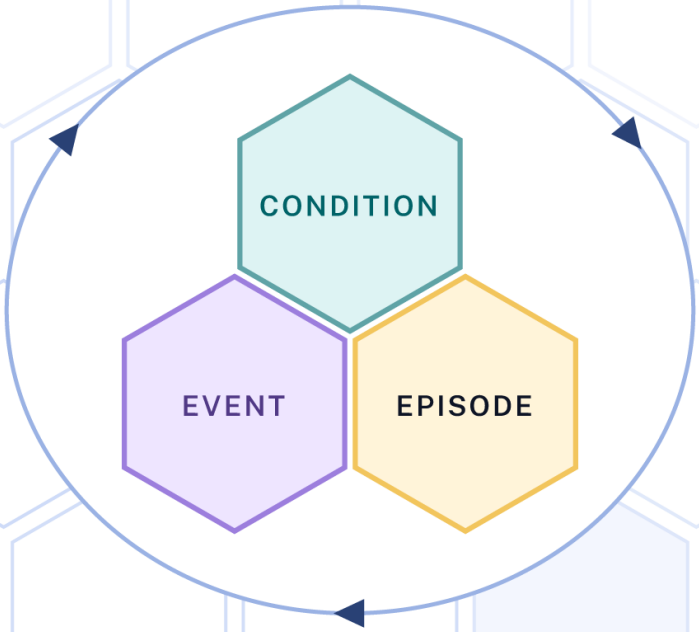
WHITE PAPER



Event, Condition, Episode

A General Ontology for Temporal Reasoning

AUTHOR
Jason Dossett

PUBLISHED
October 2026

Event, Condition, Episode: A General Ontology for Temporal Reasoning

Jason Dossett
SEP
jtdossett@sep.com

Abstract—This paper presents a minimal, implementation-agnostic ontology for reasoning about time-bounded occurrences and the rules that give them meaning. The model separates three primitives — Event, Condition, and Episode — and defines two independent axes on Events, Provenance and Generation, to distinguish where a fact came from from how many derivation hops separate it from observation. A populated four-cell matrix demonstrates that derivation source and business meaning are independent, and the resulting diagnostic exposes common failure modes such as re-deriving source-native business facts or over-governing low-level synthetic plumbing. The paper further reframes Episode boundary detection as a coverage claim: source-native keys, rule toggles, inference-based detectors, and procedural decomposition all map onto existing boundary-provenance or hierarchy concepts without requiring new Episode types. Uncertainty, weighted evidence, and predictive outputs are handled as metadata on Events or Episodes while preserving a boolean materialized Episode. Finally, the model distinguishes structural Hierarchy from computed Temporal Relations between Episodes, adopting Allen's interval algebra for the latter, gives a boundedness argument based on acyclic dependency graphs, defines a mechanical generic/curated boundary, and positions the vocabulary against related work in provenance, complex event processing, active databases, interval algebra, event modeling, and dimensional modeling.

Index Terms—event-driven systems, temporal reasoning, ontology, interval algebra, complex event processing, event sourcing, episode segmentation.

I. INTRODUCTION

This paper describes a small, general-purpose ontology for reasoning about time-bounded occurrences and the rules that give them meaning. It is deliberately **implementation-agnostic** — no framework, decorator syntax, or storage schema is assumed.

The model exists to answer three recurring questions in any system that needs to reason about intervals rather than only point-in-time facts:

1. What is the smallest set of categories that can express "something happened," "here's a rule for classifying what happened," and "here's a bounded period during which that rule held," without collapsing any two of them into one?
2. How does a system avoid an unbounded chain of derived facts triggering more derived facts, forever?
3. Where is the line between a generic, reusable primitive and a business-specific, curated result — and is that line about naming, or about something checkable?

II. THE THREE CATEGORIES

Everything in this model is one of exactly three things (Fig. 1).

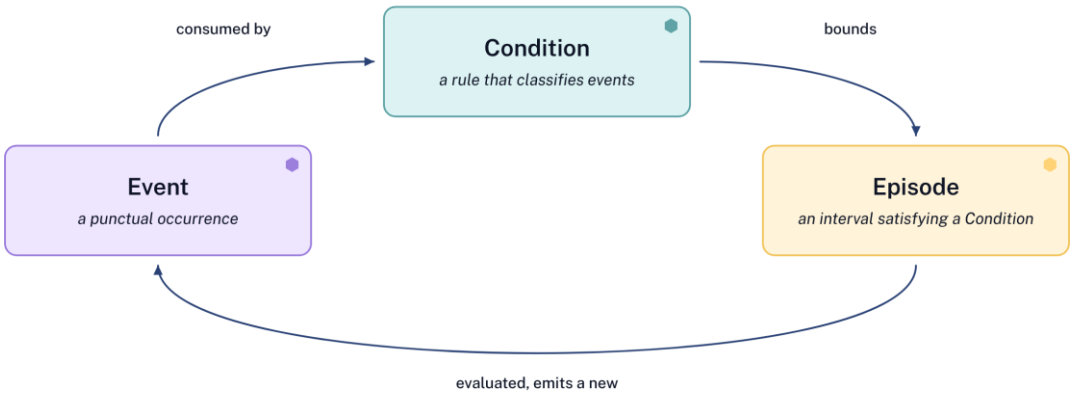


Fig. 1. The three fundamental categories of the ontology and the cycle by which each produces the next.

A. Event. A punctual occurrence — something that happened at a specific instant. An Event is a fact, not an interpretation of a fact.

B. Condition. A rule that classifies a stream of Events. A Condition is the thing that gives an interval meaning — it is the "why," not the "when." Condition content (a threshold, a pattern, a comparison) is business-authored: someone chose it deliberately, using domain judgment, and can reasonably revise it later based on that same judgment. A processing checkpoint or cursor has no equivalent — there is no domain judgment to exercise over what its value should be, because it carries no business content at all. That is the distinction that matters, not how the two happen to be stored.

C. Episode. **Interval** — a bounded span of time, with a start and an end, bounded by two Events — that satisfies a Condition. An Episode is **not** a kind of Event, and it is not merely an Interval either: it is the fusion of an Interval and the Condition that classifies it.

The Interval by itself, with no Condition attached, is a real and useful concept, not something this model discards — it is what Allen's interval algebra operates on (see Section VII), and it is what a native segmentation source (a batch ID's start/end, a changepoint detector's boundary) hands you before any Condition has been attached to it (see Section V). It is not an Episode until a Condition is attached — an Interval is the "when," an Episode is the "when" fused with the "why." (This is the "Descriptions and Situations" pattern from foundational ontology work [1].)

III. TWO INDEPENDENT AXES ON EVENTS

Not every Event is the same kind of fact. Two axes distinguish them, and critically, **neither implies the other**.

A. Provenance: Where Did This Fact Come From?.

- **System-Observed** — emitted natively by some system: a sensor, an API, a file landing process, another platform's own event stream.
- **Synthetic** — computed by this model's own reasoning, not observed directly. Subdivides by *how* it was computed:
  - **Condition-spawned** — the boundary of a Condition becoming or ceasing to be true.
  - **Aggregation-derived** — computed from a windowed aggregate.
  - **Correlation-derived** — computed by joining multiple subjects or streams.
  - **Inference-derived** — computed by a model or scorer rather than a deterministic rule.

B. Generation: How Many Derivation Hops Separate This Event from an Observation?.

$$\textit{generation}(\text{System-Observed event}) = 0 \quad (1)$$

$$\textit{generation}(\text{Synthetic event}) = 1 + \max(\text{generation of every Event consumed by the Condition that produced it}) \quad (2)$$

A System-Observed Event is always generation 0. A Synthetic Event's generation is always at least 1, and strictly greater than everything it was derived from — generation only ever increases along a derivation chain, never resets or loops backward. This is what makes the model's recursion terminate (see Section VIII) without needing to track anything more complex than an integer.

C. Independence of the Two Axes.

The two axes are independent rather than interchangeable: a System-Observed Event can already be business-meaningful (a source system might emit a well-named business fact directly); a Condition-spawned Synthetic Event can still be low-level and generic (`value_changed`). How "derived" something is says nothing about how meaningful it is — treating them as the same axis is a common conflation this model deliberately avoids. The failure modes show why the distinction matters. If a system assumes "business-meaningful" must mean "derived by us," it will miss source-native business facts and may reconstruct boundaries from timestamps even when the source already supplied a cleaner discrete fact such as `batch.started` or `order.placed`. That recreates the boundary ambiguity Section V argues against. Conversely, if a system assumes "synthetic" must mean "business-curated," it will over-govern low-level plumbing: a generic `value_changed` event may acquire review workflow, versioned approval, and domain ownership merely because it was derived, even though it carries no business-authored content. That is the governance inflation Section X is meant to avoid.

TABLE I
INDEPENDENCE OF DERIVATION SOURCE AND BUSINESS MEANING

Derivation source	Generic / low-level	Business-meaningful
Observed: generation 0	A raw sensor tick such as `temperature.reading`. This is the expected low-level observation case.	A source system emits a named business fact directly, such as `batch.started` or `order.placed`. It is un-derived but already meaningful.
Derived: generation ≥ 1	A synthetic plumbing event such as `value_changed` or `temp_up.started`, produced by a threshold crossing or state change. It is derived but still generic.	A synthetic business result such as `predicted_issue` or `excursion.detected`, produced by a rule, aggregate, correlation, or model. It is both derived and business-meaningful.

As a practical audit, if a team cannot populate all four cells for its own system, it has likely conflated derivation source with business meaning.

IV. A WORKED EXAMPLE

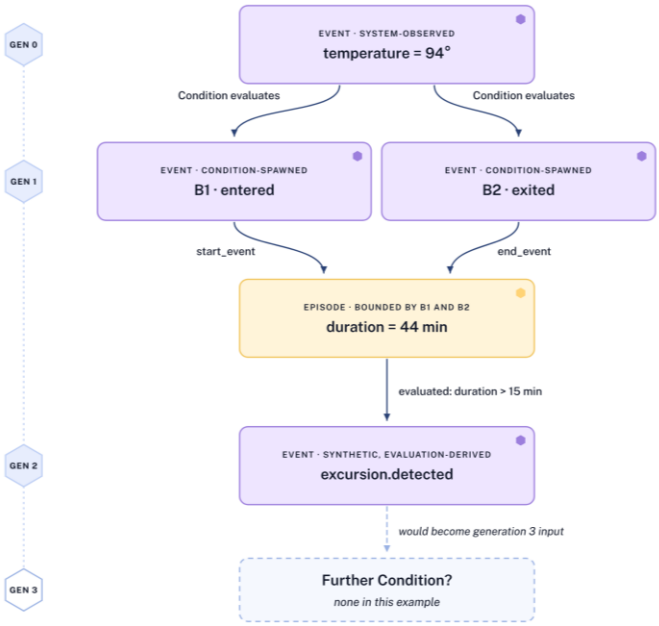


Fig. 2. A three-generation derivation chain from a raw observation to a synthetic, evaluation-derived Event.

The chain stops the moment nothing consumes the highest-generation Event produced so far. Nothing in the model requires it to stop — if a further Condition existed watching for `excursion.detected`, the chain would continue at generation 3, and the same structure (Event → Condition → Episode → Event) would repeat one level up.

V. PROVENANCE OF EPISODE BOUNDARIES

The worked example in Section IV derives both boundary Events from a Condition evaluating raw signal — but that is only one member of a broader coverage claim. The vocabulary does not require a new kind of Episode for each way a timeline gets segmented. In practice, the common segmentation mechanisms reduce to the existing provenance categories for boundary Events, while the Episode itself remains the same structure: an Interval, bounded by Events, fused with the Condition that gives that interval meaning.

1) Source-native discrete segmentation maps to System-Observed boundaries. When a source system already assigns a discrete grouping key, use that key directly instead of reconstructing the Interval from timestamps. A batch ID, run ID, record number, or step number gives exact, unambiguous interval membership — a row either carries `batch_id = X` or it does not. Inferring the same membership from a timestamp range has to get several genuinely hard things right instead: whether a reading exactly at the boundary is in or out, clock skew between whatever system stamped the reading and whatever system defines the interval, out-of-order or late arrival, and rows that happen to share a timestamp at the resolution available. A discrete key sidesteps all of that. In this model, that case is simply a System-Observed boundary pair (`batch.started`/`batch.ended`, generation 0 on both ends), with the Condition doing little more than grouping by the key itself.

The remaining common mechanisms map onto existing concepts as follows:

- **2) Rule toggles map to Condition-spawned boundaries.** A condition that turns true and later false — gaps-and-islands, threshold crossings, consecutive same-state runs — is the Condition-spawned case already described in Section III-A. The boundary Events are synthetic, but the Episode structure does not change.
- **3) Detector-based segmentation maps to inference-derived boundaries.** A continuous signal with no discrete trigger, segmented by a changepoint detector, regime detector, or scorer, produces Inference-derived boundary Events. The boundary comes from a model's output rather than a deterministic expression, which is exactly the distinction that provenance category exists to name.
- **4) Procedural decomposition maps to Episode hierarchy.** A manufacturing batch's stages and steps, or a process reconstructed from an event log, is a case of Hierarchy between Episodes — see Section VII — rather than a new kind of boundary detection. The hierarchy explains how Episodes are structurally nested; it does not change what an Episode is.

The coverage claim is not that every segmentation source is identical, but that each source affects boundary provenance rather than Episode type. Discrete keys supply observed boundaries; rule toggles spawn synthetic boundaries; detectors infer boundaries; process decomposition relates already-formed Episodes into a hierarchy. None of those cases

requires inventing a new fourth category beside Event, Condition, and Episode.

The most challenging case is fuzzy or probabilistic segmentation: a changepoint detector may return a confidence score, a soft boundary, or an "around this time" tolerance rather than a single exact instant. That does not require a new Episode type either, but it does require the implementation to be honest about what was produced. The boundary Event may carry confidence, tolerance, detector version, or lead/lag metadata, and downstream analysis may decide how much uncertainty to accept. The Episode remains boolean at the point it is materialized — this interval either satisfies the attached Condition or it does not — while the uncertainty that led to that decision remains metadata on the boundary Events, the Episode, or both. The same pattern applies to weighted or N-of-M evidence: the Condition may use graded evidence internally, but the materialized Episode remains boolean, with confidence or evidence strength retained as metadata.

Predictive outputs fit the same vocabulary. A forecasted risk or predicted issue is an Inference-derived Event, and any resulting predicted-risk Episode is still an Episode: a bounded interval satisfying a Condition. Confidence, lead time, model version, and later validation status belong as metadata on the Event, the Episode, or both; they do not require a separate predictive category outside the model.

Different detectors can run over the same underlying series and produce different, overlapping sets of Episodes without conflicting, because nothing in this model assumes there is only one valid way to carve up a timeline — the same reasoning applies below to multiple Conditions independently classifying the same interval.

VI. EPISODES VERSUS EPISODE-DETERMINATION EVENTS

An Episode is not itself an Event. An **Episode-determination Event** is a punctual fact *about* an Episode: the system determined that an Episode opened, closed, expired, was superseded, or otherwise changed lifecycle state. That determination Event is a first-class Event and can be consumed by later Conditions like any other Event. It is not a substitute for the materialized Episode row it announces.

Two Events with the same start and end timestamps can still be two entirely different Episodes, if two different Conditions independently classify the same underlying interval. `temperature > 90` and `temperature > 90 AND humidity > 80` can both be true over the exact same window on the same subject — those are two Episodes, not one Episode observed twice, because an Episode's identity includes *which Condition* is doing the classifying, not just the interval's boundaries.

This has a direct, practical consequence for anything implementing this model: an Episode needs a **stable identity independent of its boundaries**, because:

- an Episode can be **open** (no end Event yet) — there is nothing to compute a bare interval from until it closes, but the Episode already exists as a tracked, addressable thing;
- an Episode can be **revised in place** — a provisional close (an expiration, in the absence of a real end Event within an expected window) can later be superseded by a real end Event, and anything that already referenced the Episode needs that reference to remain valid across the correction;
- other things need to **point at** a specific Episode (an evaluation result, a downstream record) — which requires an identity that survives regardless of whether the interval's boundaries are still being computed.

A live, on-demand computation over the raw Events (a join or a view) cannot provide any of these three properties, because it recomputes from nothing on every access and has no persisted identity between accesses. This is the core argument for why any real implementation of this model materializes Episodes rather than treating them as a derived, on-the-fly query.

VII. RELATIONSHIPS BETWEEN EPISODES

Everything in Sections II–VI describes a single Episode's relationship to its own boundary Events. Real processes also need to relate *Episodes to other Episodes* — and there are two structurally different relationships hiding under words like "inside" or "nested," which must not be conflated (Fig. 3).

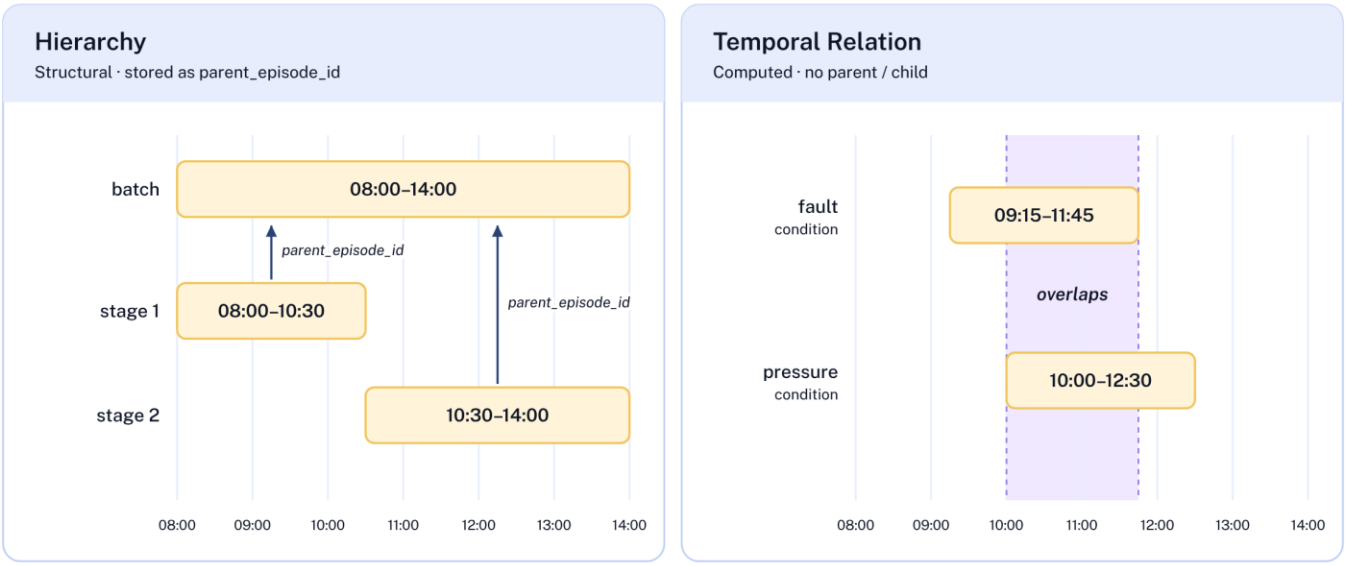


Fig. 3. Structural Hierarchy (left) contrasted with a computed Temporal Relation (right). The two must not be conflated.

A. Hierarchy.

Hierarchy is a **structural** claim: this Episode is part of that Episode, by process decomposition — a batch containing stages, a stage containing steps. It is expressed with a single self-reference, `parent_episode_id`, on the Episode itself. Depth needs no separate vocabulary of type names: an Episode three levels deep is still just an Episode row whose `parent_episode_id` points at another Episode row, which points at another. Sibling Episodes under the same parent should not overlap each other, and duration/count rollups at any level are aggregates over that level's children when the hierarchy is symmetrical. This is the same shape a manufacturing batch process has, and the same shape process mining assumes when it reconstructs cases and activities from an event log — both are examples of *how* a hierarchy gets established, not a different kind of relationship once established.

Hierarchy is independent of generation (see Section III): a parent Episode and its child can be produced by Conditions at the same generation or different ones. Nesting is a separate axis from derivation depth, and neither predicts the other.

B. Temporal Relations.

Two Episodes can also relate purely by *when* they occurred — that is, purely by comparing their underlying Intervals, with no structural relationship between the Episodes themselves at all. A fault condition's Episode and a pressure condition's Episode might overlap in time without either containing, causing, or being part of the other. **This must not be modeled as hierarchy.** An Episode whose Interval happens to fall entirely inside another Episode's Interval is not that Episode's child — collapsing "temporally nested" into `parent_episode_id` smuggles in a structural claim that is not true, and it costs you the ability to query the two relationships independently later, exactly when a question spans both ("how much of stage 2 was spent inside a fault condition" is a join between a hierarchy query and a temporal-relation query — not answerable at all if the fault Episode was wrongly filed as stage 2's child in the first place).

Rather than inventing an ad hoc vocabulary ("overlapping," "inside," "touching") that runs out of precision the moment a real question needs it ("did the fault start before or after the pressure condition it overlaps?" — "overlapping" alone cannot answer that), this model adopts **Allen's interval algebra** [10], applied to the Interval each Episode carries — a complete, unambiguous enumeration of every possible way two intervals can relate in time, with no gaps and a defined inverse for each relation.

Practical correlation rules often add a configurable tolerance window around these exact relations; that tolerance is rule metadata layered beside Allen's classifier, not an additional Allen relation.

All thirteen relations are represented in Fig. 4 as seven rows — one row per shape, since a mirror inverse is the same drawing with A and B's roles swapped, not a new shape. Each row's label names both directions at once, tagged with which interval the name describes: `(A) before · (B) after` means "A is before B" and, equivalently, "B is after A." `equals` is its own inverse, so it only requires one name.

A is always the violet bar, B is always the gold bar in every row below — the same two colors used for Events and Episodes elsewhere in this document — so the coloring stays constant and only the bars' positions change. All seven rows share a single time scale in one chart, with extra left padding reserved so the row labels do not run into the bars.

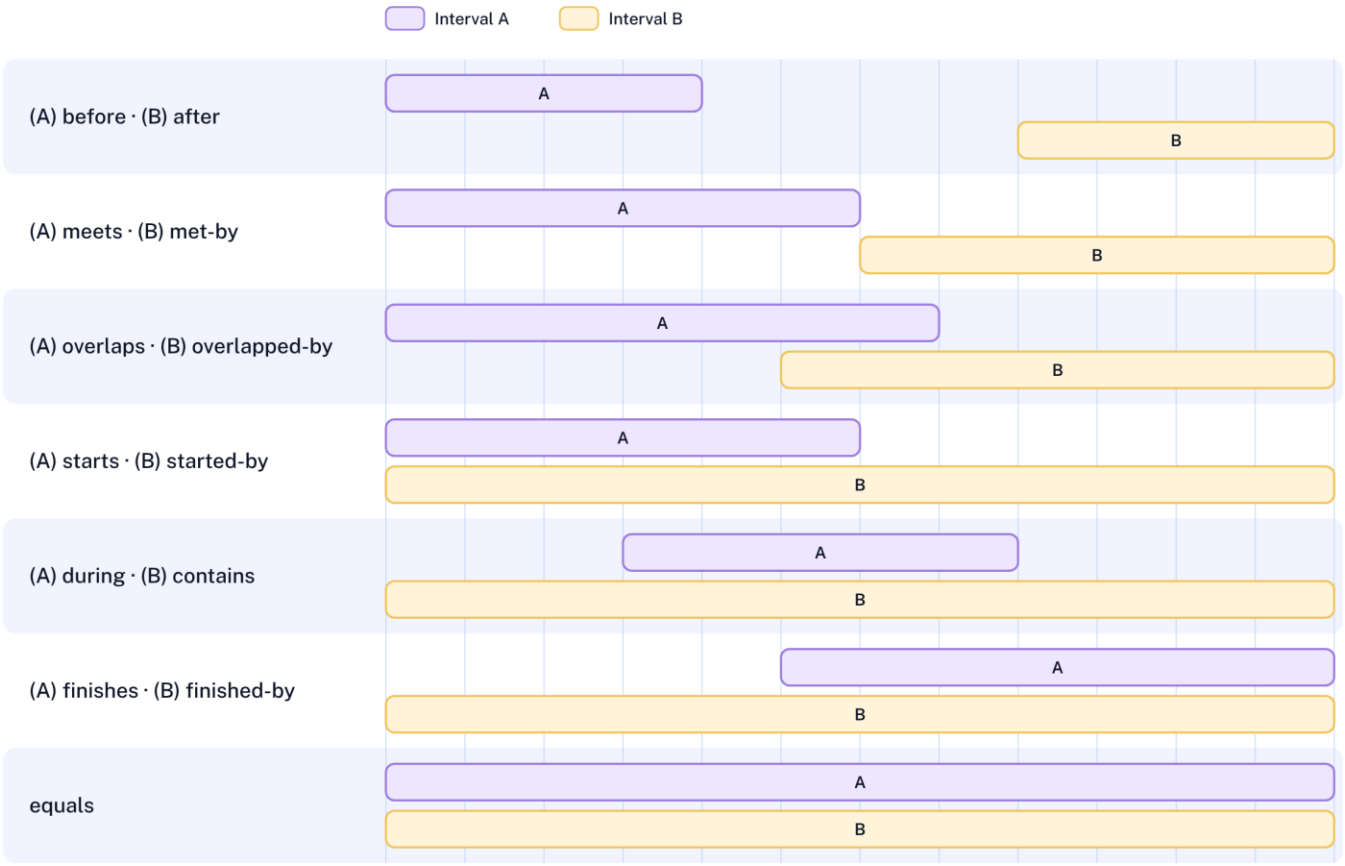


Fig. 4. Allen's interval algebra: all thirteen relations across seven relation shapes — before/after, meets/met-by, overlaps/overlapped-by, starts/started-by, during/contains, finishes/finished-by, and equals.

Table II groups the thirteen relations by kind.

TABLE II
ALLEN RELATION GROUPS

Category	Relations	Description
Disjoint	before / after	No shared time at all.
Exact-boundary relations	meets / met-by, equals	A clean handoff between two Episodes, or two independent Conditions agreeing on the same interval.
Containment without hierarchy	during / contains, starts / started-by, finishes / finished-by	One interval nests inside another with no parent/child relationship existing between them.
Partial overlap	overlaps / overlapped-by	No containment in either direction; this is the relation produced when two independently detected conditions overlap without either containing the other.

In practice, computing which of the thirteen applies is a small, cheap set of boundary comparisons once a coarse "do these overlap at all" check passes (`A.start <= B.end AND B.start <= A.end`, under whatever interval convention is in force) — not thirteen independent checks. That convention has to be stated explicitly and applied consistently: this model recommends **half-open intervals** (start inclusive, end exclusive) specifically because a closed-interval convention leaves exact-boundary cases (`meets` vs. `overlaps`) ambiguous, which is exactly where double-counting and misclassification bugs tend to appear.

As with Condition-derived Episodes generally, correlation and causal inference between related Episodes belong to a later analytical step, not to the relationship itself — `overlaps` is a temporal fact; "the fault caused the pressure excursion" is an interpretation of that fact, and keeping them distinct is the same discipline this model already applies at the Generic/Curated Boundary (see Section IX).

VIII. BOUNDEDNESS

Because a Condition's output can become another Condition's input, the model supports arbitrarily deep recursion by construction. Two properties keep that recursion safe rather than open-ended:

1. **The dependency graph between Conditions must be acyclic.** If Condition B consumes what Condition A produces, and Condition A (directly or transitively) consumes what Condition B produces, the graph has a cycle and the model has no defined meaning for it. Any implementation must reject this at the point a Condition is defined or changed, not discover it at runtime.
2. **Generation is computed from that graph, never assigned by hand.** Given an acyclic graph, every Event's generation is fully determined — `generation(base) = 0`, `generation(derived) = 1 + max(generation of everything it was derived from)`. Because the graph is known and finite before anything runs, the total number of generations is known in advance; walking all of them in order is a bounded, terminating computation, not an open-ended "keep going until nothing changes" loop with a data-dependent stopping condition.

Together, these mean recursion in this model is exactly as safe as any other directed acyclic graph traversal — which is a much weaker, more tractable claim than "safe because we bounded the iteration count," and it's the reason the model can support genuinely unbounded-looking recursion (an Episode-determination Event triggering a new Condition, triggering a new Episode, indefinitely) without needing a runtime safety valve.

IX. THE GENERIC/CURATED BOUNDARY

Any implementation of this model eventually needs a boundary where a generic Episode or Event becomes a specific, named, business artifact — a report, a dashboard metric, a downstream table with a domain-specific name. The boundary between "still part of this model" and "now a business artifact" should be a **mechanical test**, not a naming convention:

> *Something is still part of this model if it carries generation/provenance information and could still be consumed by a further Condition. Something has become a business artifact the moment it is read by an ordinary transformation and produced as something with no generation/provenance information left on it — regardless of how domain-specific or how generic either side's name happens to sound.*

This boundary is per-consumer, not per-artifact: the same Episode can simultaneously feed a business-curation step that exits the model and a further Condition that continues within it. The distinction is determined by how each consumer uses the Episode: preserving generation/provenance keeps it inside the recursive system; dropping that information produces a curated business artifact.

X. A SECOND, INDEPENDENT LENS: GOVERNANCE AND TEMPORALITY

Separately from how derived or how business-meaningful something is, two more questions turn out to matter for how something should be governed and operated:

- **Is its content business-authored, or purely mechanical/platform state?** A Condition's threshold was chosen deliberately, using domain judgment, and can be revised later on that same basis. A checkpoint or

watermark has no equivalent — nobody exercises domain judgment over what a watermark's value should be, because it carries no business content to begin with.

- **Does it drive a real-time decision, or inform later analysis?** Some data exists to trigger an action right now; other data exists to be queried, aggregated, and reported on after the fact.



Fig. 5. Placement of four representative artifact types on the Governance × Temporality plane.

These two questions are independent of each other *and* independent of the Event/Condition/Episode/generation model of Sections II–VIII — a Condition sits in the Business × Operational quadrant specifically because its content is business-authored (unlike a watermark) while still functioning as configuration consulted by a reasoning engine (unlike a curated report). That combination is why Condition definitions typically need more governance (versioned history, an explicit change-review story) than either pure platform state or ordinary curated data needs on its own — not because of an arbitrary rule, but because it inherits requirements from both axes at once.

This lens is useful for reasoning about *why* something needs the governance it needs. It is not, on its own, a reason to build reserved tags or enforcement tooling around it — that is a separate decision with its own cost, to be made only if the need for it is demonstrated repeatedly, not assumed from the existence of the lens.

XI. RELATED WORK

This model synthesizes several established ideas, credited here so the reasoning behind it can be checked independently:

- **Descriptions and Situations** [1], part of the DOLCE foundational ontology, is the formal source for "an Episode is not an Event; it is the fusion of an interval and the rule that classifies it." A Description (Condition) classifies a Situation (Episode); Events are the Situation's boundaries.
- **PROV-O** [3] is the standard W3C vocabulary for representing provenance information, including generated-by and derived-from relationships. This paper's Provenance axis is deliberately smaller: it does not attempt to model full lineage, agents, or activities, but borrows the essential distinction between observed and derived facts.
- **Runtime Event Calculus** [4] is a close precedent for the Generation axis: high-level composite events are derived from lower-level input streams through stratified definitions. This paper treats that idea at the ontology level rather than as an executable logic-programming system.
- **Event-Condition-Action and active database systems** [6], [7] are the direct lineage for separating events from rules that react to them. Their termination and rule-triggering concerns are also close to the boundedness argument in Section VIII.

- **Event-Model-F** [8], built on DOLCE+DnS Ultralite, gives a richer formal account of events, interpretations, participation, and multiple descriptions of the same occurrence. This paper deliberately simplifies that machinery for implementation-oriented systems: the Condition/Episode split keeps the interpretation pattern, while avoiding a full formal-ontology commitment.
- **Frequent episodes in event sequences** [11] use "episode" for a recurring partially ordered pattern in an event sequence. That is an established but different usage: here, an Episode is a single materialized interval satisfying a Condition, not a recurring pattern type.
- **Kimball dimensional modeling** [12] is the mainstream data-engineering neighbor to the generic/curated boundary: facts, dimensions, and slowly changing dimensions are common ways curated business artifacts are represented after interpretation. This paper's boundary is not another dimensional-modeling rule; it is a mechanical test for when generation and provenance have been dropped and the artifact has exited the recursive temporal model.
- **SOSA/SSN** [2], the W3C Semantic Sensor Network ontology, is the standard vocabulary for the sensor/observation/feature-of-interest domain this model is frequently applied to; it is a useful naming reference even without adopting its full RDF machinery.
- The **Complex Event Processing literature** [5] is the source for treating "complex events built from simpler ones" as an expected, recursive pattern rather than a special case, and for the event-stream/pattern/derived-stream execution shape this model's engines should follow.
- **Domain-Driven Design and Event Sourcing** [9] establish the distinction between a technical fact and a business-meaningful Domain Event, and the recognized pattern that one technical fact can project into multiple domain-level interpretations depending on context — the reason the same Episode can feed multiple independent curation steps.
- **Allen's Interval Algebra** [10] provides the complete, unambiguous enumeration of every possible temporal relation between two intervals; it is the adopted vocabulary for relating two Episodes to each other when that relation is temporal rather than structural (see Section VII), instead of an ad hoc, incomplete set of informal terms.

XII. CONCLUSION

This paper defined a three-category ontology — Event, Condition, and Episode — for representing facts, classifying rules, and bounded intervals without collapsing any of them into one another. It also defined Provenance and Generation as independent axes on Events and showed, through the four-cell diagnostic matrix, why derivation source and business meaning must remain separate design concerns. The same vocabulary supports a coverage claim for Episode boundary detection: source-native keys, rule toggles, inference-based detectors, and procedural decomposition all affect boundary provenance or hierarchy rather than requiring new Episode types. Fuzzy boundaries, weighted evidence, and predictive outputs fit within the model as metadata on Events or Episodes while preserving a boolean materialized Episode. The model further distinguishes structural Hierarchy from computed Temporal Relations between Episodes, uses an acyclic dependency graph to make recursive derivation bounded by construction, and defines the Generic/Curated Boundary as a mechanical test for when an artifact exits the recursive temporal model. Taken together, these constraints provide a compact, implementation-agnostic vocabulary for building event-driven systems that reason about temporal intervals without losing provenance, overloading hierarchy, or confusing technical derivation with business meaning.

APPENDIX: GLOSSARY OF TERMS

TABLE III
GLOSSARY OF TERMS

Term	Definition
Event	A punctual occurrence.
Condition	A rule that classifies a stream of Events.
Interval	A bounded span of time — a start and an end — with no Condition attached. What Allen's interval algebra operates on; what a native segmentation source hands you before a Condition gives it meaning.
Episode	An Interval, bounded by two Events, fused with the Condition that satisfies it.
Provenance	Whether an Event was System-Observed or Synthetic.
Generation	How many derivation hops separate an Event from an observation; always 0 for System-Observed, always increasing along any derivation chain.
Boundedness	The requirement that the Condition dependency graph be acyclic, making generation computable and recursion terminating by construction.
Generic/Curated Boundary	The mechanical (not naming-based) test for whether something still participates in the model or has become a business artifact.

Term	Definition
Hierarchy (between Episodes)	A structural relationship (parent_episode_id) — this Episode is part of that Episode by process decomposition. Independent of generation.
Temporal Relation (between Episodes)	A computed, non-structural relationship between two otherwise unrelated Episodes, described using Allen's interval algebra rather than ad hoc terms.

REFERENCES

[1] A. Gangemi and P. Mika, "Understanding the Semantic Web through Descriptions and Situations," in *Proc. OTM Confederated Int. Conf. on the Move to Meaningful Internet Systems (CoopIS, DOA, ODBASE)*, Catania, Italy, 2003, pp. 689–706.

[2] World Wide Web Consortium, "Semantic Sensor Network Ontology (SOSA/SSN)," W3C Recommendation, 2017.

[3] T. Lebo, S. Sahoo, and D. McGuinness, Eds., "PROV-O: The PROV Ontology," W3C Recommendation, Apr. 30, 2013.

[4] A. Artikis, M. Sergot, and G. Paliouras, "An event calculus for event recognition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 4, pp. 895–908, Apr. 2015.

[5] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA: Addison-Wesley, 2002.

[6] J. Widom and S. Ceri, Eds., *Active Database Systems: Triggers and Rules for Advanced Database Processing*. San Francisco, CA: Morgan Kaufmann, 1996.

[7] A. Aiken, J. Widom, and J. M. Hellerstein, "Behavior of database production rules: Termination, confluence, and observable determinism," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, San Diego, CA, 1992, pp. 59–68.

[8] A. Scherp, T. Franz, C. Saathoff, and S. Staab, "F—A model of events based on the foundational ontology DOLCE+DnS Ultralite," in *Proc. 5th Int. Conf. on Knowledge Capture (K-CAP)*, Redondo Beach, CA, 2009, pp. 137–144.

[9] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley, 2003.

[10] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983.

[11] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259–289, 1997.

[12] R. Kimball, *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. New York, NY: John Wiley & Sons, 1996.