Ensuring Convergence and Invariants Without Coordination (Appendix)

Dina Borrego
NOVA School of Science and Technology, Portugal
Vrije Universiteit Brussel, Belgium

Nuno Preguiça
NOVA School of Science and Technology, Portugal

Elisa Gonzalez Boix
Vrije Universiteit Brussel, Belgium

Carla Ferreira
NOVA School of Science and Technology, Portugal

A State-Dependent Policy Specification for Disjunctions

Listing 1 presents the specification of a state-based policy for maintaining a disjunction invariant. The model assumes two operations: i) `TurnTrue(toUpd)`, which sets the predicate `toUpd` to true, and ii) `TurnFalse(toUpd, toBlock)`, which sets the predicate `toUpd` to false and blocks the truth value of the predicate `toBlock`.

Listing 1 Specification of a state-based conflict resolution policy for maintaining a disjunction invariant.

```
1  def blocksEncoding(v1: TaggedOp[Time, ID, PredOp]): Boolean =
2    v1.op match {
3      case TurnTrue(_) => v1.b == new NullOp()
4      case TurnFalse(_, toBlock) => v1.b == new TurnFalse(toBlock, toBlock)
5      case _ => false
6    }
7
8  override def isBlockedBy(op1: TaggedOp[Time, ID, PredOp],
9                           op2: TaggedOp[Time, ID, PredOp]) = {
10   op1.op match {
11     case TurnFalse(b1, _) => op2.b match {
12       case TurnFalse(_, b1) => true
13       case _ => false
14     }
15     case _ => op1.op == op2.b
16   }
```

The `blocksEncoding` function defines that if the operation is `TurnTrue`, the *block* is null (represented by `NullOp`). Conversely, if the operation is `TurnFalse`, the *block* must be equal to `TurnFalse(toBlock, toBlock)`. Since pattern matching must be exhaustive, an additional case is included to handle `NullOp`, which always returns false.

As with the LWW Register, this policy also requires a custom definition of the `isBlockedBy` function. The `TurnFalse` operation blocks a specific instance of an operation. However, to correctly enforce the policy, the block must prevent all `TurnFalse` operations where the predicate being updated matches the predicate being blocked. As specified in listing 1 (lines 9–11), this condition is encoded in `isBlockedBy`: an operation in the first call (`op1.op`) is blocked if both the operation and the block of the second call (`op2.b`) are of type `TurnFalse`, and the predicate being updated (`toUpd`) in the first operation matches the predicate being blocked (`toBlock`) in the second call. If this condition is not true for `TurnFalse` operations, then the operation is not blocked. The case in line 15 ensures that pattern matching covers

all operations, including `TurnTrue` and `NullOp`. In this case, the function checks whether the operation in the first call matches the block in the second call. However, this condition will never be true because `TurnTrue` operations do not issue blocks, and `NullOp` represents null operations which do not exist in the system.

Note that two operations running on different replicas in the same state may trigger blocks on different predicates. For example, consider an album y with three tracks: x_1, x_2, and x_3. Suppose two users independently issue a request to remove x_1 from y, with each operation being processed by a different replica. In replica 1, the operation may block the predicate "track x_2 is in album y", while in replica 2, it may block the predicate "track x_3 is in album y". Although the operation is identical, a non-deterministic policy may issue different blocks even when replicas start in the same state. However, this behaviour does not compromise the correctness of the mechanism. All blocks are uniformly processed during the effect phase, ensuring consistent handling across replicas. As previously explained, when integrating an operation into the graph, the operation is compared against all its concurrent operations, including those already marked as No-Ops. Consequently, after processing the same set of operations, all replicas will converge to the same set of No-Ops, as formally proven in Lemma 1.