



Rebuilding Guard/OR Likelihood with Scenario-Based Conditioning

Overview of the Problem

In the current analytic likelihood implementation, **guarded outcomes and OR (first-of) expressions are not properly conditioned on event order**. This leads to overcounting of probability mass, especially in cases where branches share events or one branch inhibits another. For example, if outcome C fires via `inhibit(reference, by = b2)`, the current integrand multiplies the reference's density by competitors' survival probabilities that **ignore the constraint** that `b2` did not finish first. This double-counts scenarios where competitor outcomes would have required `b2` to finish earlier, inflating the probability of the guarded outcome.

Goal: We need to rewrite the likelihood calculation for guards and OR expressions to enumerate **all valid event-order scenarios**. Each scenario specifies which events finished before time t, which event finished *at* time t (triggering the outcome), and which events remained unfinished by t. By conditioning on these scenarios, we ensure outcome probabilities remain normalized and no double-counting occurs.

Scenario-Based Solution Outline

Key Idea: For any composite expression (guards, OR, AND, k-of-n pools), we derive its density as a **mixture of scenario-specific contributions**. Each scenario corresponds to a particular ordering of the leaf events relative to time `t`. We propagate these conditions through nested pools and guard expressions so that every term in the likelihood is evaluated under exact event completion/survival conditions.

What a Scenario Describes: - **Which event(s) finish exactly at time t:** The "last" required event(s) that trigger the outcome at t. - **Forced Complete set:** Leaf events that *must* have finished *by* time t (possibly earlier or at t). - **Forced Survive set:** Leaf events that *must not* have finished by t (they survive beyond t). - These sets can be partitioned into those relevant to the outcome branch itself vs. those relevant to competitor branches, but ultimately we'll combine them.

Using these, **scenario weight** =

$$\left(\prod_{e \in \text{forced survive}} S_e(t) \right) \times \left(\prod_{f \in \text{forced complete (before t)}} F_f(t) \right) \times \left(\prod_{g \in \text{exactly at t}} f_g(t) \right),$$

where S_e , F_f , f_g are survival, CDF, and density for events `e`, `f`, `g` respectively. This gives the joint probability density of that exact configuration at time t.

Guard Expressions: For `inhibit(A, by = B, unless = P...)`, we will create scenarios where outcome `A` occurs at t under the conditions: - A's required events complete by t (with one finishing at t). - The

blocker B did **not** finish before t (either B finishes after t, or if it finished earlier, a protector P finished even earlier to nullify B's effect). - These conditions will be enforced via forced survive/complete sets for B (and P if applicable).

OR Expressions: For `first_of(expr1, expr2, ...)`, scenarios cover each branch `expr_i` finishing first at t: - All events needed for `expr_i` are done by t (with one finishing at t). - Every other branch `expr_j` is **incomplete by t** (at least one required event of each `expr_j` did not finish by t). - If branches share events, this is handled naturally by forced sets (an event finished by t for branch i might also partially fulfill branch j, so to keep branch j incomplete we force some other event of j to survive).

Pools (k-of-n): We need to extend pool density/survival calculations to yield scenario mixtures instead of a single probability: - The pool can finish at t in many ways: e.g. which k-th event finished at t and which specific k-1 events finished earlier. Each such combination is a scenario. - Under forced conditions (some members forced complete or survive), the pool's scenario set will adjust (e.g. if one member is forced to survive beyond t, it cannot contribute to finishing events). - This means rewriting `.pool_density` and `.pool_survival` to **enumerate combinations of member completions** rather than using the closed-form formula blindly.

By propagating these scenarios upward, a guard or OR expression will **sum over scenarios** from its sub-expressions: - Each scenario enforces certain events' statuses, which then constrain the rest of the expression and competitors. - The outcome density becomes the sum of scenario weights, and the outcome probability is its integral (which will now correctly sum to 1 across outcomes).

Implementing the Scenario-Based Likelihood

Below we outline changes to the code, focusing on key functions. We provide a patch-like diff for clarity, showing removals (-) and additions (+). We'll address the guard case first (which is the most complex), then OR expressions and pools. Finally, we discuss caching of expensive calculations.

1. Refactor Pool Likelihood to Generate Scenarios

The current `.pool_density` computes the PDF of the k-th order statistic via a formula. We will change it to **yield scenario contributions**. Instead of returning a single density, it will return a list of scenario records (each with a weight and forced sets). Higher-level functions (guard/OR) can then combine these scenarios.

Key changes in `pool_density`: generate all combinations where the k-th finish happens at t: - Choose which member finishes *exactly* at t (as the k-th finisher). - Choose which k-1 other members finished before t (any time < t). - All remaining members survive beyond t. - Compute weight = $f(\text{chosen_at_t}) \times \prod(\text{CDF of k-1 finished members}) \times \prod(\text{Survival of others})$.

We introduce a helper (pseudo-code for clarity):

```
.pool_density_scenarios(prep, members, k, component, t, forced_complete=set(),
forced_survive=set())
```

This returns a list of scenarios, each scenario might be a list:

```
list(weight = numeric,
     forced_complete = c(...),
     forced_survive = c(...))
```

Where `forced_complete` and `forced_survive` are sets of **leaf accumulator IDs**.

We must **respect any incoming forced sets** when generating scenarios: - If a member is in forced_survive, it cannot be one of the finished ones. - If in forced_complete, it must be among the finished (and possibly could be the one at t if not already finished earlier). - If forced_complete has more members than k, that scenario has probability 0 (no valid combination). - If forced_survive makes it impossible to get k done by t, probability is 0.

After enumerating, `.pool_density` can sum scenario weights if a plain density is needed, but for internal use we'll propagate scenarios upward.

Similarly, `.pool_survival` will yield scenarios for " $< k$ finished by t " (or we can derive survival from summing scenario weights for 0 up to $k-1$ finishes).

For brevity, below is a **patch for** `.pool_density` to illustrate integrating scenario logic (full enumeration code omitted for brevity):

```
@@ .pool_density <- function(prep, pool_id, component, t) {
- # ...existing code...
- # General k-of-n order statistic density
- #  $f_{\{k\}}(t) = \sum_i f_i(t) * P(\text{exactly } k-1 \text{ of others have finished by } t)$ 
- total_density <- 0.0
- for (i in seq_along(active_members)) {
-   mid <- active_members[[i]]
-   dens_i <- (if (mid %in% forced_complete) .acc_density(...) else .pool_density(...))
-   if (dens_i == 0) next
-   # Probability exactly  $k-1$  of others finished by  $t$ :
-   # use pool_coeffs on others' survival/CDF
-   ...
-   total_density <- total_density + dens_i * prob_k_minus_1
- }
- return(total_density)
+ # Enumerate scenarios for k-of-n completion at t.
+ scenarios <- list()
+ # Determine which members could finish exactly at t
+ for (mid in active_members) {
+   # If forced_survive contains mid, it cannot finish at t
+   if (mid %in% forced_survive) next
+   # Compute density of mid at t (0 if not applicable)
```

```

+   dens_mid <- if (!is.null(acc_defs[[mid]])) .acc_density(acc_defs[[mid]], t)
+           else .pool_density(prep, mid, component, t) # for nested pool
+   if (dens_mid <= 0) next
+   # Determine how many others need to finish before t: k-1 if mid finishes
last
+   m_finished_needed <- k - 1
+   # Choose m_finished_needed members from others that must finish by t
+   others <- setdiff(active_members, mid)
+   # If any chosen set would conflict with forced_survive/complete, skip it
+   for (combo in combn(others, m_finished_needed, simplify=FALSE)) {
+     if (any(combo %in% forced_survive)) next # cannot require an event
forced to survive
+     if (!all(forced_complete %in% c(combo, mid))) next # all forced complete
must be done by t
+     # Calculate probability all in combo finished by t and all remaining
(others-combo) survive past t
+     prob_combo <- 1.0
+     for (mj in others) {
+       if (mj %in% combo || mj == mid) {
+         # finished by t
+         prob_combo <- prob_combo * (if (!is.null(acc_defs[[mj]]))
+                                         (1 - .acc_survival(acc_defs[[mj]], t))
+                                         else (1 - .pool_survival(prep, mj,
component, t)))
+       } else {
+         # survive past t
+         prob_combo <- prob_combo * (if (!is.null(acc_defs[[mj]]))
+                                         .acc_survival(acc_defs[[mj]], t)
+                                         else .pool_survival(prep, mj, component,
t))
+       }
+     }
+     if (prob_combo <= 0) next
+     scenarios[[length(scenarios)+1]] <- list(
+       weight = dens_mid * prob_combo,
+       forced_complete = c(combo, mid),
+       forced_survive = setdiff(others, combo)
+     )
+   }
+ }
+ # Sum of scenario weights (for pure density value, if needed)
+ total_density <- sum(vapply(scenarios, `[[`, numeric(1), "weight"))
+ attr(total_density, "scenarios") <- scenarios
+ return(total_density)
}

```

In this patch, we:

- Loop over each member as a candidate for finishing at t (`mid`). - Skip if it's forced to survive (cannot finish by t).
- Compute its density at t (`dens_mid`). - Choose $k-1$ other members to be finished before t (combinations of `others` taken $k-1$ at a time).
- Check **forced set consistency**: skip combinations that conflict with `forced_survive` or don't include all `forced_complete`.
- Compute probability that exactly those $k-1$ others are finished by t and the rest survive (using each member's CDF or survival).
- Record the scenario with weight and the implied forced sets (`mid` and `combo` are completed by t ; others not in `combo` survive past t).

We attach the scenarios to `total_density` (using an attribute or by returning scenarios directly, depending on design). The survival function `.pool_survival` can be handled similarly by summing scenarios where fewer than k finished (or generating scenarios for $0, 1, \dots, k-1$ finishes).

2. Rewriting Guard Likelihood via Scenarios

Currently, guard density is computed as

$$f_{\text{ref}}(t) \times S_{\text{eff}, \text{blocker}}(t) [21 \dagger L1 - L3] ,$$

where $S_{\text{eff}, \text{blocker}}(t)$ is the probability the blocker didn't successfully inhibit the reference by time t . This is correct marginally, but we need to **condition competitor outcomes on the same event-order**.

We will replace the guard calculation with explicit scenarios:

- Scenario G1:** Blocker did *not* finish by t (it's in `forced_survive`), reference finishes at t (reference's leaf events done by t).
- Scenario G2:** Blocker finished earlier than t *but* a protector finished even earlier, so blocker's inhibition was nullified. In this scenario, reference at t is still allowed. (If no protectors, this scenario has probability 0 because blocker finishing would have inhibited reference.)
- We incorporate protectors by including their event order: e.g. if a protector finished before blocker, that protector is `forced_complete` (finished before t), blocker might or might not finish by t but if it did, it was after the protector.

However, to keep it tractable, we can fold scenario G2 into G1 by treating "**blocker didn't inhibit**" as a single condition. The existing integration for `S_eff` effectively did this. We emulate that by scenario enumeration:

- If protectors exist, one protector finishing before blocker is required in scenarios where blocker finishes before t but doesn't inhibit. This becomes a bit complex to enumerate directly, so we can still use `S_eff` for computing the probability weight, but now we'll attach scenario conditions to it.

Approach: Use scenario G1 as the primary path (blocker incomplete by t), and treat any protector cases as part of that survival probability. In scenario terms:

- Blocker is forced to survive until t (did not finish before t **or** finished after a protector, which from reference's viewpoint is equivalent to "did not cause inhibition by t ").
- If protectors list is non-empty, then forcing "no inhibition by t " means either blocker incomplete or (blocker finished but at least one protector finished earlier). We can't easily enumerate that without time ordering between protectors and blocker; instead, we can compute the weight via the `S_eff` integration but attach the scenario's forced sets accordingly.

Code changes in `.eval_expr_likelihood` for guard:

Instead of directly returning `dens_ref * S_eff`, we gather scenarios and sum their weights. We remove the competitor filtering hacks (the `.expr_requires_source` and `.guards_conflict` filters) since we'll handle it systematically via forced sets.

```

@@ .eval_expr_likelihood <- function(expr, t, prep, component) {
  if (identical(kind, "guard")) {
    - # GUARD: reference fires only if blocker doesn't fire first
    - # Density: f_ref(t) * S_effective_blocker(t)
    - reference <- expr[['reference']]
    - blocker <- expr[['blocker']]
    - unless_list <- expr[['unless']] %||% list()
    - dens_ref <- .eval_expr_likelihood(reference, t, prep, component)
    - if (dens_ref == 0) return(0.0)
    - # Compute effective blocker survival S_eff:
    - if (length(unless_list) == 0) {
      - S_eff <- .eval_expr_survival(blocker, t, prep, component)
    - } else {
      - f_blocker <- function(u) { .eval_expr_likelihood(blocker, u, prep,
component) }
      - S_prot_prod <- function(u) { prod(vapply(unless_list,
function(p) .eval_expr_survival(p, u, prep, component), numeric(1))) }
      - S_eff <- 1.0
      - if (is.finite(t) && t > 0) {
        - S_eff <- tryCatch({
          - 1.0 - stats:::integrate(function(u) f_blocker(u) * S_prot_prod(u),
          - lower=0, upper=t, rel.tol=.integrate_rel_tol(),
          abs.tol=.integrate_abs_tol(),
          - stop.on.error=FALSE)$value
        - }, error=function(e) 1.0)
      - }
      - S_eff <- max(0.0, min(1.0, S_eff))
    - }
    - return(dens_ref * S_eff)
  + # GUARD: reference fires at t under condition blocker didn't inhibit by t.
  + # Enumerate guard scenarios:
  + reference <- expr[['reference']];
  + blocker <- expr[['blocker']]
  + unless_list <- expr[['unless']] %||% list()
  + scenarios <- list()
  + # Get scenarios for reference itself finishing at t (this will handle any
internal structure of reference, including pools).
  + ref_scenarios <- list()
  + if (!is.null(prep[["pools"]][[reference[["source"]]]])) {
  +   # If reference is a pool or composite, get its scenarios attribute by
computing density
  +   ref_density <- .eval_expr_likelihood(reference, t, prep, component)
  +   ref_scenarios <- attr(ref_density, "scenarios") %||% list()
  + } else {
  +   # Reference is a single event (accumulator)
  +   dens_ref <- .eval_expr_likelihood(reference, t, prep, component)
  +   if (dens_ref > 0) {
  +     ref_scenarios <- list(list(

```

```

+      weight = dens_ref,
+      forced_complete = c(reference[['source']]),
+      forced_survive = character(0)
+    )))
+  }
+
+  # For each reference scenario, incorporate guard conditions:
+  for (ref_sc in ref_scenarios) {
+    # Base weight from reference finishing at t under ref_sc conditions
+    base_w <- ref_sc$weight
+    forced_c <- ref_sc$forced_complete; forced_s <- ref_sc$forced_survive
+    # Blocker condition: blocker did not cause inhibition by t
+    # -> If no protectors: blocker didn't finish by t (forced survive)
+    # -> If protectors: either blocker survived or a protector finished
first. We treat this as blocker effectively surviving until t for conditioning.
+    forced_s_blk <- forced_s; forced_c_blk <- forced_c
+    if (!is.null(blocker) && .expr_is_event(blocker)) {
+      blk_id <- blocker[['source']]
+      # If blocker is not already in forced_complete, enforce it to survive
(not finish by t) for reference to occur.
+      if (!(blk_id %in% forced_c)) {
+        forced_s_blk <- union(forced_s_blk, blk_id)
+      }
+      # Protectors: ensure at least one protector finished before blocker if
blocker finished.
+      # We handle protectors by incorporating their survival into weight (via
integration as before), since explicit scenario ordering is complex.
+      prot_factor <- 1.0
+      if (length(unless_list) > 0) {
+        # Compute probability blocker didn't inhibit (same as S_eff used
previously)
+        prot_factor <- (function(){
+          if (!is.finite(t) || t <= 0) return(1.0)
+          # Integrate f_blocker * Prod(S_protectors) from 0 to t
+          val <- tryCatch(stats::integrate(
+            function(u) .eval_expr_likelihood(blocker, u, prep, component) *
prod(vapply(unless_list,
function(p) .eval_expr_survival(p, u, prep, component), numeric(1))),
+            0, t, rel.tol=.integrate_rel_tol(), abs.tol=.integrate_abs_tol(),
stop.on.error=FALSE)$value,
+            error=function(e) 0.0)
+            return(max(0.0, 1.0 - as.numeric(val)))
+          })()
+        }
+        if (prot_factor <= 0) next # no probability of this scenario
+        # Scenario weight = base weight * prot_factor * blocker-survival factor
(if any)

```

```

+      # (blocker survival is already implied by forcing blocker in
forced_s_blk)
+      scenarios[[length(scenarios)+1]] <- list(
+        weight = base_w * prot_factor,
+        forced_complete = forced_c_blk,
+        forced_survive = forced_s_blk
+      )
+    }
+    # Sum scenario weights as total density
+    total_dens <- sum(vapply(scenarios, `[[`, numeric(1), "weight"))
+    attr(total_dens, "scenarios") <- scenarios
+    return(total_dens)
}
}

```

Explanation: We first obtain **scenarios for the reference expression finishing at t** (`ref_scenarios`). For a simple accumulator reference, that's just one scenario with weight `f_ref(t)` and that event in `forced_complete`. If the reference is a pool or complex sub-expression, we rely on its own `.eval_expr_likelihood` to return scenarios (we assume `.eval_expr_likelihood` is modified to attach a `"scenarios"` attribute when it internally enumerates scenarios, as done in `pool_density` above).

Then for each reference scenario, we apply the guard's blocker condition: - We ensure the blocker's source ID is in the forced survive set (blocker did not finish by t) **unless** the reference scenario already had blocker in `forced_complete` (which would be odd unless the reference itself required the blocker, in which case the guard is irrelevant). - We multiply by a `prot_factor` which accounts for protectors: this is essentially the old `S_eff` integration that reduces the scenario weight if there's a chance the blocker actually finished earlier with no protector (which would have inhibited reference). `prot_factor = P(\text{blocker didn't inhibit by } t)`. If no protectors, `prot_factor = S_blocker(t)`. If protectors, we integrate as before. - This part could be further refined into explicit sub-scenarios (protector finished first vs blocker survived), but using `prot_factor` keeps things simpler and the scenario still effectively represents "reference at t & guard not triggered".

Finally, we collect all guard scenarios with their weights and forced sets. We return the sum of weights as `total_dens` and attach the scenarios list to it.

Removal of Competitor Filtering: By constructing these scenarios with blocker in `forced_survive`, any competitor expression that **requires** the blocker's completion will automatically be handled (its survival probability given blocker's forced survival will be 1 at time t). We will use these forced sets in competitor survival calculation next, instead of the old `.guards_conflict` heuristics.

3. OR (first_of) Expressions with Shared Events

OR expressions are essentially "min" (first finish) events. The current implementation assumes each branch `i` finishes first with density

$$f_i(t) \prod_{j \neq i} S_j(t) \quad [21 \dagger L1 - L3] ,$$

which breaks down if branches share events (because $S_j(t)$ is computed without conditioning on branch i's events).

We fix this by scenario enumeration: - For each branch i , for each scenario of branch i finishing at t (similar to guard reference scenarios above), enforce that every other branch j is incomplete by t . - "Branch j incomplete by t " means **at least one required leaf event of branch j is in the forced survive set** for this scenario.

Implementation: 1. Get scenario list for each OR branch's expression finishing at t . 2. For each scenario of branch i , add competitor conditions: - For each other branch j , ensure that not all of j 's required events are in the completed set. Equivalently, find at least one event of branch j to mark as forced survive. - If a competitor branch j shares all its events with branch i 's completed events (meaning branch j would also be done), then the scenario corresponds to a **tie** (both branches complete at t). We could either exclude it (if ties have zero probability) or handle tie-breaking by splitting probability. For continuous independent times, tie probability is ~ 0 , so excluding is fine for normalization. (If ties are to be handled, one could include a tie scenario and allocate half to each, or use a symmetric allocation as in the `shared_gate_pair` logic.) 3. Only include scenarios where each competitor j has at least one event forced to survive (ensuring j is incomplete). Multiply scenario weight by 1 (since we're conditioning — those survival probabilities were already factored into branch i 's scenario weight).

Code sketch for OR in `.eval_expr_likelihood`:

```

@@ if (identical(kind, "or")) {
-   # OR: first to finish
-   total_density <- 0.0
-   for (i in seq_along(args)) {
-     dens_i <- .eval_expr_likelihood(args[[i]], t, prep, component)
-     if (dens_i == 0) next
-     # Survival of all others at t
-     prod_surv <- 1.0
-     for (j in seq_along(args)) {
-       if (i == j) next
-       surv_j <- .eval_expr_survival(args[[j]], t, prep, component)
-       prod_surv <- prod_surv * surv_j
-     }
-     total_density <- total_density + dens_i * prod_surv
-   }
-   return(total_density)
+   # OR: first-to-finish (min). Sum over scenarios where each branch finishes
at t and others are incomplete.
+   scenarios <- list()
+   # Precompute scenarios for each branch finishing at t
+   branch_scen_lists <- vector("list", length(args))

```

```

+   for (i in seq_along(args)) {
+     branch_dens <- .eval_expr_likelihood(args[[i]], t, prep, component)
+     branch_scen_lists[[i]] <- attr(branch_dens, "scenarios") %||% list(
+       # If no scenarios attribute, treat as single scenario if density > 0
+       if (branch_dens > 0) list(weight=branch_dens,
+                                   forced_complete=.expr_sources(args[[i]],
+                                   prep),
+                                   forced_survive=character(0)) else NULL
+     )
+   }
+   # Enumerate OR scenarios
+   for (i in seq_along(args)) {
+     for (sc in branch_scen_lists[[i]] %||% list()) {
+       if (is.null(sc) || sc$weight <= 0) next
+       # Start with branch i scenario conditions
+       forced_c <- sc$forced_complete; forced_s <- sc$forced_survive
+       # Impose that every other branch j is incomplete by t
+       valid <- TRUE
+       for (j in seq_along(args)) {
+         if (j == i) next
+         # Branch j is incomplete if at least one of its required sources is
+         not finished by t
+         # Find one such source not in forced_complete set
+         req_sources_j <- .expr_sources(args[[j]], prep)
+         # If branch j shares all sources with branch i and they are all
+         complete, that's a tie scenario
+         if (all(req_sources_j %in% forced_c)) {
+           valid <- FALSE
+           break
+         }
+         # Ensure at least one required event of j is marked to survive
+         # If any req source of j not in forced_c or already in forced_s, that
+         branch is incomplete.
+         # Mark one of them as forced survive (for explicit conditioning) if
+         not already.
+         found_one <- FALSE
+         for (src in req_sources_j) {
+           if (!(src %in% forced_c)) {
+             # src not finished by t, ensure it's in forced_s
+             forced_s <- union(forced_s, src)
+             found_one <- TRUE
+             break
+           }
+         }
+         if (!found_one) {
+           valid <- FALSE
+           break
+         }
+       }
+     }
+   }

```

```

+      }
+      if (!valid) next
+      scenarios[[length(scenarios)+1]] <- list(
+        weight = sc$weight, # competitor survival already accounted via
forced sets
+        forced_complete = forced_c,
+        forced_survive = forced_s
+      )
+    }
+  }
+  total_density <- sum(vapply(scenarios, `[[`, numeric(1), "weight")))
+  attr(total_density, "scenarios") <- scenarios
+  return(total_density)
}

```

Here we:

- Compute each branch's scenario list (again using the "scenarios" attribute from the branch's density calculation if available, otherwise default to one scenario).
- For each branch-i scenario, we enforce all other branches j are incomplete by ensuring at least one required source of j is not in the completed set. We mark such a source as forced survive in the scenario's forced set.
- If a competitor branch j's all required sources are already in forced_complete (meaning branch j would also be complete by t under the same conditions), we detect a **tie** and mark the scenario invalid (skip or handle separately). In continuous models, this has probability ~0, so skipping is acceptable for normalization.
- We collect all valid scenarios and sum their weights. The weight for branch-i scenario was already $f_i * \dots$ including probabilities of its internal events; we don't multiply by anything else because the condition "others incomplete" has been enforced by removing any scenario where others would have been complete. (If we wanted to be explicit, we could multiply by each competitor's conditional survival given the forced survive events, but since those survive events were not in forced_complete, the original branch-i scenario weight already implicitly included those factors.)

4. Propagating Scenario Conditioning to Outcome Probability

With expressions now producing scenario mixtures, the outcome probability integrals can be updated to utilize these scenarios for competitor survival calculations. The competitor survival term in the outcome probability integrand should consider forced events from each scenario.

Previous approach:

```
surv_comp <- prod(.eval_expr_survival(comp_expr, t) for each competitor)
```

This multiplied unconditional survival probabilities.

New approach: For each scenario of the outcome expression at time t, we multiply by the survival probability of competitors **conditional on that scenario's forced sets**. Since forced sets fix certain events as completed or not, competitor survival simplifies:

- If a competitor's outcome expression requires any event in scenario's forced_survive set, that competitor *definitely* did not finish by t (survival = 1 for that branch under this scenario).
- If a competitor requires events all in forced_complete, that scenario would

have been invalid (we excluded it already to avoid ties). - Otherwise, for events not constrained by the scenario, we still need their survival probabilities.

However, given how we constructed the scenarios, we likely ensured each competitor has at least one forced-survive event. Thus, under each scenario, **every competitor is automatically incomplete** (that was our condition for a valid scenario). This means we could set competitor survival = 1 for scenario weight. If any competitor had all required events free, the scenario would have been invalid or split as a tie.

So the outcome density at t can be obtained by simply summing scenario weights for the outcome's own expression (which we did in the guard/OR assembly). To double-check normalization, we may verify that summing scenario weights for all outcomes yields the total density of any event finishing at t (which should equal the sum of densities of all leaf accumulators at t, ensuring no loss/gain of mass).

Competitor “donor” outcomes (guess outcomes): These are handled as before by adding donor mass. The donor logic should also be adjusted to use scenario-based densities if applicable (e.g. if a donor outcome has guard expressions, use its scenario-weighted density).

Given the complexity, an easier path is to perform competitor survival multiplication using forced sets explicitly when summing outcome probabilities: - When computing an outcome's probability (integral over t), integrate the **scenario-weighted density directly**, which already accounts for competitor incompleteness. - Or sum probabilities from scenario perspective: for each scenario of each outcome, integrate its weight over t (some scenarios will involve integrating over distributions of involved events).

The first approach (integrating scenario-weighted density) is simpler. We ensure that in `compute_loglik` or `response_probability`, when summing up outcome probabilities, we use the new `.eval_expr_likelihood` which already correctly conditions on competitors via forced sets. This means we might remove all the custom competitor filtering code that was previously in `.outcome_likelihood`.

Patch for outcome likelihood and competitor handling:

```
@@ .outcome_likelihood <- function(outcome_label, rt, prep, component) {
  - # Build competitor exprs for this outcome (exclude alias, special, and
  siblings mapping to same label)
  - comp_labels <- setdiff(names(outcome_defs), outcome_label)
  - comp_labels <- Filter(... complicated checks ..., comp_labels)
  - competitor_expressions <- lapply(comp_labels, function(lbl) outcome_defs[[lbl]]
$expr)
  - # Remove competitors that conflict with guard conditions (old approach)
  - if (!is.null(expr$kind) && expr$kind == "guard") {
    - ... # .expr_requires_source and .guards_conflict filtering
  - }
  + # Competitor expressions are considered implicitly via scenario conditioning.
  No need for explicit filtering here.
  + comp_labels <- setdiff(names(outcome_defs), outcome_label)
  + competitor_expressions <- lapply(comp_labels, function(lbl) outcome_defs[[lbl]]
$expr)
```

```

...
  if (is.na(rt) || !is.finite(rt) || rt < 0) {
-    # Integrate probability
-    res <- tryCatch(stats::integrate(integrand, 0, deadline, ...)$value, ...)
-    return(as.numeric(res))
+    # Compute total outcome probability by integrating the expression's density
(which now accounts for scenarios).
+    # We can integrate .eval_expr_likelihood(outcome_expr) directly, since
competitor survival is built into it.
+    prob <- tryCatch(
+      stats::integrate(function(u) .eval_expr_likelihood(expr, u, prep,
component),
+                      lower=0, upper=deadline, rel.tol=.integrate_rel_tol(),
abs.tol=.integrate_abs_tol(),
+                      stop.on.error=FALSE)$value,
+      error=function(e) 0.0
+    )
+    return(as.numeric(min(1.0, max(0.0, prob))))
} else {
-  # Compute density at rt: dens(outcome_expr)*Π_survival(competitors) + donor
contributions
-  dens_r <- .eval_expr_likelihood(expr, rt, prep, component)
-  surv_comp <- .compute_survival_product(expr, competitor_expressions, prep,
component, rt)
-  base_val <- dens_r * surv_comp
+  # Compute density at rt: The expression density already accounts for
competitor survival via scenarios.
+  base_val <- .eval_expr_likelihood(expr, rt, prep, component)
# Add donor (guess) contributions if any...

```

We simplify the outcome density calculation: `.eval_expr_likelihood(expr, t)` now yields the correctly conditioned density for that outcome at time t (with scenarios). Thus we can use it directly without multiplying by competitor survivals manually. The competitor expressions list can actually be dropped entirely in the density calculation. (If we still need competitor info for guess weighting, we can handle guess separately.)

Note: If any competitor outcomes still need to be considered (like for guess outcomes or mapped outcomes), their integration can similarly use the new scenario-based densities.

5. Caching Expensive Calculations

With the scenario approach, we are performing more integration and combination, which can be computationally heavy. There are several places to cache intermediate results **within a single trial's likelihood computation** to avoid redundant work:

- **Cache event CDF and survival values at time t :** When evaluating many scenarios, the same accumulator's $F(t)$ or $S(t)$ may be computed repeatedly. We can memoize

`.acc_survival(acc, t)` and `.acc_density(acc, t)` for each accumulator `acc` per time `t` encountered. A simple way is to maintain a dictionary (hash) keyed by `(acc_id, t, type)` in the scope of computing `.eval_expr_likelihood` for a given outcome. Because `compute_loglik` already caches entire trial likelihoods by a key, we can extend that cache to finer granularity:

- e.g., inside `.eval_expr_likelihood`, use a static or closure environment `eval_cache` to store results like `eval_cache[[paste0("S:", acc_id, "@", t)]]`.
- This ensures if the same event survival is needed again (in another scenario combination, or another branch's calculation at the same `t`), we reuse it.

• **Cache integrals for `S_eff` (guard blocker integrals) and tie-break allocations:** If the integration for `prot_factor` (blocker vs protectors) is expensive, cache it by key `(blocker_id, protectors_ids, t)` in a local environment so multiple scenarios or repeated calls don't recompute it for the same `t`. Similarly, if tie scenarios were handled via an integral (like in the `shared_gate_pair` logic), cache the partial integrals. In our scenario approach, we largely eliminated repeated tie integrals by avoiding double integration in favor of direct scenario sums, but if any remain (like computing `prot_factor` for multiple scenarios that share the same blocker/protectors), caching helps.

• **Reuse scenario lists:** Computing all scenario combinations for pools and OR branches can be done once per expression per time `t`. We can store the scenario list in the `"scenarios"` attribute (as done) for reuse by parent expressions or competitor calculations. In other words, once `.eval_expr_likelihood` computes and attaches scenarios for a given expression at `(t, component)`, another part of the code asking for the same expression's density at the same `t` can reuse the scenarios instead of recomputing.

• **No cross-trial caching needed:** As noted, each trial likely has different `t` or parameter values, so caching across `compute_loglik` calls is not very beneficial. But within one `integrate(...)` call, the integrand will be called many times at different `t`; we should ensure caching is reset appropriately for each integration to avoid unbounded growth.

Implementation of caching: We can integrate it by adding an optional cache parameter through recursion or using `options()` or an environment in closure: For example, modify `.eval_expr_likelihood` signature (internally) to accept a cache env:

```
.eval_expr_likelihood <- function(expr, t, prep, component, cache = list())
{ ... }
```

and use `cache$key <- value` to store results. Or simpler, define `.acc_density` and `.acc_survival` to use a global small cache:

```
.acc_density <- local({
  cache <- new.env()
  function(acc_def, t) {
    key <- paste(acc_def$id, t, "dens")
```

```

    if (!is.null(cache[[key]])) return(cache[[key]])
    # compute dens
    cache[[key]] <- dens
    dens
  }
})

```

and clear `cache` at appropriate times (e.g., at start of each `compute_loglik` for a trial or each integration over `t` range).

Caching Example: In the guard scenario loop, we call `.eval_expr_survival(p, u)` for each protector in an inner integration. We can precompute the survival curves on a grid or at specific integration nodes. Using `stats::integrate`, we might supply a vectorized integrand; we can cache each evaluation of the integrand by `u` as the integrator steps through. This could be complex, but we can at least cache the product of protector survivals at each `u` to avoid recomputing it multiple times in the same integral evaluation.

6. Testing the New Implementation

After making these changes, we should verify: - **Normalization:** The sum of probabilities of all outcome labels equals ~1 (within numerical tolerance). This confirms no double-counted or missing mass. - **Match with simulation:** For the provided example (and others like `test_4choice.R`), compare analytic probabilities with large-sample simulation proportions for each outcome. They should align closely. - **Performance:** Check that the runtime is reasonable. If performance is an issue, identify hotspots (likely scenario enumeration in large pools or multiple nested guards) and consider further optimizations or pruning of improbable scenarios.

Note: The above code sketches omit some details (like retrieving `acc_def$id`, handling nested protectors scenarios explicitly, etc.) for brevity. But they outline the major structural changes needed: - Use scenario lists in pool, guard, and or computations (with weights and forced sets). - Remove ad-hoc competitor survival adjustments in favor of scenario conditioning. - Use caching within these computations to avoid redundant probability calculations.

With this approach, the analytic likelihood should handle complex guard and OR cases robustly, maintaining proper normalization and aligning with the simulation results for tied or inhibitory race conditions. Each outcome's probability is derived from a sum over mutually exclusive event-order scenarios, ensuring **no double-counting** of probability mass across outcomes.