A coordination-free, convergent, and safe replicated tree

Sreeja S Nair, Filipe Meirim, Mário Pereira, Carla Ferreira, Marc Shapiro

A coordination-free, convergent, and safe replicated tree

Sreeja S. Nair, Filipe Meirim, Mário Pereira, Carla Ferreira, Marc Shapiro

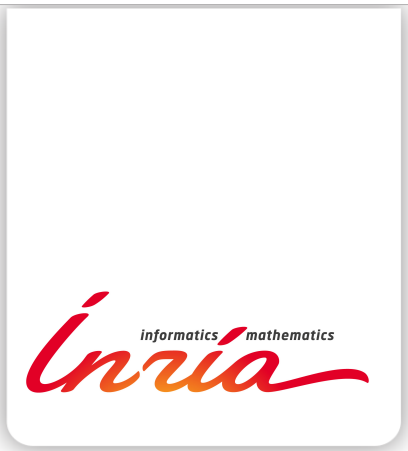
RESEARCH REPORT

N° 9395

January 2022

Project-Team DELYS

ISSN 0249-6399 ISRN INRIA/RR--9395--FR+ENG



A coordination-free, convergent, and safe replicated tree

Sreeja S. Nair*, Filipe Meirim†, Mário Pereira‡, Carla Ferreira§, Marc Shapiro¶

Project-Team DELYS

Research Report n° 9395 — January 2022 — 40 pages

Abstract: The tree is an essential data structure in many applications. In a distributed application, such as a distributed file system, the tree is replicated. To improve performance and availability, different clients should be able to update their replicas concurrently and without coordination. Such concurrent updates converge if the effects commute, but nonetheless, concurrent moves can lead to incorrect states and even data loss. Such a severe issue cannot be ignored; ultimately, only one of the conflicting moves may be allowed to take effect. However, as it is rare, a solution should be lightweight. Previous approaches would require preventative cross-replica coordination, or totally order all operations after-the-fact, requiring roll-back and compensation operations.

In this paper, we present a novel replicated tree that supports coordination-free concurrent atomic moves, and provably maintains the tree invariant. Our analysis identifies cases where concurrent moves are inherently safe, and we devise a lightweight, coordination-free, rollback-free algorithm for the remaining cases, such that a maximal safe subset of moves takes effect.

We present a detailed analysis of the concurrency issues with trees, justifying our replicated tree data structure. We provide mechanized proof that the data structure is convergent and maintains the tree invariant. Finally, we compare the response time and availability of our design against the literature.

Key-words: Distributed data structures, Conflict-free Replicated Data Type, Formal verification

* Sorbonne Université—LIP6 & Inria
† NOVA LINCS, Universidade Nova de Lisboa
‡ NOVA LINCS, Universidade Nova de Lisboa
§ NOVA LINCS, Universidade Nova de Lisboa
¶ Sorbonne Université—LIP6 & Inria

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Un arbre répliqué, convergent et sûr sans coordination

Résumé : L'arbre est une structure de données essentielle. Quand l'application est distribuée, par exemple dans un système de fichiers distribué, l'arbre est répliqué. Pour améliorer les performances et la disponibilité, les différents clients doivent pouvoir mettre à jour leurs répliques simultanément et sans coordination. Celles-ci convergent si les mises à jour commutent entre elles ; néanmoins, même dans ce cas, des opérations "move" concurrentes peuvent conduire à des états incorrects, et même à la perte de données. Au bout du compte, entre deux opérations "move" en conflit, seul l'une des deux peut être autorisée à prendre effet. Cependant, comme ce cas est rare, la solution doit être légère. Les approches précédentes nécessitaient une coordination préventive des répliques, ou des retours en arrière à posteriori.

Dans cet article, nous présentons un nouvel arbre répliqué, qui met en œuvre une opération "move" atomique sans coordination, et dont nous prouvons qu'il maintient l'invariant d'arbre. Notre analyse identifie les cas oú les "move" concurrents sont intrinsèquement sûrs, et proposons un algorithme léger, sans coordination et sans retour-arrière, pour les autres cas, de sorte qu'un sous-ensemble maximal et sûr de "move" prenne effet.

Nous présentons une analyse détaillée des problèmes de cohérence dans les arbres. Nous fournissons une preuve mécanisée que la structure des données est convergente et maintient l'invariant d'arbre. Enfin, nous comparons le temps de réponse et la disponibilité de notre concept à la littérature.

Mots-clés : Structures de données distribuées, CRDT, Vérification formelle

1 Introduction

Concurrent data structures are an important programming abstraction; designing concurrent data structures with non-trivial properties is complex. The tree data structure is used in many applications. For instance, a file system is a tree of directories and files. A move (or rename) operation transfers a subtree atomically by changing its parent. Similarly, a rich text editor maintains a DOM tree of blocks with attributes. Text editing modifies the tree structure; in particular a *drag and drop* can move a subtree from one parent to another.

A tree has a strong structural invariant: nodes are unique, there is a single root, each node has a single parent and has a path to the root, and the child-parent graph is acyclic. Much current work in concurrent data structure design focuses on lock-free or wait-free coordination using primitives such as compare-and-swap (CAS). However, in a distributed and replicated setting, even CAS is too strong. Consider a file system replicated to several locations over the globe, or through a mobile network. Network latency between continents can be between 0.1 and 0.5 seconds; the mobile network may disconnect completely. To ensure availability, a user of the file system must be able to update a replica locally, and update *without coordinating at all* with other replicas. Replicas converge eventually by exchanging their updates asynchronously.

It is a major challenge to maintain safety in this context; specifically, in this case, to maintain the tree structure. Concurrent atomic moves (also called renames in a file system) are especially problematic [3]. Consider for instance a tree composed of the root and children a and b. One replica moves a underneath b, while concurrently (without coordination) the other replica moves b under a. Naïvely replaying one replica's updates at the other produces an $a-b$ cycle disconnected from the root.

This is a widespread issue; indeed, many replicated file systems have serious anomalies, including incorrect or diverged states [17, Section 6 for some examples], violating the tree invariant [3]. However, concurrent moves are relatively rare in these systems[1] and it is important that we design a solution that has minimal overhead. Solutions in the literature include non-atomic moves [17] (resulting in duplicate copies), re-introducing coordination [15] (first one to acquire lock proceeds; others abort), or requiring roll-backs [11] (the move operation ordered first proceeds, all concurrent operations are rolled back). Najafzadeh et al. [15] shows that there can be no coordination-free solution to this problem that is not anomalous.

To support low latency, high availability, and safety, this paper introduces a new light-weight, coordination-free, safe, replicated CRDT [16] tree data structure, called *Maram*. Maram supports the usual operations to query the state, to add or to remove a node, and also supports an atomic *move* operation. The price to pay is that some move operations "lose", i.e., have no effect; achieving the same end result as previous correct approaches but at a lower cost. Query and add are unremarkable. Remove marks the corresponding node as a "tombstone," but leaves it in the data structure, as is common in replicated data structures [1]. We show that moves can be divided into two cases: two concurrent *up-move*s are always safe. We devise a deterministic arbitration rule for conflicts of *down-move*: against a concurrent up-move, the up-move wins, and the down-move loses; against a concurrent down-move, the down-move with the highest priority (as defined in Section 4.2) wins and the other loses.

We prove Maram to be safe, even in the presence of concurrent updates (including moves), despite being coordination-free and without any roll-backs. Using the Why3 proof assistant, we apply the CISE proof methodology [7], with the following steps:

1. *Sequential safety:* We show that the initial state satisfies the tree invariant, and that every individual update operation has a precondition strong enough to maintain the invariant.

[1] For example, a file system trace we analyzed contained 1198823 operations in total, 20883 create operations, 49509 remove operations and just 547 move operations (70939 structural operations altogether).

2. *Convergence:* We show that any two operations that may execute concurrently commute.
3. *Precondition stability:* We show that for any two operations u, v that may execute concurrently, u preserves the precondition of v, and vice-versa.

It follows that every state reachable from the initial state, sequentially or concurrently, satisfies the tree invariant.[2] Maram satisfies an additional desirable property, *monotonic reads* [19]. This requires that a replica that has delivered some update will not roll it back.

Losing an operation might have impacts on the causally dependent future operations. We devise a independence analysis to capture this effect. The move operations lose if it is dependent on another move operation that loses.

This paper presents the principles of Maram, proves its correctness, and compares the performance of Maram to competing solutions in a simulated geo-replicated environment. The response time of Maram is 1.35 times of the safe rollback-based design, and 1.36 times of the unsafe uncoordinated design (both due to overhead of computing the metadata required for conflict resolution), and up to 11 times faster than (safe) lock-based designs. Furthermore, Maram stabilises (updates become definitive) three orders of magnitude faster than a safe rollback-based design.

This paper proceeds as follows. Section 2 formalises our system model, explains our proof methodology, and defines the tree invariant. In Section 3 we discuss the sequential correctness of a replicated tree. Section 4 proceeds with the proof of convergence, precondition stability and independence, resulting in concurrent safety. In Section 5 we compare the performance of Maram with competing designs. Section 6 overviews the related literature. Finally, in Section 7 we discuss lessons learned and their significance.

2 Preliminaries

2.1 System Model

A distributed system is modelled as a set of processes, distributed over a (high-latency, failure-prone) communication network. The processes have disjoint memory and processing capabilities, and they communicate through message passing. A process does not fail. Every message is eventually delivered to its destination. Message delivery is consistent with happens-before (causal consistency).

State and invariant:

The data structure (in this case, a tree) is *replicated* at a number of processes, called its *replicas*. The information managed by a replica on behalf of the data structure is called its *local state*. The union of local states is called the *global state*.[3]

A data structure is associated with an *invariant*, a predicate that must always be satisfied in every local state of a replica. Although evaluated locally, an invariant describes a global property, in the sense that it must be true at all replicas.

Operations:

An unspecified client application submits an operation at some replica of its choice, which we call the *origin* replica of that operation. For availability, the origin replica should carry out the

[2] We furthermore claim (without proof) that Maram is live, in the sense that, if every message sent is eventually delivered to some replica r_1, then, given some update originating at a replica r_2, its postcondition eventually takes effect at replica r_1.

[3] Note that this global view cannot be observed by any single replica and is merely an explanatory device.

operation without waiting to coordinate with other replicas.

An update operation has a *postcondition* that specifies the state after the operation executes, and a *precondition* that indicates the domain of the operation. As discussed in more detail later, when the operation executes with no concurrency, its precondition guarantees that the operation terminates with the postcondition satisfied.

Updates:

When a client submits an operation, the origin replica generates an *effector* (a side-effecting lambda), atomically applies the effector to the origin state, and sends the effector to all the other replicas. Every replica eventually receives and delivers the effector, atomically applying it to its own local state.[4] The effector eventually executes at every replica.

We assume that effectors are delivered in causal order. This means that, if some replica that observed an effector u later generates an effector v, then any replica that observes v has previously observed u.[5]

In what follows, we ignore queries, and identify an update operation with executing its effector at all replicas.

2.2 Properties and associated Proof Rules

Consider some data structure (in this case a tree) characterized by a safety *invariant* (in this case, the tree invariant). We say that a state is *local-safe* if it satisfies the data structure's invariant. An update is *op-safe* if, starting from a local-safe state, it leaves it a local-safe state. The distributed data structure is *safe* if every update is op-safe. According to the CISE logic [7], a distributed data structure is safe if the following properties hold:

1. *Sequential safety*: Consider an environment restricted to sequential execution (operations execute one after another; there is no concurrency). If the initial state is local-safe at every replica, and each update is op-safe, it follows that the data structure is safe under sequential execution. Classically, sequential op-safety implies that each operation's precondition satisfies the weakest-precondition of the invariant with respect to the operation [4].
2. *Convergence:* Strong Eventual Consistency (SEC) [16] states that two replicas that have delivered the same set of operations must be in the same state, i.e., the system converges. If operations commute (as defined later), then SEC is guaranteed [16].
3. *Precondition stability:* In addition to sequential safety, updates must remain op-safe in the presence of concurrent (uncoordinated) updates. To ensure this, we apply the CISE precondition stability rule [7]: consider two updates u and v; if the execution of u does not make the precondition of v false, nor vice-versa (*precondition stability*), then executing u and v concurrently is op-safe. This must be true for all concurrent pairs of operations.

CISE logic helps us identify the conditions under which concurrent operations conflict. When conflicting, CISE requires the operations to acquire tokens, that bring in a global synchronization point. Hence all updates in CISE are assumed to be definitive.

In order to augment the CISE analysis for handling tentative updates, we add a condition for *independence* to check whether skipping a move affects a move that already observed the effect of the skipped one. The independency analysis is inspired from Houshmand and Lesani [8], even though they also, like CISE, do not consider tentative updates.

[4]Note that, at this point, the system is committed to this operation, and the operation's precondition must be true at the remote replica.

[5]In Section 7 we consider relaxing this requirement to eventual consistency, which states only that all updates are eventually delivered at all replicas.

Independence analysis: Consider two updates u and v that are safe, u executed before v. If moving v before u still maintains the safety of v, v is said to be independent of u. Otherwise, if v is unsafe before executing u, v is dependent on the effect of u.

2.2.1 Sequential safety

Let us refine the proof obligations of the first step, sequential safety, i.e., local-safety under sequential execution.

The set of reachable states comprises the initial state, and all states transitively reachable as a result of executing updates sequentially. The set of reachable states is a subset of the set of all possible states. Formally, we note the set of states Σ, a state σ, the initial state σ_{init}, an update u, its precondition Pre_u, and the set of updates U. When execution is sequential:

$$\sigma_{init} \in \Sigma \tag{1}$$

and

$$\forall u \in U, \sigma \in \Sigma \centerdot \sigma \models Pre_u \implies u(\sigma) \in \Sigma \tag{2}$$

Σ is the smallest set satisfying (1) and (2) through a sequence of legal updates from the initial state.

The data structure must satisfy its invariant in every sequentially reachable state: this property is called *sequential safety*. Formally, if Inv denotes the invariant, then

$$\forall \sigma \in \Sigma \centerdot \sigma \models Inv \tag{3}$$

If the initial state is safe and all sequential updates preserve the invariant, by induction, the data structure is sequentially safe. Formally, if the initial state, σ_{init}, satisfies the invariant, Inv,

$$\sigma_{init} \models Inv \tag{4}$$

and each update u executing on a state σ preserves the invariant,

$$\forall u \in U, \sigma, \sigma' \in \Sigma \centerdot \sigma \models (Inv \wedge Pre_u) \wedge u(\sigma) = \sigma' \implies \sigma' \models Inv \tag{5}$$

then the invariant holds true for all reachable states. Pre_u is the weakest precondition required to maintain the safety of update u. Weakest precondition for an update can be calculated by predicate transformer semantics as described by Dijkstra [4].

2.2.2 Concurrency

Let us now turn to concurrent execution, and consider the proof obligations for convergence and safety.

Convergence If a replica initiates an update u, while concurrently another replica initiates v, the first replica executes their effectors in the order $u; v$ and the second one in the order $v; u$. Without precaution, it is likely that their states diverge.

To prevent this, the Strong Eventual Consistency (SEC) property [16] requires that any two replicas that delivered the same updates are in equivalent states. To satisfy SEC, effector functions are designed to commute, i.e., both orders above leave the data in the same state. We define commutativity as follows:

$$\forall u_1, u_2 \in U, \sigma, \sigma_1, \sigma_2 \in \Sigma \centerdot u_1(\sigma) = \sigma_1 \wedge u_2(\sigma) = \sigma_2 \implies u_2(\sigma_1) = u_1(\sigma_2) \tag{6}$$

Precondition stability The main proof obligation, for concurrent execution, is that the precondition of any effector is stable against (i.e., not negated by) an effector that may execute concurrently [7]. This CISE rule is a variant of rely-guarantee reasoning, adapted to a replicated system where effectors execute atomically and definitively. The precondition stability condition can be formally specified as follows:

$$\forall u_1, u_2 \in U, \sigma, \sigma' \in \Sigma \centerdot \sigma \models (Inv \wedge Pre_{u_1} \wedge Pre_{u_2}) \wedge u_1(\sigma) = \sigma' \implies \sigma' \models Pre_{u_2} \tag{7}$$

Gotsman et al. [7] uses *Tokens* to formalize concurrency control. Two operations that share the same token do not execute concurrently. Since we are designing a coordination-free data structure, we consider the set of tokens to be an empty set, and hence absent from the formalisation.

Independence In order to ensure that the safety of an operation, is not impacted by skipping any previous operations, we augment the precondition stability analysis with an independence analysis as presented by Houshmand and Lesani [8]. An operation u_2 is said to be independent of operation u_1 if the precondition of u_2, Pre_{u_2}, is enabled even without executing u_1. The condition for independency can be formally specified as follows:

$$\begin{aligned}\forall u_1, u_2 \in U, \sigma, \sigma', \sigma'', \sigma''' \in \Sigma \centerdot \sigma \models (Inv \wedge Pre_{u_1}) \wedge \sigma' \models (Inv \wedge Pre_{u_2}) \\ \wedge \sigma'' \models Inv \wedge u_1(\sigma) = \sigma' \wedge u_2(\sigma') = \sigma'' \wedge u_2(\sigma) = \sigma''' \implies \sigma \models Pre_{u_2} \wedge \sigma''' \models Inv\end{aligned} \tag{8}$$

In short, u_2 is independent of u_1 if, irrespective of whether u_1 executed before u_2, the execution of u_2 is safe. This condition is required for safety only if the effect of u_1 is tentative, i.e., if u_1 has conflict resolution policies while applying the update on the state.

2.2.3 Mechanized verification

In order to mechanically discharge the proof obligations listed above, we the use Why3 system [6], augmented with the CISE3 plug-in [13]. Why3 is a framework used for the deductive verification of programs, *i.e.*, "the process of turning the correctness of a program into a mathematical statement and then proving it" [5]. The CISE3 plug-in automates the CISE proof rules described above, and generates the required sequential-safety, commutativity and stability checks. Why3 then computes a set of proof obligations, that are discharged via external theorem provers.

3 Sequential specification of a tree

3.1 State

The state of a tree data structure consists of a set of nodes, *Nodes*, and a relation from a child node to its parent, indicated by $\rightarrow$. The ancestor relation, $\rightarrow^*$ is defined as

$$\forall a, n \in Nodes \centerdot n \rightarrow^* a \implies n \rightarrow a \ \vee \ \exists p \in Nodes \centerdot n \rightarrow p \wedge p \rightarrow^* a \tag{9}$$

At initialization, the set of nodes consists of a single *root* node. The parent of the root is root itself. The initial state of the tree is thus $Nodes = \{root\}$ where $root \rightarrow root$. A crucial aspect of the abstract representation of the tree is how to express the relation between nodes. Three choices are possible, either maintain a child-to-parent relation, a parent-to-child relation, or both. In particular, when implementing a tree, traversal efficiency depends on keeping both up and down pointers [18]. Considering that child-to-parent and parent-to-child relations describe a dual view of a tree (i.e., node p is the parent of node n iff node n is a descendent of node p) we selected the

one that leads to a simpler specification. An advantage of using a child-to-parent relation is that it can be maintained as a function, as the tree properties ensure that each node has a unique parent. The alternative parent-to-child relation would need a more complex representation, e.g. a function that maps each node to its set of direct descendants, which would impact the simplicity of the specification and the proof effort.

3.2 Invariant

The invariant of the tree data structure is as follows:

$$\begin{aligned} root \in Nodes \land root \to root \ \land\ \forall \in Nodes \centerdot n \neq root \implies root \not\to n && (Root)\\ \land\ \forall n \in Nodes \centerdot \exists p \in Nodes \centerdot n \to p && (Parent)\\ \land\ \forall n, p, p' \in Nodes \centerdot n \to p \land n \to p' \implies p = p' && (Unique)\\ \land\ \forall n \in Nodes \centerdot n \to^* root && (Reachable)\\ Inv \triangleq Root \land Parent \land Unique \land Reachable && (10) \end{aligned}$$

Clause *Root* states that the root node is present in *Nodes*, and is the only node to be its own parent. Clause *Parent* asserts that every node in the tree has a parent in the tree. Clause *Unique* requires the parent of a node to be unique. Clause *Reachable* imposes that the root is an ancestor of all nodes. We call this conjunction, Equation (10), the *tree invariant*.

A further invariant which forbids cycles can be derived:

$$\forall n \in Nodes \centerdot n \neq root \implies n \not\to^* n \qquad (Acyclic)$$

Since the parent relation inductively defines the ancestor relation, by *Unique* there is a unique path to a given ancestor of a node. By *Reachable*, the root node is an ancestor of every node in the tree. In this scenario, a cycle would require a node to have multiple parents, which is prevented by *Unique*.

3.3 Operations

Add An add operation has two arguments: the node to be added, n, and its prospective parent, p. The add effector adds node n to *Nodes* and the mapping $n \to p$ to the parent relation. The postcondition of the add effector indicates this:[6]

$$Post_{add(n,p)} \triangleq n \in Nodes \land n \to p \qquad (11)$$

To ensure the tree invariant, we derive the precondition that n is a new node and p is already in the tree, i.e.,

$$Pre_{add(n,p)} \triangleq n \notin Nodes \land p \in Nodes \qquad (12)$$

Let us see how this precondition is derived. If the add operation is updating a safe state, i.e., the starting state respects the invariant, and if the precondition is satisfied, then the update should maintain the invariant. Hereafter, we highlight the precondition clauses needed to ensure each part of the invariant.[7]

[6] For readability, we simplify the postcondition to express only the changes caused by the operation. The part of the state not mentioned remains unaffected.

[7] Denoted in inference style, as in [9]. The condition above the line represents the pre-state, an update event is noted $[\![.]\!]$, and the condition below the line indicates the post-state.

Precondition	Invariant clause			
	Root	*Parent*	*Unique*	*Reachable*
$add(n, p)$	$n \notin Nodes$	$p \in Nodes$	$n \notin Nodes$	$p \in Nodes$
$rem(n)$	$n \neq root$	$\forall n' \in Nodes \,.\, n' \not\mapsto n$	`true`	$\forall n' \in Nodes \,.\, n' \not\mapsto n$
$move(n, p')$	$n \neq root$	$p' \in Nodes$	`true`	$p' \in Nodes \wedge p' \neq n \wedge p' \not\mapsto^* n$

Table 1: Precondition required by each operation to uphold specific clauses of the invariant

$$\frac{Inv \wedge n \notin Nodes \qquad [\![add(n,p)]\!]}{Post_{add(n,p)} \wedge Root \wedge Unique} \qquad \frac{Inv \wedge p \in Nodes \qquad [\![add(n,p)]\!]}{Post_{add(n,p)} \wedge Parent \wedge Reachable}$$

Table 1 lists the preconditions required by operations to preserve each invariant clause. With the derived preconditions, the add operation can be specified as follows:

(Add-Operation)
$$\frac{Inv \wedge n \notin Nodes \wedge p \in Nodes \qquad [\![add(n,p)]\!]}{Inv \wedge n \in Nodes \wedge n \to p}$$

If the add operation is issued on a state that is safe and contains p but not n, then n is added to the tree with parent p.

Remove operation Remove receives as argument a node n to be deleted. Its effector removes node n from the set of nodes. The postcondition of remove indicates this effect:

$$Post_{rem(n)} \triangleq n \notin Nodes \tag{13}$$

Similarly to add, we list the predicates needed to preserve each clause of the invariant in Table 1. In the case of the remove operation, we need to ensure that n is not the root, and n is a leaf node, i.e., there are no child nodes for n. The remove can be specified as follows:

(Remove-Operation)
$$\frac{Inv \wedge n \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\mapsto n \qquad [\![rem(n)]\!]}{Inv \wedge n \notin Nodes}$$

If a remove operation is issued on a safe state where n is not *root* and has no children, then n is removed from the tree.

Move operation The move operation takes two arguments: the node to be moved n, and the new parent p'. Its effector changes the parent of node n to p' as follows:

$$Post_{move(n,p')} \triangleq n \to p' \tag{14}$$

To preserve the expected behaviour we require that the node to be moved is already present in the tree. We derive the safety clauses as shown in Table 1. Formally, the move operation can be specified as follows:

(Move-Operation)
$$\frac{Inv \wedge n \in Nodes \wedge n \neq root \wedge p' \in Nodes \wedge p' \neq n \wedge p' \not\mapsto^* n \qquad [\![move(n,p')]\!]}{Inv \wedge n \to p'}$$

For the move operation to be safe, n is not the root, p' is in the tree, n and p' are different, and p' is not a descendant of n. These last two conditions are needed to prevent move from creating a cycle of unreachable nodes, as we show with the following counterexample.

Consider a tree composed of nodes a and b. Root node R is the parent of node a, i.e., $a \rightarrow R$ and node a is the parent of node b, $b \rightarrow a$, and hence R is the ancestor of b, $b \rightarrow^* R$. Moving a under b will make both a and b unreachable from the root, and also form a cycle. This violates the invariant by invalidating the tree structure. To avoid this scenario, a precondition is needed that prevents moving a node underneath itself. When moving node n from its current parent to the new parent p', p' should not be a descendant of n, $p' \not\rightarrow^* n$.

3.4 Mechanized verification of the sequential specification

Following the formalization of the tree data structure above, we use Why3 to mechanically prove its sequential safety. The mechanical proof requires some extra definitions and axioms.

We need a predicate for reachability. For this, we first define a path, a sequence of nodes related by the parent relation. We use $s[n]$ to indicate the nth element in the sequence s. We denote the set of possible sequences of nodes by S. The path predicate determines the validity conditions for a path s between nodes x and y in state σ. If $x = y$, the path has length zero. Otherwise, the length of the path is greater than zero, where the first path element must be x, all contiguous path elements are related by the parent relation, and node y is the parent of the last path element. We say y is reachable from x if there exists a path from x to y. Formally,

$$\begin{aligned} path(\sigma, x, y, s) \triangleq\ & length(s) = 0 \wedge x = y \\ & \vee (length(s) > 0 \wedge s[0] = x \wedge s[length(s) - 1] \rightarrow y \wedge \\ & \quad \forall\, 0 \leq i < length(s) - 1 \,.\, s[i] \rightarrow s[i + 1]) \end{aligned} \tag{15}$$

$$reachability(\sigma, x, y) \triangleq \exists s \in S \,.\, path(\sigma, x, y, s) \tag{16}$$

To formalize the properties of the *path* predicate, we define a set of axioms as follows:

$$path\_to\_parent \triangleq \forall \sigma \in \Sigma \,.\, \forall x, y \in Nodes \,.\, x \rightarrow y \implies \exists s \in S \,.\, path(\sigma, x, y, s) \wedge s = [x] \tag{17}$$

$$\begin{aligned} path\_composition \triangleq\ & \forall \sigma \in \Sigma \,.\, \forall x, y, z \in Nodes \,.\, \exists s_1 \in S \,.\, path(\sigma, x, y, s_1) \\ & \wedge\, y \rightarrow z \implies \exists s_2 \in S \,.\, path(\sigma, x, z, s_2) \wedge s_2 = s_1 + [y] \end{aligned} \tag{18}$$

$$\begin{aligned} path\_transitivity \triangleq\ & \forall \sigma \in \Sigma \,.\, \forall x, y, z \in Nodes, s_1, s_2 \in S \,.\, path(\sigma, x, y, s_1) \\ & \wedge\, path(\sigma, y, z, s_2) \implies \exists s_3 \in S \,.\, path(\sigma, x, z, s_3) \wedge s_3 = s_1 + s_2 \end{aligned} \tag{19}$$

$$\begin{aligned} path\_uniqueness \triangleq\ & \forall \sigma \in \Sigma \,.\, \forall x, y \in Nodes, s_1, s_2 \in S \,.\, path(\sigma, x, y, s_1) \\ & \wedge\, path(\sigma, x, y, s_2) \implies s_1 = s_2 \end{aligned} \tag{20}$$

$$path\_exclusion \triangleq \forall \sigma \in \Sigma \,.\, \forall x, y, z \in Nodes, s \in S \,.\, x \not\rightarrow^* y \wedge path(\sigma, z, y, s) \implies x \notin s \tag{21}$$

$$\begin{aligned} path\_separation \triangleq\ & \forall \sigma \in \Sigma \,.\, \forall x, y, z \in Nodes, s_1, s_2 \in S \,.\, path(\sigma, x, y, s_1) \\ & \wedge\, path(\sigma, y, z, s_2) \wedge x \neq y \wedge x \neq z \wedge y \neq z \implies s_1 \cap s_2 = \emptyset \end{aligned} \tag{22}$$

Axiom *path_to_parent* defines the singleton path of a node to its parent. The recursive composition of paths is axiomatized in *path_composition*. The transitivity property is defined in *path_transitivity*. Axiom *path_uniqueness* asserts there is a single path between two nodes. The *path_exclusion* expresses the conditions for excluding nodes from a path. Lastly, *path_separation* defines a convergence criterion essential for Why3's SMT solvers, asserting that the direction of the path is converging towards the root.

We also require extra axioms to express the properties of the unaffected nodes in the case of

add and move operations. They are as follows:

$$\sigma_{add} = add(n,p)(\sigma)$$
$$\sigma_{move} = move(n,p)(\sigma)$$
$$remaining\_nodes\_add \triangleq \forall \sigma \in \Sigma \,.\, \forall n' \in Nodes, s_1, s_2 \in seq(Nodes) \,.\, n' \neq n \wedge path(\sigma, n', root, s_1) \wedge path(\sigma_{add}, n', root, s_2) \implies s_1 = s_2 \quad (23)$$
$$descendants\_move \triangleq \forall \sigma \in \Sigma \,.\, \forall n' \in Nodes, s_1, s_2 \,.\, path(\sigma, n', c, s_1) \wedge path(\sigma_{move}, n', c, s_2) \implies s_1 = s_2 \quad (24)$$
$$remaining\_nodes\_move \triangleq \sigma \in \Sigma \,.\, \forall n' \in Nodes, s_1, s_2 \,.\, n' \not\rightarrow^* n \wedge path(\sigma, n', root, s_1) \wedge path(\sigma_{move}, n', root, s_2) \implies s_1 = s_2 \quad (25)$$

The state σ_{add} is obtained by applying $add(n,p)$ operation to σ. The axiom *remaining_nodes_add* asserts that the paths already present in the tree remain in the tree after executing the add operation. Given that the move operation updates σ to σ_{move}, axiom *descendants_move* asserts that the descendants of the node being moved continue to be its descendants, and *remaining_nodes_move* asserts that other paths are not affected. These axioms are defined to ensure that the paths to the root, from nodes unaffected by move or add operations, remain unchanged. The specification proven using Why3 is available in Meirim et al. [14].

4 Concurrent tree specification

In this section, we discuss the convergence and concurrent safety of the tree. In a sequential execution environment, as seen in Section 3, if the initial state and each individual update are safe, then all reachable states are safe. This is not true when executing concurrently on multiple replicas. In this case, there are three extra proof obligations (Subsection 2.2.2, Subsection 2.2.2, Subsection 2.2.2):

- Ensuring that different replicas converge, despite effectors being executed concurrently in different orders.
- Ensuring that safety of an update is not violated by a concurrent update.
- Ensuring that a tentative update does not effect the safety of the dependent update.

For ease of exposition, first we discuss concurrent safety; convergence is deferred to Subsection 4.3, since the conflicts occurring in the latter can be addressed using the policies discussed in the former, and independence is discussed in Subsection 4.4.

4.1 Precondition stability

We use the precondition stability rule of CISE logic (Subsection 2.2.2) to analyze the concurrent safety of our tree data structure. For each operation, we analyze whether it violates the precondition of any other concurrent operation. Formally, operation op_1 is stable under operation op_2 if,

$$\frac{Inv \wedge Pre_{op_1} \wedge Pre_{op_2} \qquad [\![op_2]\!]}{Inv \wedge Post_{op_2} \wedge Pre_{op_1}} \quad (26)$$

We check the sequential specification for stability. If this fails, then it will be necessary to modify the specification, so that it does satisfy stability.

4.1.1 Stability of add operation

Concurrent adds: First we check the stability of the precondition of add against itself. Let us consider two operations $add(n_1, p_1)$ and $add(n_2, p_2)$. Using Equation (26), we get

$$\begin{aligned} Pre_{add(n_1,p_1)} &\triangleq n_1 \notin Nodes \wedge p_1 \in Nodes \\ Pre_{add(n_2,p_2)} &\triangleq n_2 \notin Nodes \wedge p_2 \in Nodes \\ Post_{add(n_2,p_2)} &\triangleq n_2 \in Nodes \wedge n_2 \rightarrow p_2 \end{aligned}$$

$$\frac{Inv \wedge Pre_{add(n_1,p_1)} \wedge Pre_{add(n_2,p_2)} \wedge \boxed{n_1 \neq n_2} \qquad [\![add(n_2,p_2)]\!]}{Inv \wedge Post_{add(n_2,p_2)} \wedge Pre_{add(n_1,p_1)}} \tag{27}$$

The highlighted clause $n_1 \neq n_2$ is required for the stability condition. Indeed, the sequential specification does not disallow adding the same node at different replicas, and the clause $n \notin Nodes$ is unstable therein. Thus the analysis highlights a subtlety.

Concurrent remove: Let us check the stability of the precondition of $add(n_1, p_1)$ against a concurrent $remove(n_2)$. Using (26), we get:

$$\begin{aligned} Pre_{add(n_1,p_1)} &\triangleq n_1 \notin Nodes \wedge p_1 \in Nodes \\ Pre_{remove(n_2)} &\triangleq n_2 \neq root \wedge \forall\, n' \in Nodes \centerdot n' \not\rightarrow n_2 \\ Post_{remove(n_2)} &\triangleq n_2 \notin Nodes \end{aligned}$$

$$\frac{Inv \wedge Pre_{add(n_1,p_1)} \wedge Pre_{remove(n_2)} \wedge \boxed{n_2 \neq p_1} \qquad [\![remove(n_2)]\!]}{Inv \wedge Post_{remove(n_2)} \wedge Pre_{add(n_1,p_1)}} \tag{28}$$

In the sequential specification, clause $p_1 \in Nodes$ in the precondition of add is unstable against a remove of its parent; performing those operations concurrently would be unsafe.

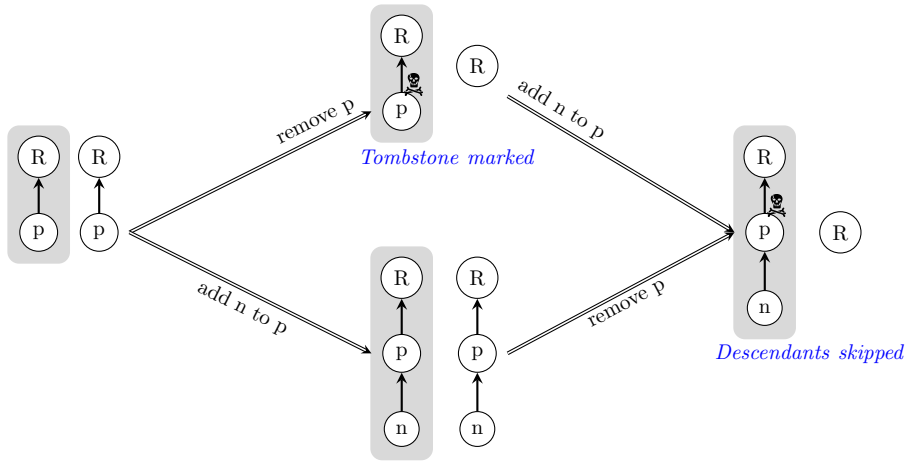
To fix this, we see two possible approaches. The classical way is to strengthen the precondition with coordination, for instance locking to avoid concurrency. We reject this, as it conflicts with our objective of availability under partition. Our alternative is to weaken the specification thanks to coordination-free conflict resolution. We apply a common approach, to mark a node as deleted, as a so-called *tombstone*, without actually removing it from the data structure.[8]

We now distinguish a *concrete* state and its *abstract* view. We modify the specification to include a set of tombstones, *TS* (initially empty), in the concrete state. The abstract state is the resolved state as seen by some application using Maram. An *abstraction function* maps the concrete state to the abstract state.

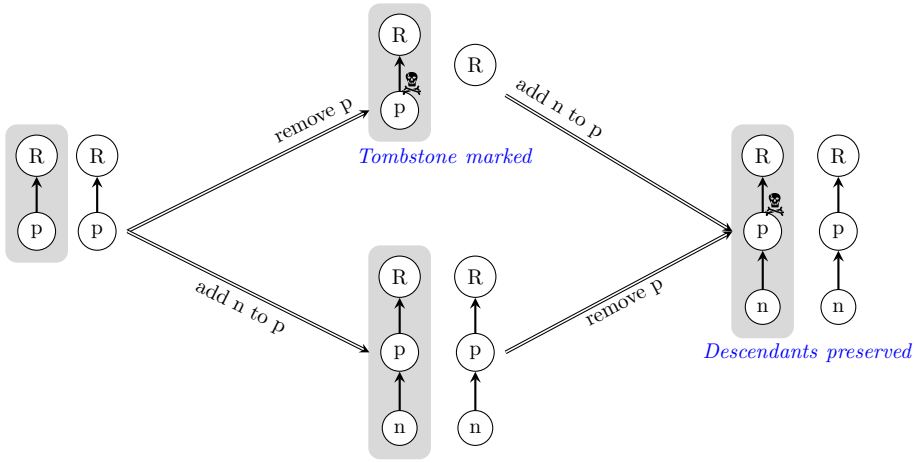
The concrete and abstract states of a tree are the same if either there are no nodes in the set of tombstones or for each node in the set of tombstones, all its descendants are also present in the set of tombstones. In other cases, the abstraction function need to provide guidance on the presence of the descendants of a node that appears in the set of tombstones.

We present two *abstraction functions* - *skipping_abstraction* and *keeping_abstraction*. The *skipping_abstraction* skips the descendants of the node that is marked as a tombstone. The *keeping_abstraction*, on the other hand, preserves the tombstoned node if it observes the node has a descendant not in the set of tombstones. Both the abstraction functions satisfy the required safety properties since they only change the view of the tree for an application. Therefore the choice is application-specific.

[8] Ideally, one will remove the tombstone at some safe time in the future; this is non-trivial [2] and out of the scope of this paper.



(a) Skipping abstraction



(b) Keeping abstraction

Figure 1: Resolving conflict of concurrent remove and add

Formally, if $Nodes_{con}$ and $Nodes_{abs}$ denote the set of nodes in the concrete and abstract state respectively,

$$skipping\_abstraction \triangleq \forall n \in Nodes_{con} \,.\, n \notin TS \wedge \nexists n' \in Nodes_{con} \,.\, n' \in TS \wedge n \rightarrow^* n' \iff n \in Nodes_{abs} \tag{29}$$

$$keeping\_abstraction \triangleq \forall n \in Nodes_{con} \,.\, n \notin TS \vee \exists n' \in Nodes_{con} \,.\, n' \notin TS \wedge n' \rightarrow^* n \iff n \in Nodes_{abs} \tag{30}$$

To illustrate the difference, consider the tree consisting of the root and a single child, as shown in Figure 1. One replica performs a remove of node p, while concurrently another replica adds n under p. In the first replica, node p is marked as a tombstone in the concrete state (the shaded box). Thus, the abstract state shows node p removed. When the replicas exchange their updates, they converge to the concrete state (the state in the shaded box). Figure 1a and Figure 1b show the result of a *skipping_abstraction* and *keeping_abstraction* respectively. In both the cases, node

p is marked as a tombstone. In the case of the *skipping_abstraction*, node p and the descendants are "skipped". Meanwhile for *keeping_abstraction*, since its descendant n is not a tombstone, p is "revived" in the abstract view.

With tombstones, let us update the postcondition for remove:

$$Post_{remove(n)} \triangleq n \in TS \tag{31}$$

Let us now derive the predicates needed to preserve each clause of the invariant in this refined case.

$$\frac{Inv \wedge \boxed{n \neq root} \qquad [\![remove(n)]\!]}{Post_{remove(n)} \wedge Root} \qquad \frac{Inv \wedge \boxed{\texttt{true}} \qquad [\![remove(n)]\!]}{Post_{remove(n)} \wedge Parent}$$

$$\frac{Inv \wedge \boxed{\texttt{true}} \qquad [\![remove(n)]\!]}{Post_{remove(n)} \wedge Unique} \qquad \frac{Inv \wedge \boxed{\texttt{true}} \qquad [\![remove(n)]\!]}{Post_{remove(n)} \wedge Reachable}$$

To maintain sequential safety in the modified remove specification, the precondition forbids only removing the root node. As the remove operation does not alter the tree structure, reachability is not impacted. The refined specification of the remove operation is as follows:

$$\frac{\begin{array}{c}\text{(Remove-Operation)}\\ Inv \wedge n \neq root \qquad [\![remove(n)]\!]\end{array}}{Inv \wedge n \in TS}$$

The application could strengthen this precondition with an added clause to delete only the leaf nodes visible in the abstract view. This helps prevent accident loss of a sub-tree. Since this is not necessary for safety, we are not considering that condition.

Concurrent move: Next we check the stability of the precondition of add under a concurrent move operation. Let us consider two operations $add(n_1, p_1)$ and $move(n_2, p_2')$. Using (26), we get

$$\begin{aligned} Pre_{add(n_1,p_1)} &\triangleq n_1 \notin Nodes \wedge p_1 \in Nodes \\ Pre_{move(n_2,p_2')} &\triangleq n_2 \in Nodes \wedge n_2 \neq root \wedge p_2' \in Nodes \wedge p_2' \neq n_2 \wedge p_2' \not\rightarrow^* n_2 \\ Post_{move(n_2,p_2')} &\triangleq n_2 \rightarrow p_2' \end{aligned}$$

$$\frac{Inv \wedge Pre_{add(n_1,p_1)} \wedge Pre_{move(n_2,p_2')} \wedge \boxed{\texttt{true}} \qquad [\![move(n_2, p_2')]\!]}{Inv \wedge Post_{move(n_2,p_2')} \wedge Pre_{add(n_1,p_1)}} \tag{32}$$

We see that the precondition of add is stable against a concurrent move operation.

4.1.2 Stability of remove operation

Concurrent add: Consider the sequential specification of two operations $remove(n_1)$ and $add(n_2, p_2)$. Using (26), we get

$$\begin{aligned} Pre_{remove(n_1)} &\triangleq n_1 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n_1 \\ Pre_{add(n_2,p_2)} &\triangleq n_2 \notin Nodes \wedge p_2 \in Nodes \\ Post_{add(n_2,p_2)} &\triangleq n_2 \in Nodes \wedge n_2 \rightarrow p_2 \end{aligned}$$

$$\frac{Inv \wedge Pre_{remove(n_1)} \wedge Pre_{add(n_2,p_2)} \wedge \boxed{n_1 \neq p_2} \qquad [\![add(n_2, p_2)]\!]}{Inv \wedge Post_{add(n_2,p_2)} \wedge Pre_{remove(n_1)}} \tag{33}$$

We see that the clause that node n_1 has to be a leaf node is not satisfied if $n_1 = p_2$ since add operation introduces a child node under p_2. However, the refined specification of tombstones as described above does not require the node n_1 to be a leaf node. So that solution fixes this conflict as well.

Concurrent remove: Consider the sequential specification of two remove operations $remove(n_1)$ and $remove(n_2)$. Using (26), we get

$$\begin{aligned} Pre_{remove(n_1)} &\triangleq n_1 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n_1 \\ Pre_{remove(n_2)} &\triangleq n_2 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n_2 \\ Post_{remove(n_2)} &\triangleq n_2 \notin Nodes \end{aligned}$$

$$\frac{Inv \wedge Pre_{remove(n_1)} \wedge Pre_{remove(n_2)} \wedge \texttt{true} \qquad [\![remove(n_2)]\!]}{Inv \wedge Post_{remove(n_2)} \wedge Pre_{remove(n_1)}} \tag{34}$$

We see that the remove operation is stable under a concurrent remove. Furthermore, the refined specification is also stable since it adds n_1 and n_2 to TS.

Concurrent move: Consider the sequential specification of two operations $remove(n_1)$ and $move(n_2, p_2')$. Using (26), we get

$$\begin{aligned} Pre_{remove(n_1)} &\triangleq n_1 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n_1 \\ Pre_{move(n_2,p_2')} &\triangleq n_2 \in Nodes \wedge n_2 \neq root \wedge p_2' \in Nodes \wedge p_2' \neq n_2 \wedge p_2' \not\rightarrow^* n_2 \\ Post_{move(n_2,p_2')} &\triangleq n_2 \rightarrow p_2' \end{aligned}$$

$$\frac{Inv \wedge Pre_{remove(n_1)} \wedge Pre_{move(n_2,p_2')} \wedge n_1 \neq p_2' \qquad [\![move(n_2, p_2')]\!]}{Inv \wedge Post_{move(n_2,p_2')} \wedge Pre_{remove(n_1)}} \tag{35}$$

We see that the clause for the remove operation that n_1 should be a leaf node is violated if a node is moved under it. Again, we can observe that the refined specification of remove eliminates this issue due to the absence of the violation-causing clause.

4.1.3 Stability of move operation

Concurrent add: Consider the sequential specification of two operations $move(n_1, p_1')$ and $add(n_2, p_2)$. Using (26), we get

$$\begin{aligned} Pre_{move(n_1,p_1')} &\triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p_1' \in Nodes \wedge p_1' \neq n_1 \wedge p_1' \not\rightarrow^* n_1 \\ Pre_{add(n_2,p_2)} &\triangleq n_2 \notin Nodes \wedge p_2 \in Nodes \\ Post_{add(n_2,p_2)} &\triangleq n_2 \in Nodes \wedge n_2 \rightarrow p_2 \end{aligned}$$

$$\frac{Inv \wedge Pre_{move(n_1,p_1')} \wedge Pre_{add(n_2,p_2)} \wedge \texttt{true} \qquad [\![add(n_2, p_2)]\!]}{Inv \wedge Post_{add(n_2,p_2)} \wedge Pre_{move(n_1,p_1')}} \tag{36}$$

The precondition of move is stable against a concurrent add operation.

	Stability	Stable against concurrent operation		
		$add(n_2, p_2)$	$remove(n_2)$	$move(n_2, p_2')$
Operations	$add(n_1, p_1)$	$n_1 \neq n_2$	$p_1 \neq n_2$	$true$
	$remove(n_1)$	$n_1 \neq p_2$	$true$	$n_1 \neq p_2'$
	$move(n_1, p_1')$	$true$	$p_1' \neq n_2$	$p_1' \not\rightarrow^* n_2$

Table 2: Stability analysis of sequential specification

Concurrent remove: Consider the sequential specification of two remove operations $move(n_1, p_1')$ and $remove(n_2)$. Using (26), we get

$$
\begin{aligned}
Pre_{move(n_1,p_1')} &\triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p_1' \in Nodes \wedge p_1' \neq n_1 \wedge p_1' \not\rightarrow^* n_1 \\
Pre_{remove(n_2)} &\triangleq n_2 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n_2 \\
Post_{remove(n_2)} &\triangleq n_2 \notin Nodes \\
\frac{Inv \wedge Pre_{move(n_1,p_1')} \wedge Pre_{remove(n_2)} \wedge \; n_2 \neq p_1' \qquad [\![remove(n_2)]\!]}{Inv \wedge Post_{remove(n_2)} \qquad \wedge Pre_{move(n_1,p_1')}} &
\end{aligned} \tag{37}
$$

Observe here that removing n_2 violates the clause $p_1' \in Nodes$ if n_2 and p_1' are the same. However, in our refined specification, the postcondition of remove is $n_2 \in TS$, keeping the clause $p_1' \in Nodes$ stable.

Concurrent move: Consider the sequential specification of two operations $move(n_1, p_1')$ and $move(n_2, p_2')$. Using (26), we get

$$
\begin{aligned}
Pre_{move(n_1,p_1')} &\triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p_1' \in Nodes \wedge p_1' \neq n_1 \wedge p_1' \not\rightarrow^* n_1 \\
Pre_{move(n_2,p_2')} &\triangleq n_2 \in Nodes \wedge n_2 \neq root \wedge p_2' \in Nodes \wedge p_2' \neq n_2 \wedge p_2' \not\rightarrow^* n_2 \\
Post_{move(n_2,p_2')} &\triangleq n_2 \rightarrow p_2' \\
\frac{Inv \wedge Pre_{move(n_1,p_1')} \wedge Pre_{move(n_2,p_2')} \wedge \; p_1' \not\rightarrow^* n_2 \qquad [\![move(n_2, p_2')]\!]}{Inv \wedge Post_{move(n_2,p_2')} \wedge Pre_{move(n_1,p_1')}} &
\end{aligned} \tag{38}
$$

We see here that a concurrent move of p_1 or an ancestor of p_1 invalidates the precondition clause $p_1' \not\rightarrow^* n_1$ that prevents a cycle from forming. This is a subtle condition missed in many previous works [11, 15, 17]; hence it highlights the value of a formal analysis. We discuss this condition in more detail in Subsection 4.2 and explain how we refine the specification for stability.

Table 2 shows the summary of the stability analysis on the sequential specification discussed in Section 3. A condition indicates that the precondition of the operation in that row is stable under the operation in the column under the condition.

4.2 Safety of concurrent moves

We closely examine how a move operation on a remote replica might affect the precondition of a concurrent move in the local replica. Consider an operation $move(n, p')$. In a sequential execution, precondition clause $p' \not\rightarrow^* n$ forbids moving a node under itself (which would cause a cycle). However a concurrent move of p' under n will not preserve the precondition of the operation, $p' \not\rightarrow^* n$, resulting in a cycle.

Property name	Definition
critical_ancestors	$\{a \in Nodes \,.\, p' \rightarrow^* a \wedge n \not\rightarrow^* a\}$
critical_descendants	$\{d \in Nodes \,.\, d \rightarrow^* n\}$

Table 3: Critical ancestors and critical descendants of $move(n, p')$

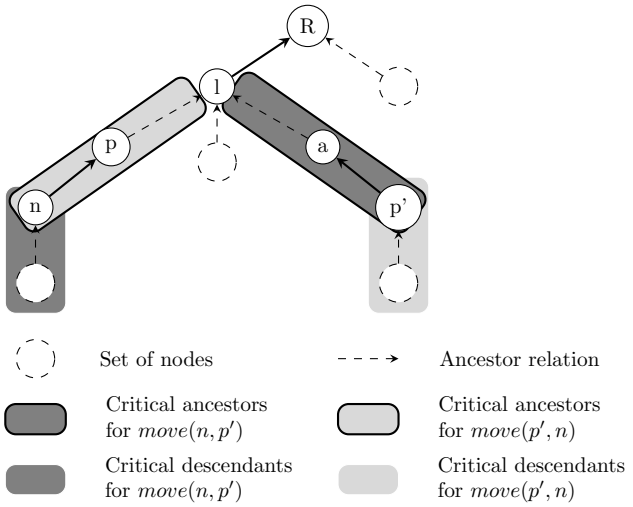


Figure 2: Critical ancestors and critical descendants

This issue generalizes to p' or its ancestor concurrently moving under n or a descendant of n. For easy reference, we call this move as a *cycle-causing-concurrent-move*. Observe that the precondition prevents an ancestor of n moving under itself in sequential execution. Therefore, only the ancestors of p' that are not ancestors of n would lead to a cycle. We call this set of ancestors *critical ancestors*, and the set of n and its descendants *critical descendants* as defined in Table 3.

Consider two concurrent move operations $move(n, p')$ and $move(p', n)$. Figure 2 shows the critical ancestors and critical descendants of both move operations. The node l is common ancestor of both n and p' farthest from the root. The critical ancestors and critical descendants of $move(n, p')$ are grouped together in the dark gray region with and without a border respectively, and that of $move(p', n)$ are grouped together in the light gray region.

Note that the set of critical descendants of a move overlaps with the critical ancestors set of its corresponding cycle-causing-concurrent-move. Hence, we consider only the critical ancestors of move operations.

Let us take a step back and analyze the types of move operations. Some move operations result in a node moving farther away from the root, called *down-move*s, and another set of move operations result in the node moving nearer to the root, or to remain at the same distance from the root, called *up-move*s. We define *rank* as the distance of a node from the root node, as follows:

$$rank(root) = 0 \tag{39}$$

$$rank(n) = rank(p) + 1 \mid \forall n, p \in Nodes \,.\, n \rightarrow p \tag{40}$$

$$up\text{-}move(n, p') \implies rank(n) > rank(p') \tag{41}$$

$$down\text{-}move(n, p') \implies rank(n) \leq rank(p') \tag{42}$$

Commutativity	Operations		
	$add(n_2, p_2)$	$rem(n_2)$	$move(n_2, p_2')$
$add(n_1, p_1)$	✔	✔	✔
$rem(n_1)$	✔	✔	✔
$move(n_1, p_1')$	✔	✔	$\neg(n_1 = n_2 \wedge p_1' \neq p_2')$

Table 4: Result of commutativity analysis of the sequential specification discussed in Section 3

Consider a move operation, $move(n, p')$, moving node n at the same level or towards the root, i.e., an up-move. This gives us that $rank(n) > rank(p')$. In this case, the rank of a critical descendant will be always greater than the rank of a critical ancestor. Formally,

$$\begin{aligned} &\forall n, p, p', d, a \in Nodes \centerdot n \rightarrow p \wedge rank(n) > rank(p') \\ &\wedge d \rightarrow^* n \wedge p' \rightarrow^* a \implies rank(d) > rank(a) \end{aligned} \tag{43}$$

This implies that a cycle-causing-concurrent-move can only be a down-move. Hence, we have that concurrent up-moves are safe; stability issues can occur only between two concurrent down-moves, or between an up-move and a down-move.

Our next step is to design a coordination-free conflict resolution policy for the moves that conflict. The conflict resolution policy is required if both the concurrent move operations move a node in the set of critical ancestors of the other. If we have up-moves, we apply the effect of the operation. In case of a concurrent down-move and up-move, up-move wins and the down-move is skipped. In case of concurrent down-moves, we apply a deterministic conflict resolution policy; the operation with highest *priority number* wins. The priority number of a move operation is specific to each application, with a condition that it must be unique for each move.

Contrast our approach with the alternative that uses shared-exclusive locks for concurrent moves [15]. Consider concurrent operations $move(n, p')$, moving node n under p', and $move(p', n)$, moving node p' under n. These operations compete for a lock. The one that succeeds first will apply its move, blocking the other. When it releases the lock, this releases the second one, but its precondition is no longer valid and it cannot execute. Thereby, safety is preserved, at the cost of aborting the second move. This work essentially achieves the same end result, but without the overhead of locking. Our experiments in Section 5 show the performance difference.

4.3 Convergence

As discussed in Section 2.2, to ensure convergence, we design the data structure such that concurrent updates commute [16]. Add and remove operations result in adding the added and removed node to *Nodes* and *TS* respectively. Since set union is commutative, each of these two operations commutes with itself and with the other.

The move operation changes the parent pointer of a node. It commutes with add and remove, since it doesn't have an effect on set membership.

However, observe that in the sequential specification two moves do not commute, if the same node is moved to two different places. This issue is fixed by the conflict resolution policy discussed earlier. The results of the commutativity analysis is show in Table 4.

4.4 Independence

We use the independence conditions from Subsection 2.2.2 to check for safety violations due to tentative moves. We check whether each operation is independent of up-move and down-move since they are the only operations that have tentative effects. For the dependent operations, we compute the condition under which it is dependent and use it to devise dependency resolution policies. In order to compute dependency conditions, we use the dependency analysis proposed by Houshmand and Lesani [8]. An operation op_2 is dependent on op_1 if the execution of op_1 enabled Pre_{op_2} that was not enabled before its execution, i.e.,

$$\frac{Inv \wedge Pre_{op_1} \qquad [\![op_1]\!]}{Inv \wedge Post_{op_1} \wedge Pre_{op_2}} \tag{44}$$

If the dependency condition evaluates to `true`, then op_2 is independent of op_1. We use this analysis to check the independence of add, remove, up-move and down-move with respect to a historical tentative operation, i.e., an up-move or down-move performed before the operation under observation.

4.4.1 Independence of add operation

Historical up-move: We use Equation 4.4 as follows:

$$\begin{aligned} Pre_{up\text{-}move(n_1,p_1')} &\triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p_1' \in Nodes \wedge\ n_1 \neq p_1' \wedge p_1' \not\rightarrow^* n_1 \wedge rank(n_1) > rank(p_1') \\ Post_{up\text{-}move(n_1,p_1')} &\triangleq \texttt{skip} \vee n_1 \rightarrow p_1' \\ Pre_{add(n_2,p_2)} &\triangleq p_2 \in Nodes \wedge n_2 \notin Nodes \end{aligned}$$

Since the historical up-move doesn't change the membership of *Nodes*, we can see that add is independent of up-move.

Historical down-move: An add operation is independent of a historical down-move in the same manner because it does not change the membership of *Nodes* either.

4.4.2 Independence of remove operation

Historical up-move: For checking the independence of remove, Equation 4.4 becomes:

$$\begin{aligned} Pre_{up\text{-}move(n_1,p_1')} &\triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p_1' \in Nodes \wedge\ n_1 \neq p_1' \wedge p_1' \not\rightarrow^* n_1 \wedge rank(n_1) > rank(p_1') \\ Post_{up\text{-}move(n_1,p_1')} &\triangleq \texttt{skip} \vee n_1 \rightarrow p_1' \\ Pre_{remove(n_2)} &\triangleq n_2 \neq root \end{aligned}$$

Since $n_2 \neq root$ is unaffected by a historical up-move, remove is independent of up-move.

Historical down-move: Similarly to historical up-move, a historical down-move also has no impact of the precondition of a remove operation. Hence remove is independent of a historical down-move.

4.4.3 Independence of up-move operation

Historical up-move: Now we analyse whether an up-move is independent of a historical up-move.

$$Pre_{up\text{-}move(n_1,p'_1)} \triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p'_1 \in Nodes \wedge\ n_1 \neq p'_1 \wedge p'_1 \not\rightarrow^* n_1 \wedge rank(n_1) > rank(p'_1)$$
$$Post_{up\text{-}move(n_1,p'_1)} \triangleq \texttt{skip} \vee n_1 \rightarrow p'_1$$
$$Pre_{up\text{-}move(n_2,p'_2)} \triangleq n_2 \in Nodes \wedge n_2 \neq root \wedge p'_2 \in Nodes \wedge\ n_2 \neq p'_2 \wedge p'_2 \not\rightarrow^* n_2 \wedge rank(n_2) > rank(p'_2)$$

We first divide the postcondition of the historical up-move into two parts: on the one hand, `skip`, which leaves the state as it was; and on the other hand, $n_1 \rightarrow p'_1$, which changes the parent relation. Then we divide the precondition of the second up-move into two parts, $n_2 \in Nodes \wedge n_2 \neq root \wedge p'_2 \in Nodes \wedge\ n_2 \neq p'_2$, which is unaffected by the historical up-move, and $p'_2 \not\rightarrow^* n_2 \wedge rank(n_2) > rank(p'_2)$, which is potentially effected by the second part of the postcondition of the historical up-move.

Note that $Pre_{up\text{-}move(n_2,p'_2)}$ was not enabled before the execution of op_1, i.e., the execution of op_1 enabled at least one predicate $p'_2 \not\rightarrow^* n_2$ or $rank(n_2) > rank(p'_2)$. Let us consider them one at a time.

Let us derive the conditions under which moving a node to a different parent introduces an ancestor relation that enables the condition $p'_2 \not\rightarrow^* n_2$ (it was previously disabled). This means that the historical up-move operation caused a disconnection between p'_2 and n_2. This will happen only if the node being moved by the historical up-move was either n_2 or a descendant of n_2 and the new parent of the current move was either n_1 or a descendant of n_1 (the node moved by the historical up-move). Hence we have $(n_1 = n_2 \vee n_1 \rightarrow^* n_2) \wedge (p'_2 = n_1 \vee p'_2 \rightarrow^* n_1)$.

The condition $rank(n_2) > rank(p'_2)$ will be enabled after an up-move only if $rank(p'_2)$ decreased.[9] This will happen only if p'_2 was the node moved or its descendant. Hence we have that $p'_2 = n_1 \vee p'_2 \rightarrow^* n_1$.

The historical up-move either enabled one or both of the conditions. Combining them gives $p'_2 = n_1 \vee p'_2 \rightarrow^* n_1$, the condition under which an up-move, $up\text{-}move(n_2, p'_2)$, is dependent on a historial up-move, $up\text{-}move(n_1, p'_1)$.

Historical down-move: To check for independence of an up-move with a historical down-move, we have the following condition:

$$Pre_{down\text{-}move(n_1,p'_1)} \triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p'_1 \in Nodes \wedge\ n_1 \neq p'_1 \wedge p'_1 \not\rightarrow^* n_1 \wedge rank(n_1) \leq rank(p'_1)$$
$$Post_{down\text{-}move(n_1,p'_1)} \triangleq \texttt{skip} \vee n_1 \rightarrow p'_1$$
$$Pre_{up\text{-}move(n_2,p'_2)} \triangleq n_2 \in Nodes \wedge n_2 \neq root \wedge p'_2 \in Nodes \wedge\ n_2 \neq p'_2 \wedge p'_2 \not\rightarrow^* n_2 \wedge rank(n_2) > rank(p'_2)$$

We apply the same reasoning as in the previous case for the condition $p'_2 \not\rightarrow^* n_2$, obtaining $(n_1 = n_2 \vee n_1 \rightarrow^* n_2) \wedge (p'_2 = n_1 \vee p'_2 \rightarrow^* n_1)$ as the condition under which an up-move is dependent under a historical down-move.

There is a difference in the second part though; the condition $rank(n_2) > rank(p'_2)$ will be enabled after a down-move only if $rank(n_2)$ increases (not possible for a down-move to decrease the rank). This will happen only if n_2 was the node moved or its descendant, i.e., $n_2 = n_1 \vee n_2 \rightarrow^* n_1$.

Combining both the conditions, we have $((n_1 = n_2 \vee n_1 \rightarrow^* n_2) \wedge (p'_2 = n_1 \vee p'_2 \rightarrow^* n_1)) \vee (n_2 = n_1 \vee n_2 \rightarrow^* n_1)$ as the condition under which an up-move, $up - move(n_2, p'_2)$, is dependent on a historial down-move, $down\text{-}move(n_1, p'_1)$.

[9] Note that $rank(n_2)$ cannot increase since an up-move does not cause the rank of any move to increase.

4.4.4 Independence of down-move operation

Historical up-move: The pre and postconditions required to analyse the dependence of a down-move operation under a historical up-move is as follows:

$$Pre_{up\text{-}move(n_1,p_1')} \triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p_1' \in Nodes \wedge\ n_1 \neq p_1' \wedge p_1' \not\rightarrow^* n_1 \wedge rank(n_1) > rank(p_1')$$

$$Post_{up\text{-}move(n_1,p_1')} \triangleq \texttt{skip} \vee n_1 \rightarrow p_1'$$

$$Pre_{down\text{-}move(n_2,p_2')} \triangleq n_2 \in Nodes \wedge n_2 \neq root \wedge p_2' \in Nodes \wedge\ n_2 \neq p_2' \wedge p_2' \not\rightarrow^* n_2 \wedge rank(n_2) \leq rank(p_2')$$

Note that the reasoning for the up-move operation also remains valid here since the effect of both moves are the same, only their preconditions differ, only the clause comparing the ranks of the node and the new parent differs. The first part of the dependency condition remains, $(n_1 = n_2 \vee n_1 \rightarrow^* n_2) \wedge (p_2' = n_1 \vee p_2' \rightarrow^* n_1)$.

The condition $rank(n_2) \leq rank(p_2')$ will be effected only if the historical up-move decreased the rank of n_2. Hence we have the condition $n_2 = n_1 \vee n_2 \rightarrow^* n_1$.

Combining the clauses, we have $((n_1 = n_2 \vee n_1 \rightarrow^* n_2) \wedge (p_2' = n_1 \vee p_2' \rightarrow^* n_1)) \vee (n_2 = n_1 \vee n_2 \rightarrow^* n_1)$, the condition under which a down-move is dependent on a historical up-move.

Historical down-move: We consider the following pre and postconditions:

$$Pre_{down\text{-}move(n_1,p_1')} \triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p_1' \in Nodes \wedge\ n_1 \neq p_1' \wedge p_1' \not\rightarrow^* n_1 \wedge rank(n_1) \leq rank(p_1')$$

$$Post_{down\text{-}move(n_1,p_1')} \triangleq \texttt{skip} \vee n_1 \rightarrow p_1'$$

$$Pre_{down\text{-}move(n_2,p_2')} \triangleq n_2 \in Nodes \wedge n_2 \neq root \wedge p_2' \in Nodes \wedge\ n_2 \neq p_2' \wedge p_2' \not\rightarrow^* n_2 \wedge rank(n_2) \leq rank(p_2')$$

We use the reasoning as in the previous cases on these ang get $p_2' = n_1 \vee p_2' \rightarrow^* n_1$, the condition under which a down-move, $down\text{-}move(n_2, p_2')$, is dependent on a historial down-move, $down\text{-}move(n_1, p_1')$.

We see that up-move and down-move operations are dependent on each other and add and remove are independent of up-move and down-move. We also derived the conditions under which up-moves and down-moves are dependent on each other. We use this information to design dependence resolution policies.

4.5 Safe specification of a replicated tree

We incorporate the stability, commutativity, and independence analysis results and the design refinements, resulting in the coordination-free, safe and convergent replicated tree data structure specified in Specification 3. The state now consists of a set of nodes, *Nodes*, and tombstones, *TS*. Since the tombstones also form part of the tree, they also have to maintain the tree structure. The invariants refer to the set of nodes which includes tombstones.

We also introduce some definitions to help define the coordination-free and conflict-free up-move and down-move operations. We define an operation as a tuple consisting of its type (add, remove, up-move or down-move), its parameters, and its priority. The priority is arbitrary (e.g. supplied by the application); the only condition being that priorities are totally ordered. We define `C` as the set of operations concurrent with the operation under consideration. `H` is the set of operations seen by the current operation. We also define operations on critical ancestors as *crit-anc-overlap*, where the node being moved is a member of the set of critical ancestors of the other operation. *self-or-under* indicates the node itself and its descendants.

With the help of these definitions, we define the up-move and down-move operations in three parts: the actual precondition needed to ensure sequential safety, the conflict resolution condition (highlighted in light blue), the dependency condition (highlighted in dark blue), and

State: $Nodes \times TS$

Invariant:

$$root \rightarrow root \wedge \forall n \in Nodes \,.\, root \not\rightarrow n \wedge root \notin TS \qquad (Root)$$

$$\wedge\ \forall n \in Nodes \,.\, n \neq root \wedge \exists p \in Nodes \,.\, n \rightarrow p \qquad (Parent)$$

$$\wedge\ \forall n, p, p' \in Nodes \,.\, n \rightarrow p \wedge n \rightarrow p' \implies p = p' \qquad (Unique)$$

$$\wedge\ \forall n \in Nodes \,.\, n \neq root \implies n \rightarrow^* root \qquad (Reachable)$$

Add operation:

(Add-Operation)

$$\frac{Inv \wedge p \in Nodes \wedge n \notin Nodes \qquad [\![add(n,p)]\!]}{Inv \wedge n \in Nodes \wedge n \rightarrow p}$$

Remove operation:

(Remove-Operation)

$$\frac{Inv \wedge n \neq root \qquad [\![remove(n)]\!]}{Inv \wedge n \in TS}$$

Definitions:

$$operation \triangleq (type, params, priority)$$

$$\texttt{C} \triangleq set\ of\ concurrent\ operations$$

$$\texttt{H} \triangleq history\ of\ operations\ available\ at\ the\ origin\ replica$$

$$crit\text{-}anc\text{-}overlap(op_1, op_2) \triangleq op_1.params.n \in critical\_ancestor(op_2) \wedge op_2.params.n \in critical\_ancestor(op_1)$$

$$self\text{-}or\text{-}under(n) \triangleq \{n' | n' = n \vee (n' \in Nodes \wedge n' \rightarrow^* n)\}$$

Move operation:

(Up-move-Operation)

$$\frac{\begin{array}{c} Inv \wedge n \in Nodes \wedge n \neq root \wedge p' \in Nodes \wedge\ n \neq p' \wedge p' \not\rightarrow^* n \wedge rank(n) > rank(p') \\ [\![up\text{-}move(n,p')]\!] \end{array}}{\begin{array}{l} \nexists op \in \texttt{C} \,.\, op.type = up\text{-}move \wedge\ op.params.n = n \wedge\ op.priority > priority \\ \nexists op \in \texttt{H} \,.\, (op.type = up\text{-}move \wedge\ \ p' \in self\text{-}or\text{-}under(op.params.n)) \\ \quad \vee (op.type = down\text{-}move \wedge\ \ (n \in self\text{-}or\text{-}under(op.params.n) \\ \qquad \vee\ (op.params.n \in self\text{-}or\text{-}under(n) \\ \qquad\quad \wedge\ p' \in self\text{-}or\text{-}under(op.params.n)))) \qquad \implies Inv \wedge n \rightarrow p' \end{array}}$$

(Down-move-Operation)

$$\frac{\begin{array}{c} Inv \wedge n \in Nodes \wedge n \neq root \wedge p' \in Nodes \wedge\ n \neq p' \wedge p' \not\rightarrow^* n \wedge rank(n) \leq rank(p') \\ [\![down\text{-}move(n,p')]\!] \end{array}}{\begin{array}{l} \nexists op \in \texttt{C} \,.\, op.type = up\text{-}move \\ \quad \wedge\ (crit\text{-}anc\text{-}overlap(down\text{-}move(n,p'), op) \vee\ \ op.params.n = n) \\ \wedge \nexists op \in \texttt{C} \,.\, op.type = down\text{-}move \\ \quad \wedge\ (crit\text{-}anc\text{-}overlap(down\text{-}move(n,p'), op) \vee\ \ op.params.n = n) \\ \quad \wedge\ op.priority > priority \\ \nexists op \in \texttt{H} \,.\, (op.type = up\text{-}move \wedge\ \ (n \in self\text{-}or\text{-}under(op.params.n) \\ \qquad \vee\ (op.params.n \in self\text{-}or\text{-}under(n) \\ \qquad\quad \wedge\ p' \in self\text{-}or\text{-}under(op.params.n)))) \\ \quad \vee (op.type = down\text{-}move \wedge\ \ p' \in self\text{-}or\text{-}under(op.params.n)) \qquad \implies Inv \wedge n \rightarrow p' \end{array}}$$

Figure 3: Concurrent specification of Maram

the update on the state. Note that the conflict resolution and dependency checks are performed while applying the effect of the operation on the local and remote replicas, while the precondition is checked only at the local replica.

4.6 Mechanized verification of the concurrent specification

We use the CISE3 plug-in, presented in Section 2.2.3, to identify conflicts as shown in Tables 2 and 4. Given the sequential specification from Section 3, CISE3 automatically generates a set of meta-operations to check stability and commutativity of executing pairs of operations.

4.6.1 Provable concurrent execution

We update the Why3 specification according to the conflict resolution policies from Section 4.5. For example, for the add operation we place the new precondition that nodes must be uniquely identified:

```
assume { ... ∧ n1 ≠ n2 }
```

Next, we refine the definition of type `state` to include tombstones, as follows:

```
type state = { mutable nodes: fset elt; ...;
      mutable tombstones: fset elt; }
```

We update the specification of the `rem` accordingly:

```
val rem (n : elt) (s : state) : unit
  ensures  { s.tombstones = add n (old s).tombstones }
```

where `add` stands for the logical adding operation on sets.

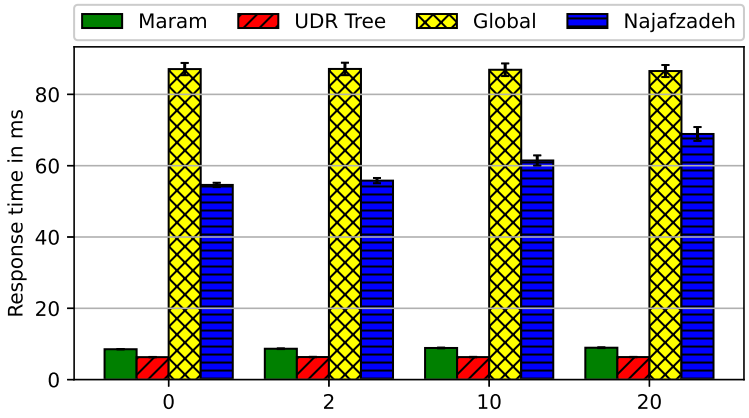
Finally, the implementation of the conflict resolution policy for a pair of `move` operations requires us to be a bit more creative. We update the `state` type definition to include ranking and critical ancestors information. We implement a custom analysis `move_refined` operation since concurrent operations are not available off-the-shelf in Why3, a framework for verification of sequential specifications. We encode the arguments of two move operations as arguments of the `move_refined` operation: `n1` (`n2`), `np1` (`np2`), and `pr1` (`pr2`) stand for the node to be moved, the new parent, and the unique priority levels respectively, of the first (second) move.

All analysis functions, except `move_refined`, are automatically generated by the CISE3 plug-in of Why3. Finally, 55 verification conditions are generated for the implementation and given specification of `move_refined`. All of these are automatically verified, using a combination of SMT solvers. The specification and the proof results are available at [14].

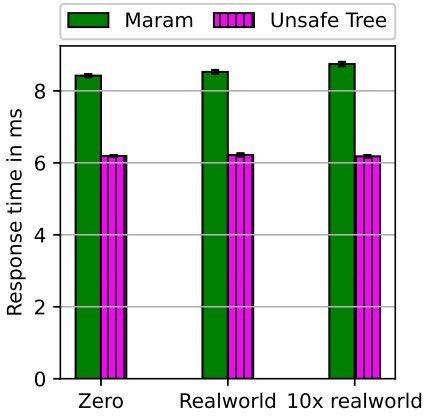
5 Evaluation

This paper presents the design of a coordination-free, safe, convergent, and highly available replicated tree. The specification of Maram doesn't require any synchronization to execute an operation; this implies that the design is coordination-free. Sections 3 and 4 provide a mechanized proof that our design is safe and convergent. In this section, we conduct an evaluation to showcase the high availability of our design.

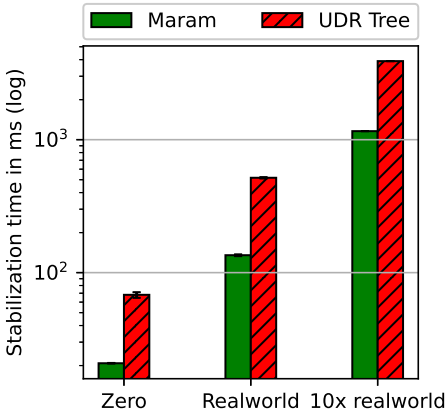
We measure availability in two parts - *response time* and *stabilization time*. The first metric, *response time*, is the time taken to log and acknowledge a client request. Recall that the effect of a move operation in our specification consists of either updating the state, or a skip. The effect of the update will be definitive only after being aware of all its concurrent operations. In order



(a) Response time for different conflict rates



(b) Overhead of conflict resolution for different latencies



(c) Stabilization time for different latencies in logarithmic scale

Figure 4: Experimental results. Each bar is the average of 15 runs, the error bars show standard deviation

Latency	Replicas		
	Paris	Bangalore	New York
Paris	0	144	75
Bangalore	144	0	215
New York	75	215	0

Table 5: Real world latency configurations in ms

to measure this, we introduce a metric called *stabilization time*. Stabilization time measures the duration for which an update is in a transient state.

We run the experiments[10] with three replicas connected in a mesh with a FIFO connection and simulate different network latencies, zero latency, real world latency as shown in Table 5 and 10 times real world latency. Our warm-up workload, a mix of add, remove and move operations, creates a tree with 997 nodes including the root. We then have concurrent workloads[11] on the three replicas, varying conflict rates at 0%, 2%, 10%, and 20%.

We compare Maram with three solutions from the literature: (i) UDR tree (short for Undo–Do-Redo tree) [11]; (ii) all move operations acquiring a global lock (Global_l); and, (iii) move operations acquiring read locks on critical ancestors and write lock on the moving node (Subtree_l) [15].

The average response time for each design for different conflict rates with latency configuration 2 (Table 5) are shown in Figure 4a. Observe that Maram and UDR tree show similar average response time across different conflict rates; owing to the synchronization-free design. Maram has a slightly higher response time than UDR tree due to the overhead of metadata calculation. The response time for Subtree_l [15] increases with an increase in the conflict rate due to lock contention, whereas that of Global_l is the same across all conflict rates since the proportion of lock-acquiring-moves remains the same.

Figure 4c shows the average stabilization time for our design and the UDR tree design [11] on a logarithmic scale, for different latency configurations. Our solution gives lower stabilization time, since only down-moves and a very few up-moves have transient state in the case of Maram whereas for a UDR tree [11] all operations are in transient state until a local replica asserts that there are no more concurrent operations.[12]

Next, we run an experiment to measure the overhead introduced by the conflict resolution policy. As a lower bound, we compare the response time of Maram with a naïve unsafe implementation, that uses a simple eventual consistency approach, and thus is not safe. Figure 4b shows the response time of both the designs. For Maram metadata calculation implies slightly higher response time.

6 Related work

Several works have addressed the problem of designing a replicated tree. Martin et al. [12] introduce some designs for conflict-free replicated tree data types. They use set CRDTs to

[10] On DELL PowerEdge R410 machine with 64 GB RAM, and 24 cores @2.40GHz Intel Xeon E5645 processor.

[11] 250 operations per replica - 60% add, 12% remove, 14% upmove and 14% downmove.

[12] Note here that Maram's stabilisation time does not depend on conflict rate, but only on the proportion of moves in the workload. As this proportion grows towards 100%, the stabilisation time of Maram tends to be the same as that of UDR.

Independent		Under		
		$add(n_2, p_2)$	$remove(n_2)$	$move(n_2, p_2')$
Operation	$add(n_1, p_1)$	$p_1 \neq n_2$	*true*	*true*
	$remove(n_1)$	$n_1 \neq n_2$	*true*	*true*
	$move(n_1, p_1')$	$n_1 \neq n_2 \vee p_1' \neq n_2$	*true*	$n_2 \notin self\text{-}or\text{-}under(n_1)$ $\vee p_1' \notin self\text{-}or\text{-}under(n_2)$

Table 6: Result of dependency analysis. The cell shows the condition under which the operation in the row is independent of the operation in the column.

construct replicated trees with different semantics. Add and remove operations are supported in their design. However they do not consider move operations.

Kleppmann et al. [11] propose the UDR tree, which supports atomic move operations, using the notion of opsets. Opsets totally order all operations eventually. This is more expensive than our solution based on partial order. When a new operation is performed, all the later operations are redone. Thus all operations pay a heavy price, and not just the conflicting moves. But UDR requires eventually consistent delivery layer, whereas Maram requires a more expensive causal delivery layer.

Compared to the work of Kleppmann et al. [11], we also show under what conditions a move operation might skip.

Najafzadeh et al. [15] designs the replicated tree called Subtree_l in Section 5. Their solution introduces coordination; acquires read locks on the critical ancestors, and a write lock on the node being moved. This approach is not available under partition, but only move operations pay an overhead.

Tao et al. [17] propose a replicated tree with a move operation that does not require any coordination between replicas, replacing each move with non-atomic copy and delete operations. This might lead to having multiple copies of the same node.

Compared to all the above solutions, our design supports atomic move operation that depends on partial ordering without acquiring any locks. An atomic update provides all or no guarantee, i.e., either the update is applied or it is not. Ensuring atomicity avoids partial execution of updates.

Kaki et al. [10] introduce the concept of Mergeable Replicated Data Types (MRDTs) inspired by three-way-merge. The safety of an MRDT binary tree depends on the labeling of the child-parent relations (whether it belongs to the right or left of the ancestor). It also requires to keep track of all the ancestor relations apart from the parent-child relations. A generic MRDT tree can be considered as an extension to the MRDT binary tree, but requires tracking all ancestor relations and a complex lexicographical ordering when concretizing the merged result.

7 Discussion

7.1 Moving from causal consistency to eventual consistency

Houshmand and Lesani [8] propose dependency analysis to help relax the requirement of causal delivery. We run this analysis for all operations, irrespective of whether the update is tentative or definitive.

Table 6 shows the results of the dependency analysis of Maram. We can observe that no operations are dependent on remove, and add and remove are not dependent on move. As there is no fully independent operation, relaxing causal delivery is not helpful to Maram.

7.2 Message overhead for conflict resolution

In order to use Maram in a real-world application, we need to understand the overhead of conflict resolution. Conflict resolution requires some meta information that is sent along with the update message from the origin replica. This may have an impact on the bandwidth lost, hence understanding the components is important.

The conflict resolution policy of Maram needs information to compute a set of concurrent operations. Assuming a replica works as a single threaded process, we use vector clocks. The size of vector clocks is linear with the number of replicas. This poses an additional overhead.

Conflict resolution also takes as input the set of critical ancestors, descendants, and the priority.The size of the set of critical ancestors depends on the depth of the subtree comprising the least common ancestor of the node being moved and the destination parent. The size of the set of critical ancestors is linear to the difference in the rank of the new parent and the least common ancestor. The size of the set of descendants might be large for the nodes nearer to the root. This poses an overhead on the metadata. The priority can be a single number or a string and is independent of other factors. Hence using the conflict resolution of Maram will cause a considerable overhead on message delivery.

The time taken to compute this metadata is the difference between the response time of a naïve unsafe replicated tree and Maram in Figure 4b.

7.3 Computing the set of concurrent moves

Maram requires a set of concurrent operations to apply the conflict resolution. For this, the Maram system layer does not busy-wait. Every replica makes progress locally, without waiting to receive remote logs (availability under partition). Conflict resolution applies only after a replica receives a concurrent conflicting operation.

To conclude, Maram is a safe, coordination-free replicated tree, designed using conflict resolution policies.

8 Conclusion

This paper presents the design of a light-weight, coordination-free, safe, convergent and highly available replicated tree data structure, Maram. We provide mechanized proof of safety and convergence of Maram, and experimentally demonstrate the efficiency of the design by comparing it with the existing solutions.

References

[1] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 259–268, Chicago, IL, USA, July 2016. Assoc. for Computing Machinery, Assoc. for Computing Machinery. doi: 10.1145/2933057.2933090. URL http://dx.doi.org/10.1145/2933057.2933090.

[2] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based CRDTs operation-based. In Kostas Magoutis and Peter Pietzuch, editors, *Int. Conf. on Distr. Apps. and Interop. Sys. (DAIS)*, volume 8460 of *Lecture Notes in Comp. Sc.*, pages 126–140, Berlin, Germany, June 2014. Int. Fed. for Info. Processing (IFIP), Springer-Verlag. doi: 10.1007/978-3-662-43352-2_11. URL http://dx.doi.org/10.1007/978-3-662-43352-2_11.

[3] Nikolaj Bjørner. Models and software model checking of a distributed file replication system. In *Formal Methods and Hybrid Real-Time Systems*, pages 1–23, 2007. URL http://dx.doi.org/10.1007/978-3-540-75221-9_1.

[4] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975. doi: 10.1145/360933.360975. URL https://doi.org/10.1145/360933.360975.

[5] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397, Aug 2011. ISSN 1433-2787. URL https://doi.org/10.1007/s10009-011-0211-0.

[6] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792 of *LNCS*, Rome, Italy, March 2013. Springer. URL https://hal.inria.fr/hal-00789533.

[7] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm Strong Enough: Reasoning about consistency choices in distributed systems. In *Symp. on Principles of Prog. Lang. (POPL)*, pages 371–384, St. Petersburg, FL, USA, 2016. Assoc. for Computing Machinery. doi: 10.1145/2837614.2837625. URL http://dx.doi.org/10.1145/2837614.2837625.

[8] Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication coordination analysis and synthesis. *Proc. ACM Program. Lang.*, 3(POPL):74:1–74:32, January 2019. ISSN 2475-1421. URL http://doi.acm.org/10.1145/3290387.

[9] Gowtham Kaki, Kapil Earanky, K. C. Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. In *Conf. on Object-Oriented Prog. Sys., Lang. and Applications (OOPSLA)*, Proc. ACM Program. Lang., pages 164:1–164:27, Boston, MA, USA, November 2018. doi: 10.1145/3276534. URL https://doi.org/10.1145/3276534.

[10] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. URL https://doi.org/10.1145/3360580.

[11] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. Opsets: Sequential specifications for replicated datatypes (extended version). *CoRR*, abs/1805.04263, 2018. URL http://arxiv.org/abs/1805.04263.

[12] Stéphane Martin, Mehdi Ahmed-Nacer, and Pascal Urso. Abstract unordered and ordered trees CRDT. Research Report RR-7825, INRIA, December 2011. URL https://hal.inria.fr/hal-00648106.

[13] Filipe Meirim, Mário Pereira, and Carla Ferreira. CISE3: Verifying weakly consistent applications with why3, 2020.

[14] Filipe Meirim, Mário Pereira, Carla Ferreira, Sreeja Nair, and Marc Shapiro. Maram proof files. https://fmeirim.github.io/Maram_proofs/, 2021.

[15] Mahsa Najafzadeh, Marc Shapiro, and Patrick Eugster. Co-design and verification of an available file system. In Işıl Dillig and Jens Palsberg, editors, *Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 10747 of *Lecture Notes in Comp. Sc.*, pages 358–381, Los Angeles, CA, USA, January 2018. Assoc. for Computing Machinery Special Interest Group on Pg. Lang. (SIGPLAN), Springer-Verlag. doi: 10.1007/978-3-319-73721-8_17. URL https://doi.org/10.1007/978-3-319-73721-8_17.

[16] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag. doi: 10.1007/978-3-642-24550-3_29. URL https://doi.org/10.1007/978-3-642-24550-3_29.

[17] Vinh Tao, Marc Shapiro, and Vianney Rancurel. Merging semantics for conflict updates in geo-distributed file systems. In *ACM Int. Systems and Storage Conf. (Systor)*, pages 10.1–10.12, Haifa, Israel, May 2015. Assoc. for Computing Machinery. doi: 10.1145/2757667.2757683. URL http://dx.doi.org/10.1145/2757667.2757683.

[18] Vinh Thanh Tao. *Ensuring Availability and Managing Consistency in Geo-Replicated File Systems*. PhD thesis, Sorbonne-Université–Université Pierre et Marie Curie, Paris, France, December 2017. URL https://hal.inria.fr/tel-01673030.

[19] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149, Austin, Texas, USA, September 1994.

A Specification of sequential tree in Why3

This section presents the Why3 specification of a sequentially safe tree.

```
use export int.Int
use import map.Map as M
use import set.Fset as F
use seq.Seq, seq.Mem, seq.Distinct

(* auxiliary lemmas on sequences *)
lemma append_empty: forall s: seq 'a.
  s ++ empty == s

lemma empty_length: forall s: seq 'a.
  length s = 0 ↔ s == empty

predicate disjoint_seq (s1 s2: seq 'a) =
  forall i j. 0 ≤ i < length s1 →
               0 ≤ j < length s2 → s1[i] ≠ s2[j]

(* Arbitrary type for a tree node *)
type elt

(* Verifies if two nodes are equal *)
val equal (e1 e2 : elt) : bool
  ensures { result ↔ e1 = e2 }

(* Indicates if two nodes are connected by an edge in the tree *)
predicate edge (x y : elt) (f : elt → elt) =
  x ≠ y ∧ f x = y

(* recursive predicate for expressing a path between two nodes *)
(* in the main text a few cosmetic changes were done, namely *)
(* rename of -f- to -parent- and expand the edge definition *)
predicate path (f: elt → elt) (x y: elt) (p: seq elt) =
  let n = length p in
  n = 0 ∧ x = y
   ∨
  n > 0 ∧
  p[0] = x ∧
  edge p[n - 1] y f ∧
  distinct p ∧
  (forall i. 0 ≤ i < n - 1 → edge p[i] p[i + 1] f) ∧
  (forall i. 0 ≤ i < n     → p[i] ≠ y)

predicate reachability (f: elt → elt) (x y: elt) =
  exists p. path f x y p

(* If there is an edge between nodes x and y
(* the path is defined as the singleton seq with node x  *)
axiom path_to_parent: forall x y : elt, f : elt → elt.
  edge x y f → path f x y (cons x empty)

(* If there is a path from [from] to [middle] and a path from [middle] to
```

```
  [until] then there is a path from [from] to [until] *)
axiom path_transitivity: forall from middle until f pth1 pth2.
  path f from middle pth1 → path f middle until pth2 →
  disjoint_seq pth1 pth2 → from ≠ until →
  (forall j. 0 ≤ j < length pth1 → pth1[j] ≠ until) →
        path f from until (pth1 ++ pth2)

(* Recursive path composition *)
axiom path_composition: forall n x y: elt, f : elt → elt, pth : seq elt.
  n ≠ y → not (mem y pth) →
  distinct (snoc pth x) → path f n x pth → edge x y f →
  path f n y (snoc pth x)

 (* If there is a path between two nodes, that path is unique *)
axiom path_uniqueness: forall x y: elt, f: elt → elt,
                               pth1 pth2: seq elt.
  path f x y pth1 → path f x y pth2 → pth1 == pth2

 (* If node np is not reachable to node c, then np will
    not belong to any path that contains node c *)
axiom path_exclusion: forall f x c np p.
  not (reachability f np c) → path f x c p → not (mem np p)

 (* Given a path between two nodes, there is no overlap between any two consecutive
    subpaths *)
axiom path_separation: forall final initial middle : elt, f : elt → elt,
                               p1 p2 : seq elt.
  path f middle final p2 → path f initial middle p1 →
  final ≠ initial → middle ≠ initial → middle ≠ final →
  disjoint_seq p1 p2

constant n: elt (* constant used for defining a state witness *)

type state [@state] = {
  (* parent relation: up-pointers to direct ancestor *)
  mutable parent : elt → elt;
  (* parent root *)
  mutable root   : elt;
  (* nodes in the parent *)
  mutable nodes  : fset elt;
} invariant { F.mem root nodes }
  invariant { parent root = root }
  invariant { forall x. F.mem x nodes → F.mem (parent x) nodes }
  invariant { forall x. F.mem x nodes → reachability parent x root }
  invariant { forall x. F.mem x nodes →
                reachability parent root x → x = root }
  by { parent = (fun _ → n); root = n; nodes = F.singleton n }

(* Paths already present in the tree remain in the tree after executing
   the add operation *)
axiom remaining_nodes_add: forall n w p: elt, s: state, l: seq elt.
  path s.parent w s.root l → not (mem n l) →
  F.mem w s.nodes → F.mem p s.nodes → not (F.mem n s.nodes) →
```

```
  w ≠ n → n ≠ p → path (M.set s.parent n p) w s.root l

(* Descendants of the node being moved continue to be its descendants *)
axiom descendants_move: forall x c np: elt, f: elt → elt, p: seq elt.
  x ≠ np → c ≠ np → x ≠ c → not (reachability f np c) →
  path f x c p → distinct (cons c p) → not (mem np p) →
  (path (M.set f c np) x c p)

(*  Paths nodes unreachable to the node being moved are not affected *)
axiom remaining_nodes_move: forall x c np: elt, s: state, p: seq elt.
  c ≠ np → x ≠ c → not (reachability s.parent np c) →
  path s.parent x s.root p → (not reachability s.parent x c) →
  distinct p → (path (M.set s.parent c np) x s.root p)

let ghost add (n p : elt) (s : state) : unit
  requires { [@expl:pre_add1] not F.mem n s.nodes }
  requires { [@expl:pre_add2] F.mem p s.nodes }
  ensures  { s.parent = M.set (old s.parent) n p }
  ensures  { edge n p s.parent }
  ensures  { s.nodes = F.add n (old s).nodes }
= s.parent     ← M.set s.parent n p;
  s.nodes      ← F.add n s.nodes;

let ghost remove (n : elt) (s : state) : unit
  requires { [@expl:pre_remove1] forall x. s.parent x ≠ n }
  requires { [@expl:pre_remove2] n ≠ s.root }
  ensures  { s.nodes = F.remove n (old s).nodes }
= s.nodes ← F.remove n s.nodes;

let ghost move (c np : elt) (s : state) : unit
  requires { [@expl:pre_move1] F.mem np s.nodes }
  requires { [@expl:pre_move2] F.mem c s.nodes}
  requires { [@expl:pre_move3] not (reachability s.parent np c) }
  requires { [@expl:pre_move4] c ≠ s.root }
  requires { [@expl:pre_move5] c ≠ np }
  ensures  { edge c np s.parent }
  ensures  { s.parent = M.set (old s.parent) c np }
= s.parent ← M.set s.parent c np;
```

B CISE analysis on sequential specification

```
use export why3.BuiltIn.BuiltIn
use export why3.Bool.Bool
use export why3.Unit.Unit
use export file_system_alternative.S
use export why3.Tuple2.Tuple2
use export int.Int
use import map.Map as M
use import set.Fset as F
use seq.Seq, seq.Mem, seq.Distinct

predicate same_ext (m1 m2: 'a → 'b) = forall x: 'a. m1 x = m2 x

val equal_elt (e1 e2 : elt) : bool
 ensures {result ↔ e1 = e2}

let ghost predicate state_equality (s1 s2 : state)
 =
   same_ext s1.parent s2.parent &&
   equal_elt s1.root s2.root &&
   F.(==) s1.nodes s2.nodes

let ghost move_move_analysis (ghost _:()) : (state, state)
   ensures  { match result with
              | x1, x2 → state_equality x1 x2
              end }  =
   let ghost c1 = any elt in
   let ghost np1 = any elt in
   let ghost state1 = any state in
   let ghost c2 = any elt in
   let ghost np2 = any elt in
   let ghost state2 = any state in
   assume { (F.mem np1 state1.nodes) ∧
            (F.mem np2 state2.nodes) ∧
            (F.mem c1 state1.nodes) ∧
            (F.mem c2 state2.nodes) ∧
            (not (reachability state1.parent np1 c1)) ∧
            (not (reachability state2.parent np2 c2)) ∧
            (c1 ≠ state1.root) ∧
            (c2 ≠ state2.root) ∧
            (c1 ≠ np1) ∧
            (c2 ≠ np2) ∧
            state_equality state1 state2 };
  move c1 np1 state1;
  move c2 np2 state1;
  move c2 np2 state2;
  move c1 np1 state2;
  (state1, state2)

let ghost remove_remove_analysis (ghost _:()) : (state, state)
   ensures  { match result with
              | x1, x2 → state_equality x1 x2
```

```
                end } =
   let ghost n1 = any elt in
   let ghost state1 = any state in
   let ghost n2 = any elt in
   let ghost state2 = any state in
   assume { (F.mem n1 state1.nodes) ∧
            (F.mem n2 state2.nodes) ∧
            (forall x. state1.parent x ≠ n1 ) ∧
            (forall x. state2.parent x ≠ n2 ) ∧
            (n1 ≠ state1.root) ∧
            (n2 ≠ state2.root) ∧
           state_equality state1 state2 };
   remove n2 state1;
   remove n1 state1;
   remove n1 state2;
   remove n2 state2;
   (state1, state2)

let ghost add_add_analysis (ghost _:()) : (state, state)
   ensures  { match result with
                | x1, x2 → state_equality x1 x2
                end } =
   let ghost n1 = any elt in
   let ghost p1 = any elt in
   let ghost state1 = any state in
   let ghost n2 = any elt in
   let ghost p2 = any elt in
   let ghost state2 = any state in
   assume { ((not F.mem n1 (nodes state1) ∧ F.mem p1 (nodes state1)) ∧
            not F.mem n2 (nodes state2) ∧ F.mem p2 (nodes state2)) ∧
            state_equality state1 state2 };
   add n1 p1 state1;
   add n2 p2 state1;
   add n2 p2 state2;
   add n1 p1 state2;
   (state1, state2)

let ghost remove_move_analysis (ghost _:()) : (state, state)
   ensures  { match result with
                | x1, x2 → state_equality x1 x2
                end } =
   let ghost n1 = any elt in
   let ghost state1 = any state in
   let ghost c2 = any elt in
   let ghost np2 = any elt in
   let ghost state2 = any state in
   assume { (forall x. state1.parent x ≠ n1 ) ∧
            (n1 ≠ state1.root) ∧
            (F.mem np2 state2.nodes) ∧
            (F.mem c2 state2.nodes) ∧
            (not (reachability state2.parent np2 c2)) ∧
            (c2 ≠ state2.root) ∧
            (c2 ≠ np2) ∧
```

```
            state_equality state1 state2 };
   move c2 np2 state1;
   remove n1state1;
   remove n1 state2;
   move c2 np2 state2;
   (state1, state2)

let ghost add_move_analysis (ghost _:()) : (state, state)
   ensures  { match result with
               | x1, x2 → state_equality x1 x2
               end } =
   let ghost n1 = any elt in
   let ghost p1 = any elt in
   let ghost state1 = any state in
   let ghost c2 = any elt in
   let ghost np2 = any elt in
   let ghost state2 = any state in
   assume { (not F.mem n1 (nodes state1) ∧ F.mem p1 (nodes state1)) ∧
            (F.mem np2 state2.nodes) ∧
            (F.mem c2 state2.nodes) ∧
            (not (reachability state2.parent np2 c2)) ∧
            (not (reachability state1.parent np2 c2)) ∧
            (c2 ≠ state2.root) ∧
             (c2 ≠ np2) ∧
            state_equality state1 state2 };
   add n1 p1 state1;
   move c2 np2 state1;
   move c2 np2 state2;
   add n1 p1 state2;
   (state1, state2)

let ghost add_remove_analysis (ghost _:()) : (state, state)
   ensures  { match result with
               | x1, x2 → state_equality x1 x2
               end } =
   let ghost n1 = any elt in
   let ghost p1 = any elt in
   let ghost state1 = any state in
   let ghost n2 = any elt in
   let ghost state2 = any state in
   assume { (not F.mem n1 (nodes state1) ∧ F.mem p1 (nodes state1)) ∧
            (forall x. state2.parent x ≠ n2 ) ∧
            (n2 ≠ state2.root) ∧
            state_equality state1 state2 };
   remove n2 state1;
   add n1 p1 state1;
   add n1 p1 state2;
   remove n2 state2;
   (state1, state2)
```

B.1 Why3 proof sessions

This section presents the quantitative results of using Why3 and CISE3 to analyse the sequential operations. Each table presents the set of generated verification conditions for a specific pair of

operations, whose names appear in the head of the table. For each verification condition, we run the available SMTs until one of them is able to discharge it, or else every one fails to complete the proof. Finally, each verification condition is identified with a name of the form `lemma P`, where `P` for the nature of the condition, *i.e.*, it is either a precondition or a postcondition. For instance, on the first table, `pre_move1` stands for the first precondition of the `move` operation, and so on. Times are given in seconds.

Proof obligations		Alt-Ergo 2.3.2	CVC4 1.7	Z3 4.8.6
lemma VC for move_move_analysis	lemma pre_move1		0.10	
	lemma pre_move2		0.10	
	lemma pre_move3		0.07	
	lemma pre_move4		0.10	
	lemma pre_move5		0.11	
	lemma pre_move1		0.21	
	lemma pre_move2		0.22	
	lemma pre_move3	(1s)	(1s)	(1s)
	lemma pre_move4		0.15	
	lemma pre_move5		0.09	
	lemma pre_move1		0.11	
	lemma pre_move2		0.10	
	lemma pre_move3		0.09	
	lemma pre_move4		0.09	
	lemma pre_move5		0.11	
	lemma pre_move1		0.24	
	lemma pre_move2		0.22	
	lemma pre_move3	(1s)	(1s)	FAILURE
	lemma pre_move4		0.14	
	lemma pre_move5		0.09	
	lemma postcondition	0.26	(1s)	FAILURE

C Specification of Maram in Why3

This section presents the Why3 specification of Maram.

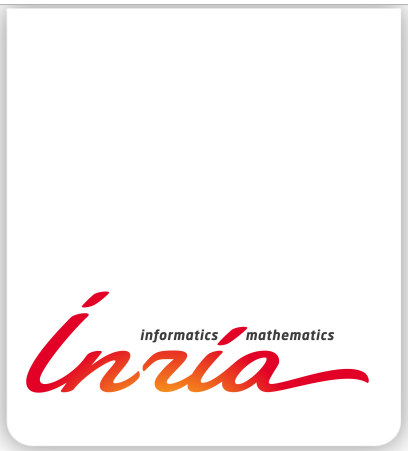
```
type state = {
  (* parent relation: up-pointers to direct ancestor *)
  mutable parent     : elt → elt;
  (* parent root *)
  mutable root   : elt;
  (* nodes in the parent *)
  mutable nodes      : fset elt;
  (* rank for each node *)
  mutable rank       : elt → int;
```

Proof obligations		Alt-Ergo 2.3.2	CVC4 1.7	Z3 4.8.6
lemma VC for remove_remove_analysis	lemma pre_remove1		0.16	
	lemma pre_remove2		0.22	
	lemma pre_remove1		0.20	
	lemma pre_remove2		0.16	
	lemma pre_remove1		0.27	
	lemma pre_remove2		0.21	
	lemma pre_remove1		0.24	
	lemma pre_remove2		0.15	
	lemma postcondition		0.10	
			0.09	
			0.23	
lemma VC for add_add_analysis	lemma pre_add1		0.11	
	lemma pre_add2		0.11	
	lemma pre_add1	(1s)	(1s)	FAILURE
	lemma pre_add2		0.22	
	lemma pre_add1		0.10	
	lemma pre_add2		0.11	
	lemma pre_add1	(1s)	(1s)	(1s)
	lemma pre_add2		0.22	
	lemma postcondition	0.51	(1s)	FAILURE
lemma VC for remove_move_analysis	lemma pre_move1		0.24	
	lemma pre_move2		0.23	
	lemma pre_move3		0.93	
	lemma pre_move4		0.15	
	lemma pre_move5		0.11	
	lemma pre_remove1	(1s)	(1s)	FAILURE
	lemma pre_remove2		0.23	
	lemma pre_remove1		0.19	
	lemma pre_remove2		0.19	
	lemma pre_move1		0.32	
	lemma pre_move2	(1s)	(1s)	FAILURE
	lemma pre_move3		0.15	
	lemma pre_move4		0.17	
	lemma pre_move5		0.11	
	lemma postcondition		0.22	
			0.16	
			0.23	

Proof obligations		**Alt-Ergo 2.3.2**	**CVC4 1.7**	**Z3 4.8.6**
lemma `VC for add_move_analysis`	lemma `pre_add1`		0.12	
	lemma `pre_add2`		0.11	
	lemma `pre_move1`		0.35	
	lemma `pre_move2`		0.41	
	lemma `pre_move3`		0.11	
	lemma `pre_move4`		0.15	
	lemma `pre_move5`		0.09	
	lemma `pre_move1`		0.10	
	lemma `pre_move2`		0.22	
	lemma `pre_move3`		0.12	
	lemma `pre_move4`		0.11	
	lemma `pre_move5`		0.11	
	lemma `pre_add1`		0.31	
	lemma `pre_add2`		0.34	
	lemma `postcondition`		0.35	
			0.20	
			0.35	
lemma `VC for add_remove_analysis`	lemma `pre_remove1`		0.17	
	lemma `pre_remove2`		0.16	
	lemma `pre_add1`		0.19	
	lemma `pre_add2`	(1s)	(1s)	FAILURE
	lemma `pre_add1`		0.25	
	lemma `pre_add2`		0.23	
	lemma `pre_remove1`		0.29	
	lemma `pre_remove2`		0.11	
	lemma `postcondition`		0.27	
			0.19	
		(1s)	(1s)	(1s)

```
  (* tombstone nodes  *)
  mutable tombstones : fset elt;
  (* ancestors relation: all ancestors of a node *)
  mutable ancestors  : elt → fset elt;
} invariant { F.mem root nodes }
  invariant { parent root = root }
  invariant { forall x. F.mem x nodes → F.mem (parent x) nodes }
  invariant { forall x. F.mem x nodes → reachability parent x root }
  invariant { forall x. F.mem x nodes →
                reachability parent root x → x = root }
  invariant { forall x y. F.mem x nodes ∧ F.mem y nodes ∧ x ≠ root ∧
                x ≠ y ∧ F.mem y (ancestors x) → rank x > rank y }
  invariant { forall x. F.mem x nodes ∧ x ≠ root →
                ancestors x = F.add (parent x) (ancestors (parent x))}
by { parent = (fun _ → n); root = n; nodes = F.singleton n ;
     rank = (fun _ → 1); tombstones = F.empty;
     ancestors = (fun _ → F.empty) }

let ghost predicate state_equality (s1 s2 : state)
= same_ext s1.parent s2.parent ∧
  equal_elt s1.root s2.root ∧
  F.(==) s1.nodes s2.nodes ) ∧
  same_ext s1.rank s2.rank ∧
  same_ext s1.ancestors s2.ancestors ∧
  F.(==) s1.tombstones s2.tombstones

val ghost add (n p : elt) (s : state) : unit
  requires { [@expl:add1] not F.mem n s.nodes }
  requires { [@expl:add2] F.mem p s.nodes }
  writes   { s.parent, s.nodes, s.rank, s.ancestors }
  ensures  { s.parent = M.set (old s.parent) n p }
  ensures  { edge n p s.parent }
  ensures  { s.nodes = F.add n (old s).nodes }
  ensures  { s.rank = M.set (old s).rank n ((s.rank p) + 1) }
  ensures  { s.ancestors = M.set ((old s).ancestors) n
               (F.add p ((old s).ancestors p)) }

val ghost remove (n : elt) (s : state) : unit
  requires { n ≠ s.root }
  writes   { s.tombstones }
  ensures  { s.tombstones = F.add n (old s).tombstones }

val ghost move (c np : elt) (s : state) : unit
  requires { F.mem np s.nodes }
  requires { F.mem c s.nodes }
  requires { not (reachability s.parent np c) }
  requires { c ≠ s.root }
  requires { c ≠ np }
  writes   { s.parent, s.ancestors, s.rank }
  ensures  { [@expl:post1] s.parent = M.set (old s.parent) c np }

let ghost move_refined (c1 np1 c2 np2 : elt) (pr1 pr2 : int)
                       (s : state) : unit
```

```
  requires { pr1 ≠ pr2 }
  requires { F.mem c1 s.nodes ∧ F.mem c2 s.nodes }
  requires { F.mem np1 s.nodes ∧ F.mem np2 s.nodes }
  requires { not (reachability s.parent np1 c1) ∧
             not (reachability s.parent np2 c2) }
  requires { c1 ≠ s.root ∧ c2 ≠ s.root }
  requires { c1 ≠ np1 ∧ c2 ≠ np2 }
  ensures  { (s.parent = M.set (old s.parent) c2 np2)  ∨
             (s.parent = M.set (old s.parent) c1 np1)  ∨
             (same_ext s.parent (old s).parent) }
= if (equal_elt c1 c2) then
  if (pr1 < pr2) then move c2 np2 s else move c1 np1 s
  else if (s.rank np1 < s.rank c1) then move c1 np1 s
  else if (F.mem c2 (diff (F.add np1 (s.ancestors np1))
           s.ancestors c1))) then
  if (s.rank np2 < s.rank c2) then ()
  else if (pr1 < pr2) then move c2 np2 s else move c1 np1 s
  else move c1 np1 s
```



RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399