the first full and modular TLA$^+$ proof of Moir & Anderson renaming algorithm [33].

A natural extension of our work is to apply the framework to other concurrent algorithms. Another direction is to extend our techniques to the case of *refined tasks* and *interval-sequential* objects, recently defined in [8]. These two formalisms are generalization of the task and sequential object formalisms with strictly more expressiveness; particularly, contrary to the task formalism, refined tasks are *multi-shot*, namely, each process may perform several invocations, possibly infinitely many.

A third direction is to study if the duality between the epistemic logic approach and the topological approach shown in [16] might be useful in verifying concurrent algorithms. Generally speaking, it is shown in [16] that a task can be represented as a *Kripke model* with an *action model*, specifying the knowledge obtained by processes when solving the task. It could be interesting to explore how this knowledge could be reflected in our set/get construction and if it could be useful in proving correctness.

Acknowledgements. Armando Castañeda was supported by PAPIIT projects IN108720 and IN108723. Aurélie Hurault and Philippe Quéinnec were supported by PARDI project ANR-16-CE25-0006.

References

[1] Dan Alistarh. The renaming problem: Recent developments and open questions. *Bulletin of the EATCS*, 117, 2015.

[2] Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. Revisiting snapshot algorithms by refinement-based techniques. *Computer Science and Information Systems*, 11(1):251–270, 2014.

[3] James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. *Theory Comput. Syst.*, 55(3):451–474, 2014.

[4] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.

[5] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC '93: Proceedings of the ACM Symposium on Theory of computing*, pages 91–100, 1993.

[6] Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

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

[8] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *Journal of the ACM*, 65(6):45:1–45:42, 2018.

[9] Armando Castañeda, Aurélie Hurault, Philippe Quéinnec, and Matthieu Roy. Tasks in modular proofs of concurrent algorithms. In *21st Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2019)*, volume 11914 of *Lecture Notes in Computer Science*, pages 69–83. Springer, October 2019.

[10] 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 *Lecture Notes in Computer Science*, pages 142–148. Springer, 2010.

[11] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.

[12] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel SMT-based model checker for parameterized systems. In *24th International Conference on Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 718–724. Springer, 2012.

[13] Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *25th International Conference on Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2013.

[14] Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In *17th ACM Symposium on Principles of Distributed Computing, PODC '98*, pages 143–152, 1998.

[15] Silvio Ghilardi and Silvio Ranise. MCMT: A model checker modulo theories. In *5th International Joint Conference on Automated Reasoning IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 22–29. Springer, 2010.

[16] Éric Goubault, Jérémy Ledent, and Sergio Rajsbaum. A simplicial complex model for dynamic epistemic logic to study distributed task computability. In *Ninth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2018*, pages 73–87, 2018.

[17] Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. Modular verification of concurrency-aware linearizability. In *29th International Symposium on Distributed Computing (DISC)*, pages 371–387, 2015.

[18] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *Journal of Parallel and Distributed Computing*, 70(1):1–12, 2010.

[19] Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2013.

[20] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[21] 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.

[22] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[23] Aurélie Hurault and Philippe Quéinnec. Proving a non-blocking algorithm for process renaming with TLA$^+$. In *13th International Conference on Tests and Proofs, TAP 2019*, volume 11823 of *Lecture Notes in Computer Science*, pages 147–166. Springer, October 2019.

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

[25] IEC. IEC-61508: Functional safety. `https://www.iec.ch/functionalsafety/`.

[26] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Formal Methods in Computer-Aided Design, FMCAD 2013*, pages 201–209. IEEE, October 2013.

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

[28] Leslie Lamport. The PlusCal algorithm language. In *Theoretical Aspects of Computing*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, August 2009.

[29] Leslie Lamport. Byzantizing paxos by refinement. In *25th International Symposium on Distributed Computing (DISC)*, volume 6950 of *Lecture Notes in Computer Science*, pages 211–224. Springer, September 2011.

[30] Tianxiang Lu. *Formal Verification of the Pastry Protocol*. PhD thesis, Université de Lorraine – Universität des Saarlandes, July 2013.

[31] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Towards verification of the Pastry protocol using TLA$^+$. In *International Conference on Formal Techniques for Distributed Systems FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 244–258. Springer, June 2011.

[32] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

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

[34] 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, SPAA 2005*, pages 253–262. ACM, July 2005.

[35] Gil Neiger. Set-linearizability. In *Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, page 396, August 1994.

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

[37] Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.

[38] William N. Scherer III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. *Communications of the ACM*, 52(5):100–111, 2009.

[39] William N. Scherer III and Michael L. Scott. Nonblocking concurrent data structures with condition synchronization. In *18th International Conference on Distributed Computing (DISC)*, pages 174–187, 2004.

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

[41] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

[42] Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. A compositional proof method for linearizability applied to a wait-free multiset. In *11th International Conference on Integrated Formal Methods, IFM 2014*, volume 8739 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2014.

[43] Pamela Zave. Using lightweight modeling to understand Chord. *SIGCOMM Computer Communication Review*, 42(2):49–57, April 2012.