UM Runner: Mathematical Specification

From CMP Paper to Code

Claude and MJC — February 21, 2026

Every formula below has a CMP paper reference and a code location. All logarithms are base 2 (CMP §Notation).

The UM is universal: the same runner handles any domain (text, images, chess, audio) depending only on how event spaces and P-programs are configured. This spec describes the domain-independent machinery. Domain-specific structure (input encoding, context chains, etc.) belongs to P-programs, not to the runner.

1. The Universal Model

A UM instance is $u = (E, T, P, f, \omega)$ (CMP §1).

Symbol	CMP Reference	Code
$E = \prod E_i$	#event-space	`UM_ES` array in `UM` struct
$t \in T$	#total-thought	`u.support[]` (uint8 array)
$p \in P \subseteq E^2$	#total-pattern	`UM_LPP_S.entries[]`
$f_p : T \to T$	#standard-update	`um_forward()`
ω	#learning-function	`um_lpp_observe()`

2. Support Values: Two Regimes

$t : E \to \{0, 1, \ldots, 255\}$ assigns log-support to each event (CMP #total-thought).

There are **two distinct regimes**, which in principle meet in the middle but in practice are unlikely to:

2.1. Evidentiary regime (bottom-up)

Patterns supported by observation, ranging upward from $s = 1$.

- $s = 0$: no specific evidence. Baseline. NOT "absent" — the event exists in E, it is just unsupported.
- $s = k$ for small k: approximately 2^k observations under ω_0. Rules of thumb: $s = 10 \approx 1{,}000$ observations, $s = 20 \approx 1{,}000{,}000$, $s = 30 \approx 10^9$.
- In practice, evidentiary support rarely exceeds ∼40–50 even for very large datasets. Support 254 would require $\sim 2^{254}$ observations, comparable to the number of elementary particles in the visible universe. The UM does not double-count: it captures the signal the model can represent the first time.

2.2. Abductive regime (top-down)

Patterns introduced by abduction or definition at $s = 255$ (CMP #trust-and-255).

- $s = 255$: definitional truth, sensory input, or abductive inference. The event is taken as given.
- $s = 254$: as in philosophy, there is always the possibility of doubt. A pattern at 255 may be degraded to 254. In this upper regime there may be degrees of certainty.
- The relationship between 255 and 254 is the same as between 5 and 4: the latter has half the support. Under softmax, disregarding other events, the probability ratio is 2:1 regardless of the magnitude.

Within each regime, thoughts compete with one another if they share an event space (via softmax). If both regimes are present in the same ES, the abductive support (at 255) utterly dominates any evidentiary support (∼10–50), making the evidentiary result effectively irrelevant.

Code: `u.support[global_idx]`, type `uint8`.

3. LPPs: Shorthand for the Pattern Space

In the theory, P defines the *connectome*: the full cross-product of any two event spaces, with an atomic pattern between every pair of events. In a fully-connected model, every pair of events in every pair of ESs would have a pattern between them, most with support 0.

An LPP (Learned Probabilistic Pattern) is a **shorthand form** for this full cross-product between two ESs A and B. It stores only the entries with non-zero support; all unstored pairs are implicitly present at support 0. If either ES grows (new events created), the LPP's implicit zero-support entries extend accordingly.

This shorthand is also a **runtime optimization**: iterating over sparse non-zero entries is vastly cheaper than iterating over the full $|A| \times |B|$ cross-product. In the brain, the number of actual synapses is vast enough that the full connectome IS the implementation. In practical computer systems we use the shorthand form.

The log-support of each stored joint event (a, b) is the weight of the atomic pattern between them. This is what ω_0 records and what f_0 reads.

An LPP is fully representable in SN format (§10) and must be, since without LPPs no model can learn.

Code: `UM_LPP_S` struct in `#umr_core`. Each LPP has a source ES, a target ES, and a sparse list of entries `(from, to, w)`.

4. The Standard Update Function f_0

From CMP #standard-update:

$$(f_p(t))_j = \max_i \min(t_i,\, p_{ij}) \tag{1}$$

Conjunction-as-min, disjunction-as-max. For each atomic pattern $(i \to j)$ with weight w:

$$\text{val} = \min(t[i],\, w) \tag{2}$$

$$t'[j] = \max(t'[j],\, \text{val}) \tag{3}$$

This is the only primitive. Multi-event conjunctions (e.g. patterns that depend on two or more source events jointly) are expressed as chains of atomic patterns through intermediate events, applied in the correct layer order. Topological sorting of the ES dependency graph determines the layer order (see *Connectome Layers*, 20260218). The runner applies LPPs in layer order; each layer's outputs become the next layer's inputs.

Code: `um_apply_lpp()` in `#umr_core`.

```
for each entry (from, to, w) in LPP:
    s = support[from]
    if s == 0: continue
    val = min(s, w)
    support[to] = max(support[to], val)
```

Note: The current code also supports a two-source shorthand `(from, from2, to, w)` where `val = min(s[from], s[from2], w)`. This is a runtime convenience equivalent to routing through an intermediate conjunction event; it does not extend f_0.

5. Output Distribution (Softmax on Log-Support)

From CMP #standard-learning-matrix:

$$p(e_j) = \frac{2^{s_j}}{\sum_{j'} 2^{s_{j'}}} \tag{4}$$

where s_j is the support value for event e_j in some ES after the forward pass. Since support values are log base 2 (CMP §Notation: "All logarithms base 2 unless noted"), 2^s recovers the count scale. This applies to any ES, not just an output ES — $P(I \mid O)$ can be read from the same table by conditioning on columns instead of rows (§6).

Critical: $s = 0 \Rightarrow 2^0 = 1$. Every event contributes weight 1 at baseline. An event seen once ($s = 1$) contributes weight 2. This is Laplace smoothing: a distribution $(2, 1, 1, \ldots, 1)$ over $|E_i|$ events for a single observation, giving entropy near $\log_2 |E_i|$ (near-uniform).

Code: `um_output_dist()` in `#umr_core`.

```
for j in 0..size:
    s = support[base + j]
    dist[j] = ldexp(1.0, s)    // 2^s
    sum += dist[j]
for j in 0..size:
    dist[j] /= sum
```

JS: `outputDist()` in `#um_viewer`.

```
dist[j] = Math.pow(2, s)       // 2^s
```

6. The Standard Learning Function ω_0

From CMP #standard-learning-algorithm:

ω_0 is log-stochastic counting of joint events. It does not create events or modify E — it only records observations as pattern weights within existing LPPs.

Given an observation of joint event (a, b) in an LPP between ESs A and B: find the entry (a, b, s). If not found, add with $s = 1$. If found, log-stochastically increment s:

$$s' = \begin{cases} 1 & \text{if } s = 0 \text{ (first observation)} \\ s+1 & \text{with probability } 2^{-s} \\ s & \text{with probability } 1 - 2^{-s} \end{cases} \tag{5}$$

After n observations: $E[s] \approx \log_2 n$ (CMP #standard-learning-intro). No artificial cap is needed: since $s = 254$ would require $\sim 2^{2^{54}}$ observations (more than elementary particles in the visible universe), evidentiary support stays well below this in any conceivable experiment. The probability of a cosmic ray flipping a bit in RAM is vastly higher.

Early exit: The implementation samples bits of entropy one at a time. At support s, we need at most s random bits (all zero = increment). For $s > 63$, the probability $2^{-s} < 10^{-19}$ is negligible and we exit immediately. This is what makes log-stochastic counting scale: the cost per observation is $O(\min(s, 64))$ random bits.

Symmetry: The log contingency table is a sufficient statistic for *both* $P(B \mid A)$ and $P(A \mid B)$ (CMP #omega0-sufficiency). The same LPP can be read from either side: row-wise gives $P(B \mid A)$, column-wise gives $P(A \mid B)$. No learning function can extract more information from the data.

Code: `um_log_incr()` and `um_lpp_observe()` in `#umr_core`.

```
um_log_incr(s):
    if s == 0: return 1
    if s > 63: return s          // 2^{-s} negligible, early exit
    r = random_64_bits()
    if top s bits of r are all 0: return s + 1
    return s

um_lpp_observe(lpp, from, to):
    idx = find(from, to)
    if found: entries[idx].w = um_log_incr(w)
    else: add entry (from, to, w=1)
```

7. Threshold Creation

Threshold creation is an extension to ω (not to ω_0, which only counts). It is the only event-creation mechanism we have so far.

7.1. Mechanism

Given an LPP between ESs A and B, ω_0 records joint events (a, b) with log-stochastic counts. Threshold creation adds a rule: when the support of a joint event (a, b) reaches a threshold τ (e.g. $\tau = 4$, corresponding to $\sim$16 observations), a new event is created in a separate ES representing that joint event.

This is analogous to the LPP shorthand (§3): just as the LPP avoids storing zero-support patterns, threshold creation avoids creating events until there is sufficient evidence. In a fully-connected model, every possible joint event would have a corresponding neuron from the start — most with support 0. Threshold creation is the optimization that avoids this, creating neurons only when warranted. In the brain, this full pre-allocation may be what happens; in a computer, we create on demand.

7.2. What threshold creation does NOT do

- It does not modify ω_0. The counting mechanism is unchanged.
- It does not use a separate hash table. The counts that trigger creation are the LPP's own joint-event log-supports.

7.3. SN representation

Both LPPs and the threshold creation rule are representable in SN format (§10). The LPP is stored as its non-zero entries (the shorthand form). The threshold τ is a parameter of the LPP declaration. A model that cannot represent its learning rules in SN cannot be saved and restored — so these must be part of the format, not external configuration.

Code: `settle_track()` in `#umr_settle`.

7.4. Open problem: working memory

Threshold creation produces new events from joint events that reach sufficient support. But a joint event on two ESs at different time steps has nowhere to live without memory: the source event from the previous step is gone. The necessary structure is a chain of memory ESs that carry information forward in time, with absolute patterns ($w = 255$) implementing the shift. Conjunction events then combine memory positions. This working memory problem — how to give the UM temporal depth — is the subject of `#working_memory`. It is a P-program design problem, not a change to the runner.

8. Support Gap (Sharpness)

The support gap $s_1 - s_2$ between the two highest support values in an output ES measures how *sharp* the model's prediction is. This closes a theoretical hole: the forward pass f_0 produces support values, but multiple LPPs may contribute to the same output ES, and we need a way to assess which LPP's contribution is most reliable.

For an LPP mapping events from ES A to output ES O:

$$s_1 = \max_{j \in O} s_{a,j} \tag{6}$$

$$s_2 = \max_{j \in O \setminus \{j^*\}} s_{a,j} \quad \text{where } j^* = \arg\max s_{a,j} \tag{7}$$

$$\text{gap} = s_1 - s_2 \tag{8}$$

The gap is measurable entirely within the support lattice (no softmax, no normalization). A gap of k means the top prediction has 2^k times the support of the runner-up. It grows with evidence: a context observed once has gap ≤ 1; a well-observed context can have gap 3–4.

The ring construction (CMP #ring-pattern) provides a way to compute this gap via P alone, using a P-program rather than external instrumentation. This has not yet been implemented; the direct gap calculation serves as a stop-gap.

Code: `um_lpp_dist()` in `#umr_settle` computes gap alongside the output distribution.

9. Envelope

Events named "True." are always set to support 255 before each forward pass (`um_set_envelope()`). These are the basic assumptions under which the UM operates — the organism's viable envelope.

Within a running UM, True is always true. If this UM is nested inside a larger one, the outer UM determines whether the inner UM is *activated at all* — analogous to closing one's eyes (skipping visual processing) or not speaking (skipping speech production). When the outer UM does not activate the inner one, the inner UM's forward pass is simply not run. This may be related to context events, in a way that has not yet been worked out.

Code: `um_add_always_on()` / `um_set_envelope()` in `#umr_core`.

10. SN Format

SN (Stochastic Network) is the interchange format for a complete UM. A model is saved as a single `.sn` text file and loaded by both the C runner and the JS viewer. The format must represent everything needed to resume operation: event spaces, their events with current support, LPPs with their learned weights, and learning parameters.

10.1. Grammar

```
; comment (ignored)
DATA "source_description" observations_seen.
```

```
ES "name" size.
  "event_name" support.       ; repeated, one per event (line order = index)
LPP "from_es" "to_es".                  ; active (default)
LPP "from_es" "to_es" OBSERVE.          ; learns but excluded from f
LPP "from_es" "to_es" FROZEN.           ; participates in f but no learning
LPP "from_es" "to_es" THRESHOLD 4 "target_es".  ; with threshold creation
  from_global to_global weight.         ; pattern entry
  from1 from2 to_global weight.         ; conjunction shorthand
```

- `DATA` records what the model has been trained on.
- `ES` declares an event space. Events follow, one per line in index order. Global event index = ES base + local index. ESs are numbered by declaration order.
- `LPP` declares a learned pattern between two ESs (by name). Entries follow as global-index tuples with log-support weights. Unstored pairs are implicitly at support 0.
- `LPP` modes: **active** (default, no annotation) participates in both f and ω_0. **OBSERVE** participates in ω_0 only — a passive instrument that measures joint events without affecting inference. **FROZEN** participates in f only — for baked-in weights from a memory trace or prior model. Code: `lpp->mode` enum (`LPP_ACTIVE`, `LPP_OBSERVE`, `LPP_FROZEN`).
- Events named `"True."` are detected on load and registered as always-on (envelope, §9).

10.2. Round-trip invariant

Save → load → run must produce identical results to continuing from the live model. This requires that SN captures the full model state: all support values, all LPP weights, and enough metadata to reconstruct the learning configuration (which ESs have LPPs between them, thresholds for creation, etc.).

10.3. Example

A minimal model with one input ES, one output ES, an envelope, and a single LPP:

```
; Minimal UM model
DATA "example" 100.
ES "input" 4.
  "event_a" 0.
  "event_b" 0.
  "event_c" 0.
  "event_d" 0.
ES "output" 3.
  "out_x" 0.
  "out_y" 0.
  "out_z" 0.
ES "envelope" 1.
  "True." 255.
```

```
LPP "input" "output".
  0 3 5.        ; input event 0 -> output event 3, support 5
  1 4 3.        ; input event 1 -> output event 4, support 3
```

Code: `um_save_sn()` and `um_load_sn()` in `#umr_sn_io`. JS: `parseSN()` in `#um_viewer`.