# The Wundsam-Shenker Algorithm$^{\text{TM}}$

The original MCS algorithm assumed that causal dependencies were known *a priori*. The revised algorithm infers them programatically.

## Revised Algorithm

Our algorithm performs three steps for each input event: (i) prune the input event, (ii) iteratively infer the causal dependencies for the remaining input events, and (iii) check whether the correctness violation has occurred. If the correctness violation *did* occur at the end, we reinsert the input event, and the causal dependencies are the same as the last iteration.

We infer causal dependencies by (a) starting from the first input event, (b) playing the execution forward for a fixed time window and recording the internal events that occur, (c) replaying from the first input event again, this time injecting the next input event when all of the internal events found in step (b) that were its causal dependencies in the original execution have occurred, and (d) going to step (a) again for the next input event until we reach the end of the log.

Detailed pseudocode for the algorithm follows. The algorithm takes as input a totally-ordered log of the (uniquely labeled) input events and internal events from the production run of the system. The algorithm maintains an array of 'immediate predecessors', where the $ith$ element of the array contains the label of the internal event that immediately precedes the $ith$ input event. The array is initialized to the immediate predecessors from the original (unpruned) log.

```
def findMCS(log):
  ips = [] // 'immediate predecessors'
  foreach input in log:
    ips[input] = internal event immediately preceding input

  foreach input in log:
    new_ips = [] // immediate predecessors after pruning input
    new_ips[0...input] = ips[0...input] // prefix is the same
    new_ips[input] = null // prune (null means 'do not inject')
    foreach successor_input in (input+1)..n:
      internal_predecessor = peek(successor_input, new_ips, log)
      new_ips[successor_input] = internal_predecessor

    violation = replay_and_check_correspondence(new_ips)
    if violation:
      // this input was not a necessary condition
      ips = new_ips // internal events may have changed

    // else we don't prune, and the causal dependencies are the same as
    // last iteration

  return all non-null indices in ips
```

The peek algorithm takes the label of the next input event to be injected, the prefix of immediate predecessors leading up to the next input event, and the complete happens-before constraints from the original execution, and returns the label of the internal event that immediately precedes the next input event:

```
def peek(successor_input, ips, log):
  for input in 0..(ips.length):
    if ips[input] is null:
      // this input was pruned
      continue
    else:
      play forward until ips[input]
      inject input

  internal_events = play the execution forward for a fixed time window
  fingerprint(internal_events)
  return max(internal_events)
```

The fingerprint algorithm is the least clear in my mind. At a high level, it should match up internal events from the pruned execution with internal events from the original execution. If an internal event appears that did not occur in the original execution, it is assigned a null label, otherwise it is assigned the same label as the original internal event. The details of how this matching is implemented need to be fleshed out.

## Complexity analysis

We express the runtime complexity of `findMCS` in terms of the number of input events replayed. We begin by observing that each iteration of `peek(i,..)` replays `i` events. For each input event `i`, we invoke `peek(i,..)`, `peek(i+1,..)`, `...`, `peek(N,..)`, for a total of $(N+1) \times \frac{N-i}{2}$ (Gauss' summation) replayed input events. There are $N$ input events, for a total of $\Sigma_{i=0}^{N-1}(N+1) \times \frac{N-i}{2} = N^3 + N^2 - \frac{N^3-N}{2} \in O(N^3)$ replayed input events.

### Snapshotting

It's possible to reduce our algorithm to $O(N^2)$ by trading off space for computation: if we store a snapshot of the system's state after each invocation of `peak(i,..)`, then the runtime of `peak(i,...)` becomes $O(1)$, since we don't have to replay all events leading up to `i`. Some controllers use many gigabytes of memory though, so it's not clear whether it's tractable to store $O(N)$ snapshots.