The Emergent Truth: From Declarative Simplicity to Conceptual Completeness

Author: EJ Alexandra
Email: start@anabstractlevel.com
Affiliations: ssot.me & effortlessAPI.com

Abstract

We begin with the simplest possible statements—e.g., distinguishing “something” (1) from “nothing” (0)—and observe how additional facts and constraints organically yield deep insights, like the Pythagorean theorem. This purely declarative logic, unburdened by any imperative “update steps,” naturally extends beyond simple geometry: it applies just as well to baseball (runs, outs, stats) or quantum mechanics (wavefunction superposition, measurement). Rather than delving into domain-specific complexities, we highlight how the same universal “rulebook” of five declarative primitives—**Schema**, **Data**, **Lookups**, **Aggregations**, and **Lambda Calculated Fields**—effortlessly captures reality across very different fields, without resorting to specialized syntax or procedural code.

By exploring three illustrative domains—**triangleness**, **quantum walks**, and **baseball**—this paper shows that **emergent truths** arise simply from enumerated facts, relationships, and aggregator formulas, all living in a snapshot-consistent environment. We ultimately introduce the **Conceptual Model Completeness Conjecture (CMCC)** as the unifying framework behind these examples: once you grasp how each domain’s logic is spelled out structurally (and never with “setScore()” or “collapseWavefunction()” calls), you can see how “truth” emerges purely from the lattice of defined facts. **No imperative steps required**—just a small handful of domain-agnostic building blocks that scale from basic geometry to advanced physics and real-world sports.

Table of Contents

1. Introduction: The Emergence of Truth

1.1 Purpose of This Paper

This paper offers a narrative demonstration of how a purely declarative modeling approach—free of any imperative "update" calls—lets us capture the logic of three seemingly distinct domains: basic geometry, baseball scoring, and quantum measurement. We rely on just five primitives (Schema, Data, Lookups, Aggregations, and Lambda Calculated Fields) that unify knowledge modeling across these domains. Rather than proving Turing-completeness or diving into domain-specific minutiae, we focus on the big picture: **why** enumerated facts and aggregator formulas are enough to uncover "theorems" like Pythagoras, "three outs end an inning," and quantum collapse.

1.2 Motivation: From "I Think, Therefore I Am" to Declarative Worlds

Philosophical questions about truth often begin with René Descartes's cornerstone statement: "I think, therefore I am." Simply by acknowledging that **something** exists rather than nothing, we introduce the binary notion of presence (1) versus absence (0). Remarkably, this elementary distinction—1 vs. 0—can bootstrap an entire conceptual universe:

1. **Bits**: Once we establish 1 and 0, we can form binary strings.
2. **Numbers**: Strings of bits can represent numeric values.
3. **Coordinates**: We map these numbers into positions (x, y, etc.) and define points.
4. **Shapes**: By declaring points and the relationships among them, geometric structures "emerge," leading us straight to properties like angles and distances.

No one ever calls a function like `CreateTriangle()`; we simply **declare** that three points are connected, and a triangle necessarily appears in the conceptual space. From these connections alone, we inevitably reach geometry's famous result: the Pythagorean theorem. Thus, "theorems" arise from enumerated facts and constraints, not from stepwise procedures.

1.3 From Geometry to Baseball (and Beyond)

Having illustrated how binary facts yield geometric results, we then show how the same declarative mindset applies to domains that **seem** procedural, such as:

- **Baseball**: A sport that appears full of stepwise increments—runs, outs, innings—but which can just as easily be modeled by declaring events (`RunEvent`, `OutEvent`) and letting aggregator formulas compute the scoreboard.
- **Quantum Walk**: Where wavefunction superposition and measurement typically imply "collapsing" code routines. Instead, we treat them as data and constraints, allowing aggregator logic to handle amplitude distributions and measurement outcomes.

By comparing geometry, baseball, and quantum mechanics in one unified story, we highlight that no specialized DSL or imperative logic is needed. Instead, enumerating domain facts in JSON (or any data format) combined with aggregator fields can deliver the "magic" of emergent truths—be they Pythagoras's theorem, a final score, or a collapsed wavefunction.

2. Starting at Zero: Something versus Nothing

When we say “something exists,” we tacitly admit there’s a distinction between **presence** (1) and **absence** (0). This binary viewpoint forms the bedrock upon which all further logic stands.

2.1 Binary as the Declarative Foundation

1. **Defining Bits**: We start by asserting that `1` denotes existence and `0` denotes non-existence.
2. **Binary Representation**: A sequence of such bits is recognized as a valid representation for higher-level concepts—like integers, floats, or any enumerated code.

Hence, an essential truth emerges by definition: “Numbers are declaratively represented using binary.”

2.1.1 From Bits to Floats

Once we assert that 1 represents “existence” and 0 represents “absence,” it’s natural to wonder how such bits become meaningful quantities like integer coordinates or floating-point values. In practice, we rely on standardized mappings from bit patterns to numeric values:

1. **Binary Integers (Unsigned):**
    - A straightforward way to interpret a sequence of nnn bits, bn−1bn−2…b1b0b_{n-1} b_{n-2} \ldots b_{1} b_{0}bn−1bn−2…b1b0, is as an unsigned integer in base 2. For example, the 4-bit pattern 1011\texttt{1011}1011 can be interpreted as 1×23+0×22+1×21+1×20=111 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 111×23+0×22+1×21+1×20=11 in decimal.
2. **Signed Integers (Two’s Complement):**
    - Modern computer systems typically represent negative numbers using two’s complement. For an nnn-bit value, the most significant bit can indicate negativity. For instance, an 8-bit pattern 11111111\texttt{11111111}11111111 often corresponds to −1-1−1. This encoding is a purely declarative choice that a system can store and interpret consistently.
3. **Floating-Point Formats (e.g., IEEE 754):**
    - Real-number approximations typically follow the IEEE 754 standard, splitting a bit pattern into sign, exponent, and mantissa. This lets us represent decimal values like 12.512.512.5, albeit with rounding. For example, a 32-bit float might look like 01000001010010000000000000000000\texttt{01000001010010000000000000000000}01000001010010000000000000000000, internally representing 12.512.512.5.

Crucially, no function `constructFloat(bitPattern)` is “run” in an imperative sense. We simply declare that a given bit pattern is *interpreted* as a floating-point coordinate (x,y)(x, y)(x,y). Once established, any aggregator-based logic (e.g., measuring distances between points) can rely on these numeric values. Real-world systems layer these standard definitions of integer or float over raw bits, ensuring consistent numeric semantics.

2.2 Constructing a Conceptual Space

Once we have numbers, it’s natural to **map** these values into coordinates:

1. **Numerical Coordinates**: We declare that two numbers `(x, y)` define a point in a conceptual (e.g., Cartesian) space.
2. **Points, Not Procedures**: We don't run a function "makePoint(…)"; we simply record that `(x, y)` is a point. That declaration alone is enough for a declarative system to recognize the point's existence.

It's worth noting that in practical systems, mapping numbers to coordinates often involves floating-point approximations, which must be handled carefully so that derived facts (like exact angles) do not trigger false 'contradictions' caused by rounding errors.

2.3 Emergence of Geometric Structures

From these points, lines and polygons appear as logical consequences:

- **Lines**: Connecting two points yields a line segment. Its length, direction, or angles with other segments follow automatically from aggregator formulas.
- **Triangles**: Declaring a third point connected to the previous two creates a closed shape with three edges. A final aggregator clarifies the shape as a "triangle," complete with an angle sum of 180°.

2.4 Right Triangles Appear

Refining these facts, we add:

> "If one internal angle is 90°, the shape is called a right triangle."

Now a hypotenuse emerges as the longest side opposite that angle. We label it `c`, with the other sides `a` and `b`. Again, no function calls—just structured facts about angles and side lengths.

2.5 The Pythagorean Theorem as an Emergent Fact

From the preceding constraints, $a^2 + b^2 = c^2$ becomes unavoidable. Nothing in the data instructs the system to "use Pythagoras"; rather, once a triangle is declared right-angled, aggregator logic that measures lengths from coordinates will **necessarily** reveal the Pythagorean relationship. Any contradictory assertion (like forcing $a^2 + b^2 \neq c^2$ in a right triangle) yields an inconsistency. Thus, Pythagoras exemplifies how powerful truths can surface naturally in a declarative environment.

---

3. Building a Conceptual Universe: Points, Lines, Shapes

Binary-encoded coordinates let us place any number of points in a 2D (or higher) space. Once we declare enough points, lines, and angles, the entire world of Euclidean geometry unfolds—without writing a single imperative routine like "drawShape()" or "computeAngle()."

3.1 Mapping Bits onto Coordinates

- **Numbers as Data**: Each coordinate value is just a bitstring that the system stores as "this is the x-part, this is the y-part."
- **Declarative Syntax**: We label `(x, y)` as a `Point`, letting aggregators handle any resulting lengths or angles.

3.2 From Points to Shapes

- **Line Segments**: Indicate that two points are "connected," and a line segment emerges. The system can track length and direction through aggregator formulas like distance calculations.
- **Angles**: With multiple connected segments, aggregator constraints define angles at the intersection point. If an angle measures 90°, the data tags it as a "right angle."

3.3 Triangles, Quadrilaterals, and Beyond

Expanding from two or three connected points to four or more leads us to quadrilaterals, polygons of n-sides, and so on:

- **Classification**: Aggregators can classify shapes (e.g., triangles vs. rectangles) by checking edge counts and angle values.
- **Dimensional Extension**: Extending to 3D (adding a z-value) or higher dimension is equally straightforward—simply store extra coordinates and let aggregator formulas do the rest.

All these structures are discovered by enumerating consistent data, not by imperative geometry calls.

3.4 Preview of the Same Logic in Baseball

This same principle—declaring facts, letting aggregators yield higher-level truths—applies seamlessly in **non-geometric** domains like baseball. Instead of angles and edges, we have:

- **RunEvent** and **OutEvent** records, referencing which team or inning they belong to.
- **Aggregators** for total runs and outs, automatically updating a "score" or signaling "three outs."

We'll see that, as in geometry, "no function `incrementScore()` is required." The aggregator-based approach unifies geometry, baseball, and even quantum mechanics under one declarative framework.

3.5 Conclusion of the Foundational Layer

By this point, we have:

1. A binary building block enabling numeric representation,
2. A method to declare points and shapes purely by storing data and constraints,
3. A proven pattern of "inevitable facts" emerges once we fix consistent relationships (e.g., the Pythagorean theorem).

With these fundamentals in hand, we now pivot to more "procedural-seeming" domains—starting with baseball—to highlight just how universal the declarative approach can be.

---

4. Emergent Geometry: No "Theorem," Just Constraints

Having introduced lines and polygons, we now see that many "theorems" from classical geometry are actually **inevitable facts**—provided we **declare** a consistent set of relationships (like "three edges, one angle at 90°," etc.). None of these outcomes are coded or computed step by step; they naturally **arise** from the constraints.

4.1 Listing the Facts: "Three Angles," "Sum = 180°," "Right Triangle"

For a **triangle**, these basic facts become aggregator definitions:

- **Angle Count**: “Number of internal angles = 3.”
- **Angle Sum**: “Sum of internal angles = 180°.”
- **Right Triangle**: “If one angle = 90°, label the shape ‘right triangle’.”

In a typical geometry curriculum, you might see an entire proof dedicated to angle sums. Here, it’s simply encoded as a constraint in the aggregator logic—**no** separate proposition or lemma required.

4.2 “Hypotenuse,” “Legs,” and Observations That Appear Without “Proof”

Once you declare a triangle to be “right-angled,” further labels or definitions follow suit:

- **Hypotenuse**: The aggregator that locates the longest side automatically identifies it as “c.”
- **Legs**: The other two sides, typically labeled “a” and “b,” are recognized by an aggregator or a “lambda calculated field” (e.g., “IF side.length < hypotenuse THEN side is a leg”).

Observe that these are still facts, not commands: The system “knows” which side is longest, so the notion of “hypotenuse” is forced by the constraints, not by an explicit assignment.

4.3 The Pythagorean Relationship as an “Inevitable Fact”

Here we arrive at one of geometry’s crown jewels, but from a purely **constraint-based** vantage. Once an aggregator recognizes a right triangle, the squared lengths of the legs and hypotenuse automatically satisfy a2+b2=c2a^2 + b^2 = c^2a2+b2=c2. Because this relationship *cannot* be contradicted without invalidating prior definitions (angles, side lengths, etc.), it appears as a **logical necessity**—not a manually entered rule. For clarity:

- **No “Theorem”**: We never typed “theorem Pythagoras.”
- **No “Proof”**: We never stepped through a proof in the classic sense.
- **Only Declarative Constraints**: We enumerated angles, identified a right angle, recognized a longest side, and the aggregator formulas forced the outcome.

This underscores the emergent nature of geometry in a purely declarative model: once you set the shape and angle constraints, the Pythagorean theorem is *unavoidable*.

In real-world data, strict equalities like $a^2 + b^2 = c^2$ might instead be replaced by a tolerance check—for example, $|a^2 + b^2 - c^2| < \varepsilon$—thereby reflecting the inevitable imprecision of floating-point or measured lengths.

4.4 The Role of Consistency: Why Contradicting the Fact List Gets Harder

If any newly introduced “fact” (e.g., “sum of angles = 200°” for a Euclidean triangle) conflicts with these aggregator rules, the system simply flags an inconsistency—there’s no feasible way to reconcile the new claim with the existing definitions. As more facts accumulate, contradictions become both more likely if you deviate and *more obviously impossible* unless you correct the original statement. This is how purely declarative systems **enforce** geometry: not by code, but by unyielding structural consistency.

4.5 Handling Floating-Point Tolerances

Real-world coordinates or amplitude measurements are rarely exact. To accommodate numerical imprecision, a declarative system can implement tolerance thresholds (ε) in its aggregator checks. For instance, instead of enforcing $\sum(\text{angles}) = 180^\circ$, the aggregator might check $\bigl|\sum(\text{angles}) - 180\bigr| < \varepsilon$.

Below is a reference table of default ε-values used in different domains:

Domain	Constraint	Recommended ε	Notes
Geometry (2D)	Sum of angles = 180°	1e-7	For standard single-precision float calculations.
Geometry (3D)	Edge length constraints	1e-5	3D modeling may involve slightly looser bounds.
Baseball Stats	Batting average ≈ aggregator	1e-9	Usually integer-based, less floating rounding.
Quantum Amplitudes	(	a	^2 +

Any aggregator formula, such as `IF ABS(SUM(angles) - 180) < epsilon THEN shape=triangle`, uses these tolerance levels to avoid false "contradictions" from minimal floating-point errors. Systems can define domain-specific ε-values or even dynamically adapt them based on measured real-world error margins.

---

5. Extending the Same Logic to Broader Domains

Up to this point, we've illustrated how geometry emerges from enumerated facts about points, lines, and angles. But the power of this approach shines even brighter when we step outside of classical geometry and into domains people typically describe as "procedural." Here, we show that baseball scoring and quantum phenomena can be framed **exactly** the same way.

5.1 Baseball: Runs, Outs, and Score Without "SetScore()"

In a typical baseball simulation, you might see code like `incrementRuns(team, 1)` or `setOuts(outs + 1)`. A purely declarative approach replaces these imperative steps with **events** and **aggregator fields**:

- **RunEvent**: A factual record stating, “This run occurred in the 3rd inning, referencing player X.”
- **OutEvent**: Another factual record, “This out occurred in the bottom of the 5th inning, referencing player Y.”

From these facts alone, aggregator formulas compute totals—runs per team, outs per inning—and *enforce* baseball’s transitions (e.g., three outs end an inning) by referencing these aggregated values. No single function call says “switch innings now.” The scoreboard effectively updates itself through constraints like:

- “inning ends if `OUTS` >= 3”
- “game ends if `inning` >= 9 and `runs_teamA != runs_teamB`”

Just as we never wrote a “theorem” in geometry, we never write `updateScore()` in baseball. The aggregator logic reveals the game state from the raw facts of events.

5.2 Quantum Physics: “Wave Function Collapse” Without “Collapse()”

At first glance, quantum phenomena (superposition, interference, measurement) seem too esoteric for the same method that handled triangles and baseball. Yet the core principle is identical: once we store amplitude data for a wavefunction, aggregator formulas can track interference patterns, probabilities, or partial states.

- **Wavefunction**: A record storing amplitude values at each possible location or state.
- **MeasurementEvent**: A data record that references a specific wave function plus a measurement outcome.

In practical quantum data modeling, partial knowledge or incomplete measurement records may require representing some states with explicit uncertainty fields rather than strictly enumerated amplitudes.

Similar to baseball, no code calls `collapseWavefunction()`. Instead, aggregator constraints ensure that the wavefunction’s superposed amplitudes “reduce” to the measured outcome once the event is declared. Any contradictory measurement outcome (e.g., measuring spin-up and spin-down simultaneously) flags an inconsistency. Thus, quantum collapse is just as declarative as “triangle side lengths” or “baseball scoring.”

5.2.1 Example: Normalization Constraint

For instance, consider a wavefunction record storing amplitudes for possible states ψ=(a1,a2,…,an)\psi = (a_1, a_2, \dots, a_n)ψ=(a1,a2,…,an). A simple aggregator might compute the total probability ∑i∣ai∣2\sum_i |a_i|^2∑i∣ai∣2, and a declarative constraint can require that this sum equals 1. If a MeasurementEvent declares an outcome inconsistent with normalized amplitudes, the system flags an immediate contradiction—mirroring how physical principles disallow mutually exclusive measurement results.

Declarative Model for a Single Qubit

Entity / Field	Description	Example Value
Wavefunction	Represents a single qubit state, with two amplitudes (aaa for (	0\rangle) and bbb for (

`amplitude_0` (scalar float)	Real or complex amplitude for (	0\rangle).
`amplitude_1` (scalar float)	Real or complex amplitude for (	1\rangle).
Aggregator: normalization	Enforces (	a
MeasurementEvent	References a specific wavefunction plus an outcome (000 or 111).	outcome = “1”
Aggregator: collapse check	If `MeasurementEvent.outcome = 1`, the aggregator sets (a,b)=(0,1)(a,b) = (0,1)(a,b)=(0,1).	N/A

In this **qubit** example, the aggregator `normalization` ensures that whenever a user or process attempts to store amplitude values (a,b)(a, b)(a,b), we have ∣a∣2+∣b∣2≈1|a|^2 + |b|^2 \approx 1∣a∣2+∣b∣2≈1. A `MeasurementEvent` record “collapses” the wavefunction by declaring the outcome. If the aggregator sees a contradictory event (e.g., outcome is “0” and “1” at the same time), the system flags an inconsistency.

5.3 Any Domain: From Edges and Angles to Observers and Particles

The key insight is that **Schema, Data, Lookups, Aggregations, and Lambda Calculated Fields** form a universal basis for *all* manner of knowledge. Whether describing a triangle or a wave function, you define factual entities (e.g., edges, angles, or amplitude distributions) and aggregator-driven relationships (e.g., “sum angles to 180°,” or “probabilities must total 1”). The “rest” simply *happens* by structural necessity.

This approach reveals that “procedural” illusions—like incrementing runs or forcing wavefunction collapse—are in fact emergent aggregator outcomes. Once enough domain data is declared, any higher-level rule or “theorem” you normally would code is simply an inevitable result of the constraints.

---

6. A Snapshot-Consistent Environment: What It Means

Everything we’ve described—triangles, baseball events, quantum wavefunction states—relies on a core principle: **all facts and aggregator results exist within a single, consistent snapshot** of the data. In other words, at any moment, the system’s declared facts and derived fields must align with each other, forming a coherent picture of “what is true right now.”6.

6.1 Declarative vs. Imperative: The Fundamental Distinction

An **imperative** approach typically involves updating variables in a sequence:

1. `score = score + 1`
2. If `outs == 3`, then move to next inning
3. etc.

By contrast, a **declarative** approach never “runs” these steps; it merely **states** the relationship:

- “`score` is the sum of all RunEvents referencing this team.”
- “When the aggregator `outs` reaches 3, the inning is finished.”

Nothing *executes* at runtime to push these states forward; the aggregator logic itself ensures that whenever you look at the data snapshot, each aggregator field has the correct derived value, and each constraint is enforced.

6.2 Imperative vs. Declarative

Below is a compact table illustrating how each example domain typically uses imperative logic and how it transforms under a declarative, CMCC-based approach:

Domain	Imperative Approach	Declarative (CMCC) Approach
Geometry	- Manually compute angles or apply theorems via stepwise proofs - `computeAngles()` or `provePythagoras()` calls	- Declare points, edges, angles as data - Aggregators (like `sum_of_angles`) and constraints automatically yield Pythagoras, etc.
Baseball	- Procedural calls like `incrementScore(team)` - “End inning if outs == 3” coded imperatively	- Declare `RunEvent` and `OutEvent` facts - Aggregators compute total runs, outs, and handle inning transitions through constraints
Quantum	- Functions like `collapseWavefunction()` in simulation steps - Imperative loop over states or measurement routines	- Declare amplitude distributions and `MeasurementEvent` - Aggregator constraints unify superposition and collapse (no `collapse()` call needed)

In each domain, the CMCC approach replaces explicit function calls and stepwise updates with a small set of declarative primitives (schema, data, lookups, aggregations, and lambdas). “Truth” then emerges from consistent snapshots of these facts and constraints—no manual procedure required.

6.3 Why Aggregators and Constraints Capture All Necessary Logic

Aggregators (like `SUM`, `COUNT`, or `MAX`) plus conditional constraints can encode everything from “triangle angle sums” to “wavefunction probability amplitudes.” The key is:

- **No partial updates**: You never see a half-updated state. Either the aggregator formula has enough facts to compute a consistent result, or it flags an inconsistency.

- **No hidden side effects**: Each aggregator's output depends solely on the underlying data, so changes to any input fact are reflected automatically.
- **Universality of Conditions**: If you can define a domain rule as a function of existing facts—be it "angles must sum to 180°" or "the total probability must be 1"—you can express it as a constraint or aggregator.

So long as the system can store these relationships and maintain snapshot consistency, you never need manual procedures like `recalculateAngles()` or `collapseWavefunction()`.

Potential Cycles and Resolution

One caveat is that aggregators must not form circular dependencies, such as A depending on B while B also depends on A. In real implementations, the system typically enforces a directed acyclic graph (DAG) structure for aggregator formulas; any cycle is flagged at definition time to prevent non-terminating evaluation or ambiguous results. This requirement ensures each aggregator's output can be computed in a well-defined sequence at every snapshot.

6.4 The Tension Between "Data" and "Derived State"

In traditional architectures, you often keep "data" (e.g., table rows) and "derived state" (aggregations, indexes, caches) as separate layers, sometimes leading to race conditions or stale values. A **snapshot-consistent** environment folds them into one conceptual domain: whenever new facts are introduced, aggregator fields update *in the same snapshot*. This ensures:

1. **Atomicity**: Either all facts and aggregator results are committed together, or none are.
2. **Consistency**: Contradictions are caught immediately, rather than lurking in partial states.
3. **Isolation & Durability**: Other processes see only the final stable outcome, not an intermediate stage.

For geometry, this means you can't "partly" declare a triangle that only has two edges. For baseball, you won't see a partial increment of runs without also updating outs if needed. For quantum, you won't measure half a collapsed state. Everything is always consistent in the final committed snapshot.

Under concurrency, aggregator formulas must resolve in a consistent order; this can be implemented via transaction commits that atomically lock relevant fields to maintain snapshot coherence.

6.5 Handling Contradictions and Flagging Errors

In any purely declarative system, data might arrive that conflicts with existing facts or aggregator constraints. When this occurs in a CMCC environment, the system flags an inconsistency rather than applying partial or contradictory updates.

Example: Invalid Triangle Angle Sum

1. Suppose we have a polygon declared with `edge_count = 3`, indicating it's a triangle.
2. The angle aggregator is constrained by "sum of angles = 180°" for Euclidean geometry.
3. If a user or process mistakenly declares an angle set of `[90°, 60°, 50°]`, the aggregator tries to enforce `90 + 60 + 50 = 200`, which contradicts the "triangle sum = 180" rule.

When the snapshot is evaluated, the system detects an unresolvable conflict—your aggregator logic or constraints effectively produce an error. A simple diagnostic message might look like this:

pgsql

```
ERROR: Aggregator conflict on Polygon#123
 - Declared as triangle (edge_count=3), implies sum_of_angles=180
 - Actual angles declared: [90, 60, 50], sum=200
 - Contradiction: "Triangle angle sum mismatch"
Transaction aborted. Please correct the angle data or remove the triangle constraint.
```

How Contradictions Are Resolved

- **No partial commits**: If the system enforces snapshot consistency, it refuses to finalize the new or updated facts until the conflict is resolved.
- **Possible user or system action**: The user can adjust angles, change `edge_count`, or remove contradictory data.
- **End result**: The committed snapshot always remains consistent, and invalid data is never allowed to "coexist" in partial form.

This same pattern applies in baseball ("four outs in one half-inning" triggers a conflict) or quantum ("simultaneous contradictory measurement outcomes"). By requiring all new facts to align with existing aggregators, the system ensures emergent truths remain globally consistent.

Concurrency Considerations

In multi-user systems, new facts may arrive simultaneously from different processes. The snapshot-consistent environment solves this by bundling changes into transactions that either commit atomically or roll back on conflict. Thus, even under high concurrency, you never see partial aggregator updates—only fully consistent snapshots. This model naturally aligns with ACID transaction semantics in many modern databases.

6.6 Further Contradiction Scenarios

Contradictions aren't limited to geometric angle sums. In baseball and quantum mechanics, analogous paradoxes arise:

1. **Baseball: Four Outs in One Half-Inning**
    - Suppose we have an `InningHalf` record with an aggregator `outs = COUNT(OutEvent)`. By definition, `outs ≥ 3` triggers an inning change. If someone tries to insert a fourth `OutEvent` while the same half-inning is still active, the aggregator logic flags a conflict.

Error Message Example:
pgsql
Copy

```
ERROR: Aggregator conflict on InningHalf#451

  - outs aggregator = 3 triggers inning completion

  - Attempted insertion of OutEvent => outs=4

  - Contradiction: "A single half-inning cannot exceed 3 outs"
```

○

2. **Quantum: Contradictory Measurement Outcomes**

   - A wavefunction aggregator might enforce ∣a∣2+∣b∣2=1|a|^2 + |b|^2 = 1∣a∣2+∣b∣2=1 for a qubit. A `MeasurementEvent` that claims both outcome="0" and outcome="1" with 100% probability cannot be reconciled.

Error Message Example:
csharp
Copy

```
ERROR: WavefunctionCollapseConflict on Qubit#17

  - aggregator enforces: total probability must be 1

  - measurement recorded outcome=|0> with p=1.0 AND outcome=|1> with p=1.0

  - Contradiction: "Two mutually exclusive measurement outcomes declared simultaneously."
```

   -
   - In a snapshot-based environment, either the second measurement event is rejected, or the system commits only one outcome. No "partial collapse" can remain in the final snapshot.

These examples underscore a central principle: **once the aggregator constraints are defined, the system automatically blocks or flags any update that breaches them.** This mechanism is domain-agnostic, whether the conflict involves geometry, baseball scoring, or quantum measurement. As the fact lattice grows, contradictions become both more detectable and more definitively rejected.

---

7. Revealing the Punchline: The CMCC

Having walked through geometry, baseball, and quantum domains, we can now articulate the **Conceptual Model Completeness Conjecture (CMCC)**: *Any domain's entire logic can be captured by enumerating facts, relationships, aggregator formulas, and conditional fields within a snapshot-consistent model—requiring no additional imperative instructions.*

7.1 A Universal Rulebook for Schema (S), Data (D), Lookups (L), Aggregations (A), and Lambdas (F)

CMCC stands on **five** declarative primitives:

1. **Schema (S)**: Defines each entity type (e.g., "Polygon," "Angle," "BaseballGame," "Wavefunction") and its fields.
2. **Data (D)**: Concrete records or events (e.g., "Angle with 53°," "RunEvent in the 3rd inning," "MeasurementEvent at time t").
3. **Lookups (L)**: References linking records (e.g., "this Edge belongs to that Polygon," "this measurement references that wavefunction").
4. **Aggregations (A)**: Summations, counts, maxima, or derived metrics computed purely from data/lookups (e.g., "edge_count = COUNT(this.edges)").

5. **Lambda Calculated Fields (F)**: Declarative constraints or if-then expressions that define how domain rules apply. Example: “IF angle_degrees = 90, THEN shape_type = right_triangle.”

Crucially, none of these primitives require specifying *how* to update or iterate. The runtime enforces consistency for each aggregator or condition whenever the underlying data changes.

7.2 Surprise: You’ve Already Been Using the Logic of the CMCC All Along

In exploring geometry, baseball, and quantum examples, we’ve effectively been *using* CMCC primitives—without calling them by name:

- **Schema & Data**: We introduced “polygon,” “run event,” “wavefunction.”
- **Lookups**: We said “three points reference each other as edges.”
- **Aggregations**: We used “sum of angles,” “count of runs,” “maximum amplitude.”
- **Lambda Fields**: We had “if angle = 90°, then it’s a right triangle,” “if outs = 3, the inning is done,” “if measurement is declared, collapse wavefunction to the measured state.”

So the “magic” you witnessed—Pythagoras appearing unbidden, baseball scoreboard updates with no `setScore()`, quantum collapse with no `collapseWavefunction()`—isn’t magic at all. It’s the natural consequence of enumerating domain constraints in a purely declarative schema.

7.3 Tying It All Together: Why No Single Step in the Process Required a Theorem

The standard approach to geometry or scoring or quantum measurement would be to define a function for each special step (“prove Pythagoras,” “increment runs,” “invoke wavefunction collapse”). But if you adopt CMCC’s perspective, each domain property is simply a forced outcome of the aggregator definitions and constraints you’ve declared. Because no theorem or stepwise procedure is needed, **truth emerges** from the structure itself.

7.4 🔴 HARDCORE FALSIFICATION CHECKLIST

⚠️ STOP! BEFORE YOU CLAIM CMCC FAILS, ENSURE THAT YOU’VE DONE THIS: ⚠️

✅ Step 1: DO NOT LIST 5 POSSIBLE PROBLEMS
(Red Flags: IF your falsification attempt includes words like Could, May, Might, or Maybe, that is not a falsification, it is a presupposition - that is going to melt away under scrutiny)

✅ Step 2: PICK A SPECIFIC RULE YOU THINK BREAKS CMCC
(Write it down. If you didn’t, you’re just assuming failure instead of testing it.)

✅ Step 3: DECOMPOSE IT INTO CMCC PRIMITIVES (S, D, L, A, F)
(Write how it could be represented declaratively. If you skipped this, you didn’t actually test it.)

✅ Step 4: CHECK IF THE RUNTIME ENGINE CAN HANDLE IT
💡 Remember: CMCC defines the WHAT, not the HOW. The runtime executes the rules. Performance, and scaling are runtime concerns. Is your rule actually a runtime concern?

✅ **Step 5: IF YOU STILL THINK IT FAILS, TRY AGAIN**
Seriously. Every time I thought I falsified it, I realized I was looking at it wrong. Are you sure you aren't just assuming imperative execution is required?

🚀 **IF YOU STILL THINK CMCC FAILS… EMAIL ME YOUR CASE!** 👍
(And expect me to ask if you really followed these steps. 😂)

So far, every known domain that is Turing-computable has been mapped onto these five primitives. Nonetheless, CMCC explicitly invites scrutiny: **if** a domain stumps these declarative components, it would either demonstrate a need for new primitives or disprove CMCC's universal scope. In this sense, CMCC is not just a theoretical claim but a research call to "try and break it."

---

8. Discussion & Implications

Now that we've framed geometry, baseball, and quantum mechanics under one declarative umbrella, let's consider how this approach affects broader domains—like AI, enterprise data management, and multi-domain integrations.

8.1 The Power of Emergent Meaning in Knowledge Modeling

In typical data systems, domain logic is scattered in imperative code. By consolidating logic in aggregator formulas and constraints, you gain:

- **Transparency**: Any domain rule is discoverable as a schema or aggregator definition, rather than hidden in function calls.
- **Maintainability**: Changing a domain rule (e.g., adjusting baseball's "mercy rule" threshold) is as simple as editing a single aggregator or lambda field.
- **Interoperability**: Because no specialized syntax is needed, it's trivial to integrate multiple domains—like geometry plus quantum or baseball plus economics—by merging or referencing each other's aggregator fields and lookups.

8.2 Implications for Software, Data Management, and AI Reasoning

Consider the impact on large-scale software:

1. **Reduced Complexity**: You eliminate a raft of "update" or "synchronization" procedures and unify them into a small set of aggregator definitions.
2. **Clear Auditing**: Every emergent outcome (like a final score or collapsed wavefunction) can be traced back to the factual records that drove it—no hidden side effects.
3. **AI Transparency**: Declarative knowledge bases align well with explainable AI, since derived conclusions (like "why is this shape a triangle?") are pinned to aggregator logic, not black-box code.

8.3 Combining Many "Mini-Fact" Domains into a Single Declarative Universe

Finally, because each domain is just a **Schema + Data + Aggregators** package, it's straightforward to connect them. Imagine referencing a geometry shape inside a quantum wavefunction domain, or overlaying baseball stats with economic data. As long as each domain expresses constraints purely declaratively, their aggregator fields can coexist, giving you a "bigger universe" of emergent truths with minimal friction.

In the next (and final) section, we'll summarize how the same structural approach leads to a stable, consistent representation of "truth," even across wide-ranging domains—and where we can go from here.

8.4 Addressing Querying and Retrieval

A purely declarative model is only as useful as our ability to query and retrieve the emergent facts it encodes. In practical terms, two highly accessible platforms for building CMCC models are **Baserow** and **Airtable**, each offering a JSON-based meta-model API. In such systems:

- **Schema Access**
  You can retrieve the entire "rulebook" (Schema, Data, Lookups, Aggregations, Lambdas, Constraints) in JSON form. This ensures that both human developers and programmatic agents can inspect every declared constraint or aggregator formula on demand.

- **Snapshot-Consistent Data**
  With each commit or change, the data and all derived aggregator fields remain transactionally aligned. Any valid query against that snapshot sees a fully coherent state.

- **Template-Based Generation**
  For lightweight transformations, **Handlebars**-style templates can render the JSON data into user-facing formats (reports, HTML, etc.). At larger scales, any ACID-compliant datastore (e.g., SQL Server, MySQL, PostgreSQL) can serve as the engine beneath these declarative objects, guaranteeing the same snapshot consistency.

Thus, retrieval becomes straightforward: "facts in, queries out." When you fetch data from the aggregator fields or derived states, you inherently observe the entire truth declared at that moment—no extra code to "update" or "synchronize" anything.

8.5 Implementation & Performance Considerations

When large volumes of facts arrive simultaneously—e.g., tens of thousands of `RunEvent` or `MeasurementEvent` records per second—maintaining a snapshot-consistent environment can be challenging. Two primary approaches address this at scale:

1. **ACID Transactions in Distributed SQL**
   - Modern distributed databases (e.g., CockroachDB, Yugabyte, Google Spanner) guarantee serializable transaction isolation across multiple nodes. Each aggregator is materialized (or incrementally updated) at commit time, ensuring other readers see a consistent snapshot.
   - **Trade-Off:** Strict serializability can increase commit latency, so real-time applications might batch aggregator updates or partition data (e.g., by team, game, or geometry region).
2. **Conflict Resolution in NoSQL**
   - Systems lacking built-in ACID transactions (e.g., Cassandra, Riak) often employ eventual consistency with conflict resolution or vector clocks. Here, aggregator "final values" can briefly diverge across replicas until a resolution phase merges them.
   - **Declarative Reconciliation:** The aggregator formula acts as the single source of truth—if conflicting updates are detected (e.g., two partial sums for the same scoreboard), a final merge or "last writer wins" policy can unify them.

A Minimal Flow Diagram

Below is a simplified concurrency flow for aggregator updates:

```
┌──────────────────────┐    (1)  Insert new facts (RunEvent/OutEvent)
│  Client(s) Produce    │   ──────────────────────────────────────────────┐
└──────────────────────┘                                                   │
                                                                         .    V
               ┌──────────────────────┐       ┌───────────────────────────────┐
               │  Distributed DB        │       │ Aggregator Calculation Layer      │
               │   (ACID or NoSQL)      │<───▶ │  (materialized views, etc.)      │
               └──────────────────────┘       └───────────────────────────────┘
                                                                                 │
 (2) ACID commit  ────────────────────────────────────────────────────────▶
                                                                                 │
 (3) Consistent snapshot available  ──────────────────────────────────────▶
```

1. **Clients** insert new facts in parallel.
2. The system commits changes via an atomic transaction or merges them with conflict resolution.
3. Aggregators recalculate in the same commit or in an immediate post-commit step, providing a consistent new snapshot to subsequent queries.

These well-established distributed strategies align with CMCC's requirement: every aggregator sees a coherent view of the data at commit time. The exact choice—ACID or eventual consistency—depends on the domain's performance demands and tolerance for brief inconsistency windows.

8.6 Positioning Relative to Known Approaches

It's worth noting that **logic programming** (e.g., Prolog, Datalog) and **semantic-web** frameworks (RDF, OWL) also use declarative statements of facts and rules rather than imperative code. However, typical triple-store or ontology systems do not natively support aggregations and lambda-style calculations as first-class constructs; those often require separate SPARQL queries or specialized reasoners. Meanwhile, functional programming excels at pure functions and immutability but does not inherently define a single, snapshot-consistent data store with aggregator constraints. **CMCC** merges the two models: (1) a fact-based representation of entities and relationships, and (2) built-in aggregator and lambda fields all enforced within an atomic snapshot. This synthesis, coupled with consistent transaction semantics, turns out to be remarkably powerful for representing logic across multiple domains—without "sidecar" scripts or partial updates.

The underlying philosophy of CMCC overlaps with logic/ontology frameworks such as **RDF/OWL**, **GraphQL schemas**, or classical **relational** modeling. However, most of these systems lack three critical features that CMCC insists upon:

1. **Native Aggregations**
   CMCC treats sums, counts, min/max, means, or more advanced rollups as first-class, snapshot-consistent fields. By contrast, RDF/OWL and typical knowledge-graph systems often need external "sidecar" processes (SPARQL queries, reasoner plugins) to compute or store aggregates.

2. **Lambda Functions**
   CMCC includes "L" or "Lambda Calculated Fields" for if-then or more functional logic—again stored within the same declarative structure, rather than in a separate code layer.

3. **Strict Snapshot Consistency**
   Many existing frameworks either rely on asynchronous updates or do not guarantee that all derived fields are in sync at every moment. In CMCC, every aggregator is always consistent with the data in one atomic snapshot.

Hence, CMCC does not require external scripts, triggers, or imperative "sidecars." By embedding aggregations and functional fields alongside the raw data, it keeps everything in a single, consistent knowledge lattice. Other papers in this series delve into deeper comparisons, but these are the key differentiators for now.

In large-scale knowledge-graph scenarios, the presence of aggregator fields in a single snapshot model can reduce the need for external reasoners or domain-specific triggers, thus streamlining cross-domain data flow.

8.7 Expanding Multi-Domain Integration

Throughout the paper, we noted that geometry, baseball, and quantum mechanics can co-exist under a single rule set. A more realistic illustration might blend **economic** factors (e.g., ticket revenue, city engagement) with the performance of a hometown baseball team:

- **Home-Win Economic Impact**
  Suppose we declare a "CityEconomics" entity that aggregates foot traffic, local business revenue, and intangible morale. Each "home game" event references the same aggregator fields, so when the hometown team wins multiple games in a row, we can observe a correlated rise in local economic metrics—all within the same snapshot.

- **Simple Declarative Rule**
  "IF `winStreak >= 5` THEN `CityEconomics.moraleRating = +0.1`," purely as a lambda or aggregator constraint. No imperative "updateCityMorale()" is required.

This approach can extend indefinitely. As long as each domain expresses its rules and relationships using the same five primitives (S, D, L, A, F) in a consistent snapshot environment, the data merges seamlessly—and new emergent cross-domain truths may appear.

8.7.1 Implementation Roadmap

To adopt CMCC in real-world systems, teams typically begin by migrating one well-bounded domain (e.g., geometry or scoring) into a purely declarative aggregator model, verifying that existing workflows can be replaced by aggregator-based logic. Once successful, additional domains—such as economics or advanced physics—are gradually integrated, ensuring each domain's aggregator constraints align with the larger snapshot environment. Production implementations often use layered ACID transactions, caching, and query-optimization strategies (e.g., materialized views) to maintain performance at scale.

8.8 Exploring Large-Scale or Real-World Systems

Finally, the CMCC approach scales in principle to any real-world environment, because the model only says **what** is true, not **how** to make it so. Some practical observations:

- **Implementation Freedom**
  "Magic mice in assembler code" (or any other solution) can handle the runtime logic. So long as the

aggregator formulas and constraints are satisfied at commit time, a system can be distributed, multi-threaded, or specialized for big data.

- **Best Practices Still Apply**
  You can partition or shard large tables, use caching layers, parallelize aggregator computations, and so on. From the CMCC perspective, these are purely optimizations: the end result must remain snapshot-consistent with the declared rules.

- **Physical Reality as the Runtime**
  For advanced domains like quantum mechanics or real-time scientific experiments, one might say the laws of nature "run" the system. The CMCC model then describes the data we gather (measurement events, wavefunction states) without prescribing how the physical process actually executes.

Thus, whether you have a small local dataset in Airtable or a petabyte-scale system in a global datacenter, the essential declarative logic remains the same. **CMCC** is simply the universal rulebook, not the runtime engine.

---

9. Conclusion: Structure as Truth

We set out to show how three very different domains—basic geometry, baseball, and quantum phenomena—can be modeled without a single imperative "update" call. Instead, these domains unfold purely from enumerated facts, references, and aggregator definitions, all guaranteed consistent by a snapshot-based approach.

9.1 The Strength of Fact Piling: Incoherence Becomes Impossible

Because each new fact or constraint must integrate harmoniously into an existing snapshot, the system naturally prevents contradictions or incoherent intermediate states. For geometry, you cannot have a "triangle" with four edges. For baseball, you cannot have four outs in the same half-inning. For quantum physics, you cannot measure mutually exclusive outcomes simultaneously. The principle of snapshot consistency ensures that all derived truths (angle sums, scoreboard tallies, collapsed wavefunctions) remain consistent with the entire web of declared facts.

As more facts accrue, the "cost" of introducing false or contradictory statements grows: the system will simply flag the inconsistency. This mechanism neatly inverts the typical anxiety over "edge cases" or "corner conditions" in procedural code, since each aggregator and constraint stands as an explicit guardrail for domain coherence.

9.2 Future Directions: New Domains, Larger Ecosystems, and Handling Contradictions

The declarative perspective presented here sets the stage for an impressive array of future work. A few examples include:

1. **Larger, Cross-Domain Ecosystems**
   - Combining baseball with economic modeling or linking quantum mechanical events to a geometry-based design. Because each domain's logic is declared in the same aggregator style, cross-domain queries and insights emerge naturally.
2. **Concurrency and Distributed Systems**

- Exploring how snapshot consistency scales across distributed databases. If each node commits facts and aggregations transactionally, one could maintain a globally coherent knowledge base of indefinite size.

3. **Handling Partial Inconsistencies or Contradictions**
   - Investigating how "soft constraints" or partial aggregator definitions might allow for uncertain or evolving data (common in real-world AI systems). This might involve merging multiple snapshots or identifying domains where incomplete facts must be later reconciled.
4. **Turing Completeness and Halting**
   - Although this paper did not delve into the formal completeness proofs or the halting problem, readers may reference parallel works that show how a purely declarative aggregator system can, in principle, simulate universal computation. The boundaries and limitations (e.g., Gödelian self-reference) are ripe areas for continued investigation.

Ultimately, if the Conceptual Model Completeness Conjecture (CMCC) holds as we scale up, one might imagine an increasingly universal framework in which all computable truths—be they mathematical, athletic, or physical—are captured by the same five declarative primitives in a single snapshot-consistent environment.

9.3 Key Prior Works

1. **BRCC: The Business Rule Completeness Conjecture**
   - *Citation:* BRCC_The_Business_Rule_Completeness_Conjecture.pdf, Zenodo: https://zenodo.org/records/14735965
   - *Significance:* Introduces the foundational idea that any finite business rule can be fully decomposed using five declarative primitives (S, D, L, A, F) in an ACID-compliant environment. Establishes the falsifiability challenge central to all "completeness" conjectures.
2. **BRCC-Proof: The Business Rule Completeness Conjecture (BRCC) and Its Proof Sketch**
   - *Citation:* BRCC-3page-QED-MathematicalProof.pdf, Zenodo: https://zenodo.org/records/14759299
   - *Significance:* Provides the condensed theoretical "proof sketch" demonstrating Turing-completeness within BRCC's five-primitives framework. This early work lays the groundwork for subsequent refinements (including applications to geometry, baseball, and quantum mechanics).
3. **CMCC: The Conceptual Model Completeness Conjecture**
   - *Citation:* _CMCC_The Conceptual Model Completeness Conjecture (CMCC) as a Universal Computational Framework.pdf, Zenodo: https://zenodo.org/records/14760293

Significance: Extends the BRCC principle beyond business rules to *any* computable domain—geometry, physics, sports, etc.—all within a single declarative, ACID-based model. Forms the direct theoretical backbone for "The Emergent Truth" paper's key arguments.

References & Acknowledgments

Below is a consolidated list of works cited throughout the paper, spanning foundational philosophy, mathematics, physics, and the broader context of declarative modeling.

1. **Descartes, R. (1637)**
   Discourse on the Method. Translated and reprinted in various editions.

2. **Tarski, A. (1944)**
   "The Semantic Conception of Truth and the Foundations of Semantics." *Philosophy and*

Phenomenological Research, 4(3), 341–376.

3. **Wheeler, J. A. (1990)**
 "Information, Physics, Quantum: The Search for Links." In *Complexity, Entropy, and the Physics of Information,* W. H. Zurek (Ed.), Addison-Wesley.

4. **Kant, I. (1781)**
 Critique of Pure Reason. Multiple translations and editions; originally published in German as *Kritik der reinen Vernunft.*

5. **Wolfram, S. (2002)**
 A New Kind of Science. Wolfram Media.

6. **Quine, W. V. O. (1960)**
 Word and Object. MIT Press.

7. **Everett, H. (1957)**
 "'Relative State' Formulation of Quantum Mechanics." *Reviews of Modern Physics,* 29, 454–462.

8. **Wigner, E. (1961)**
 "Remarks on the Mind-Body Question." In I. J. Good (Ed.), *The Scientist Speculates,* Heinemann.

Other relevant resources include work on logic programming, knowledge representation, and domain modeling that parallels the ideas introduced here. The author would like to thank the contributors to open-source declarative frameworks for their ongoing dedication to clarity in knowledge modeling. Special thanks also go to readers who propose new fact-based domains, since any discovered contradiction or boundary condition helps refine and test the Conceptual Model Completeness Conjecture.

Appendices

For practical examples, code, and JSON-based domain definitions (geometry, baseball, quantum walks), please visit the project's GitHub repository:
github.com/eejai42/conceptual-model-completeness-conjecture-toe-meta-model

Appendix A: Example Declarative Statements (Geometry)

Sample aggregator definitions and field constraints illustrating how "sum of angles" or "isRightTriangle" can be purely declared, with no further code.

```
{
  "id": "CMCC_ToEMM_TriangleMinimal",
  "meta-model": {
    "name": "Minimal Triangleness Demo",
    "description": "Demonstration of a polygon model that checks for 3-edge polygons,
                    right angles, etc. using only a tiny set of aggregator formulas.",
    "version": "v1.0",
    "nickname": "triangle_demo",
```

```
        "schema": {
          "entities": [
            {
              "name": "Edge",
              "description": "A simple edge record. (No advanced geometry;
                              just a placeholder.)",
              "fields": [
                {"name": "id","type": "scalar","datatype": "string","primary_key": true},
                {"name": "label","type": "scalar","datatype": "string",
                             "description": "Optional name or label for the edge."},
                {"name": "polygon_id","type": "lookup","description": "Points back to
                          which polygon this edge belongs.","target_entity": "Polygon"}
              ],
              "lookups": [],
              "aggregations": [],
              "lambdas": [],
              "constraints": []
            },
            {
              "name": "Angle",
              "description": "Represents a single angle (in degrees)
                              belonging to a polygon.",
              "fields": [
                {"name": "id","type": "scalar","datatype": "string","primary_key": true},
                {"name": "angle_degrees","type": "scalar","datatype": "float",
                         "description": "The angle measure in degrees."},
                {"name": "polygon_id","type": "lookup","description": "Which polygon
                          this angle belongs to.","target_entity": "Polygon"}
              ],
              "lookups": [],
              "aggregations": [],
              "lambdas": [],
              "constraints": []
            },
            {
              "name": "Polygon",
              "description": "A polygon with edges and angles. We’ll check if
                              it’s a triangle, if it has a right angle, etc.",
              "fields": [
                {"name": "id","type": "scalar","datatype": "string",
                              "primary_key": true},
                {"name": "label","type": "scalar","datatype": "string",
                                 "description": "Optional name for the polygon."}
              ],
              "lookups": [
```

```
        {
          "name"          : "edges"                          ,
          "target_entity" : "Edge"                           ,
          "type"          : "one_to_many"                    ,
          "join_condition": "Edge.polygon_id = this.id"      ,
          "description"   : "All edges that form this polygon."
        },
        {
          "name"          : "angles"                             ,
          "target_entity" : "Angle"                              ,
          "type"          : "one_to_many"                        ,
          "join_condition": "Angle.polygon_id = this.id"         ,
          "description"   : "All angles belonging to this polygon."
        },
        {
          "name"          : "angle_degrees"                      ,
          "target_entity" : "this"                               ,
          "type"          : "one_to_many"                        ,
          "join_condition": "this.angles.angle_degrees"          ,
          "description"   : "An array of the angles of a triangle."
        }
      ],
      "aggregations": [
        {"name": "edge_count","type": "rollup","description": "Number of edges
                  in this polygon.","formula": "COUNT(this.edges)"},
        {"name": "angle_count","type": "rollup","description": "Number of angles
                  in this polygon.","formula": "COUNT(this.angles)"},
        {"name": "largest_angle","type": "rollup","description": "The maximum
                  angle measure among angles.",
                  "formula": "MAX(this.angle_degrees)"},
        {"name": "sum_of_angles","type": "rollup","description": "Sum of all
                  angle measures in degrees.",
                  "formula": "SUM(this.angle_degrees)"},
        {"name": "is_triangle","type": "rollup","description": "True if the
                  polygon has exactly 3 edges.",
                  "formula": "EQUAL(this.edge_count, 3)"},
        {"name": "has_right_angle","type": "rollup","description": "True if
                  any angle == 90.","formula": "CONTAINS(this.angle_degrees, 90)"},
        {
          "name"       : "shape_type",
          "type"       : "rollup",
          "description": "Naive categorization based on edge_count:
                         3 => triangle, 4 => quadrilateral, else other.",
          "formula"    : "IF( EQUAL(this.edge_count,3), 'triangle',
                         IF(EQUAL(this.edge_count,4),'square','polygon') )"
```

```
•               }
•             ],
•             "lambdas": [],
•             "constraints": []
•           }
•         ],
•         "data": {"Edge": [],"Angle": [],"Polygon": []}
•       }
•     }
•   }
```

And this is the code that is derived directly from it:

```python
"""
Auto-generated Python code from your domain model.
Now with aggregator rewriting that references core_lambda_functions.
"""
import math
import numpy as np
from core_lambda_functions import COUNT, SUM, MAX, IF, CONTAINS, EQUAL

import uuid
import re

class CollectionWrapper:
    """A tiny helper so we can do something like: obj.someLookup.add(item)."""
    def __init__(self, parent_object, attr_name):
        self.parent_object = parent_object
        self.attr_name = attr_name
        if not hasattr(parent_object, '_collections'):
            parent_object._collections = {}
        if attr_name not in parent_object._collections:
            parent_object._collections[attr_name] = []

    def add(self, item):
        self.parent_object._collections[self.attr_name].append(item)

    def __iter__(self):
        return iter(self.parent_object._collections[self.attr_name])

    def __len__(self):
        return len(self.parent_object._collections[self.attr_name])

    def __getitem__(self, index):
        return self.parent_object._collections[self.attr_name][index]

# Below are aggregator stubs not yet in core_lambda_functions:
def AVG(collection):
```

```python
    """Placeholder aggregator: real logic not yet implemented."""
    # Could do: return sum(collection)/len(collection) if numeric
    return f"/* AVG not implemented: {collection} */"

def EXISTS(condition_expr):
    return f"/* EXISTS not implemented: {condition_expr} */"

def MINBY(expr):
    return f"/* MINBY not implemented: {expr} */"

def MAXBY(expr):
    return f"/* MAXBY not implemented: {expr} */"

def MODE(expr):
    return f"/* MODE not implemented: {expr} */"

def TOPN(expr):
    return f"/* TOPN not implemented: {expr} */"


# ----- Generated classes below -----

class Edge:
    """Plain data container for Edge entities."""
    def __init__(self, **kwargs):
        self.id = kwargs.get('id')
        self.label = kwargs.get('label')
        self.polygon_id = kwargs.get('polygon_id')

        # If any 'one_to_many' or 'many_to_many' lookups exist, store them as
           collection wrappers.

class Angle:
    """Plain data container for Angle entities."""
    def __init__(self, **kwargs):
        self.id = kwargs.get('id')
        self.angle_degrees = kwargs.get('angle_degrees')
        self.polygon_id = kwargs.get('polygon_id')

        # If any 'one_to_many' or 'many_to_many' lookups exist, store them as
           collection wrappers.

class Polygon:
    """Plain data container for Polygon entities."""
    def __init__(self, **kwargs):
        self.id = kwargs.get('id')
        self.label = kwargs.get('label')
```

```python
        # If any 'one_to_many' or 'many_to_many' lookups exist, store them as
           collection wrappers.
        self.edges = CollectionWrapper(self, 'edges')
        self.angles = CollectionWrapper(self, 'angles')

    @property
    def edge_count(self):
        """Number of edges in this polygon.
        Original formula: COUNT(this.edges)
        """
        return COUNT(self.edges)

    @property
    def angle_count(self):
        """Number of angles in this polygon.
        Original formula: COUNT(this.angles)
        """
        return COUNT(self.angles)

    @property
    def largest_angle(self):
        """The maximum angle measure among angles.
        Original formula: MAX(this.angle_degrees)
        """
        return MAX(self.angle_degrees)

    @property
    def sum_of_angles(self):
        """Sum of all angle measures in degrees.
        Original formula: SUM(this.angle_degrees)
        """
        return SUM(self.angle_degrees)

    @property
    def is_triangle(self):
        """True if the polygon has exactly 3 edges.
        Original formula: EQUAL(this.edge_count, 3)
        """
        return EQUAL(self.edge_count, 3)

    @property
    def has_right_angle(self):
        """True if any angle == 90.
        Original formula: CONTAINS(this.angle_degrees, 90)
        """
        return CONTAINS(self.angle_degrees, 90)
```

```
    @property
    def shape_type(self):
        """Naive categorization based on edge_count: 3 => triangle, 4 => quadrilateral,
           else other.
        Original formula: IF( EQUAL(this.edge_count,3), 'triangle',
            IF(EQUAL(this.edge_count,4),'square','polygon') )
        """
        return IF( EQUAL(self.edge_count,3), 'triangle',
IF(EQUAL(self.edge_count,4),'square','polygon') )

    # Derived properties for 'target_entity': 'this'
    @property
    def angle_degrees(self):
        """An array of the angles of a triangle."""
        return [x.angle_degrees for x in self.angles]
```

Triangle and Polygon Results

This repository demonstrates a simple SDK for working with polygons, with a focus on triangles and their properties. The code uses a straightforward object model with properties that compute dynamically based on the shape's edges and angles.

Data Model Visualization

The following shows how data builds up as we progress through creating different polygons:

Creating an Empty Polygon

`polygon = Polygon()`

Data State:


```
{
  "Edge": [],
  "Angle": [],
  "Polygon": [
    {
      "edges": [],
      "angles": [],
      "edge_count": 0,
      "angle_count": 0,
      "angle_degrees": [],
      "largest_angle": null,
      "sum_of_angles": 0,
      "is_triangle": false,
      "has_right_angle": false,
      "shape_type": "polygon"
    }
  ]
}
```

Adding First Edge and Right Angle (90°)

```
edge1 = Edge()
angle90 = Angle()
angle90.angle_degrees = 90  # Right angle
polygon.edges.add(edge1)
polygon.angles.add(angle90)
```

Data State:


```
{
  "Edges": [{}],
  "Angles": [{ "angle_degrees": 90}],
  "Polygon": [
    {
      "edges": [{}],
      "angles": [{ "angle_degrees": 90 }],
      "edge_count": 1,
      "angle_count": 1,
      "angle_degrees": [90],
      "largest_angle": 90,
      "sum_of_angles": 90,
      "is_triangle": false,
      "has_right_angle": true,
      "shape_type": "polygon"
    }
  ]
}
```


Adding Second Edge and Angle

```
edge2 = Edge()
angle53 = Angle()
angle53.angle_degrees = 53
polygon.edges.add(edge2)
polygon.angles.add(angle53)
```

Data State:


```
{
  "Edges": [{}, {}],
  "Angles": [{ "angle_degrees": 90}, { "angle_degrees": 53}],
  "Polygons": [{
      "edges": [{}, {}],
      "angles": [{ "angle_degrees": 90 }, { "angle_degrees": 53 }],
      "edge_count": 2,
      "angle_count": 2,
      "angle_degrees": [90, 53],
      "largest_angle": 90,
      "sum_of_angles": 143,
      "is_triangle": false,
      "has_right_angle": true,
      "shape_type": "polygon"
    }
  ]
}
```

Creating a Triangle - Adding Third Edge and Angle

```
edge3 = Edge()
angle37 = Angle()
angle37.angle_degrees = 37
polygon.edges.add(edge3)
polygon.angles.add(angle37)
```

Data State:


```
{
  "Edges": [{}, {}, {}],
  "Angles": [{ "angle_degrees": 90}, { "angle_degrees": 53}, { "angle_degrees": 37}],
  "Polygons": [{
      "edges": [{}, {}, {}],
      "angles": [{ "angle_degrees": 90 }, { "angle_degrees": 53 }, { "angle_degrees": 37 }]}],
      "edge_count": 3,
      "angle_count": 3,
      "angle_degrees": [90, 53, 37],
      "largest_angle": 90,
      "sum_of_angles": 180,
      "is_triangle": true,
      "has_right_angle": true,
      "shape_type": "triangle"
    }
  ]
}
```


Changing the Right Angle to 91° (No Longer Right)

```
# Modify the existing angle
angle90.angle_degrees = 91
```

Data State:


```
{
  "Edges": [{}, {}, {}],
  "Angles": [{ "angle_degrees": 91}, { "angle_degrees": 53}, { "angle_degrees": 37}],
  "Polygons": [{
      "edges": [{}, {}, {}],
      "angles": [{ "angle_degrees": 91 }, { "angle_degrees": 53 }, { "angle_degrees": 37 }]}],
      "edge_count": 3,
      "angle_count": 3,
      "angle_degrees": [91, 53, 37],
      "largest_angle": 91,
      "sum_of_angles": 180,
      "is_triangle": true,
      "has_right_angle": false,
      "shape_type": "triangle"
    }
  ]
}
```


Making a Square - Adding Fourth Edge and Angle

```
edge4 = Edge()
angle90b = Angle()
angle90b.angle_degrees = 90
```

```
polygon.edges.add(edge4)
polygon.angles.add(angle90b)
```

Data State:


```
  "Edges": [{}, {}, {}],
  "Angles": [{ "angle_degrees": 91}, { "angle_degrees": 53}, { "angle_degrees": 37}],
  "Polygons": [{
      "edges": [{}, {}, {}, {}],
      "angles": [{ "angle_degrees": 90 }, { "angle_degrees": 90 },
                 { "angle_degrees": 90 }, { "angle_degrees": 90 }]}],
      "edge_count": 4,
      "angle_count": 4,
      "angle_degrees": [90, 90, 90, 90],
      "largest_angle": 90,
      "sum_of_angles": 360,
      "is_triangle": false,
      "has_right_angle": true,
      "shape_type": "quadrilateral"
    }
  ]
}
```


Key Features Demonstrated

1. **Dynamic Property Calculation:**
   - Edge and angle counts update automatically
   - Shape type changes based on edge count
   - Right angle detection
   - Sum of angles calculation
2. **Triangle Properties:**
   - A polygon is classified as a triangle when it has exactly 3 edges
   - Right triangle detection when any angle equals 90 degrees
   - Sum of angles should be 180 degrees (or close to it with rounding)
3. **Quadrilateral Properties:**
   - Identified when a polygon has 4 edges
   - Sum of angles should be 360 degrees for a proper quadrilateral

Appendix B: Declarative Structure of Baseball Rules

Detailed aggregator formulas for events (OutEvent, RunEvent) and advanced sabermetrics (OPS, ERA, fielding percentage) in purely data-driven form.

Team

- **Fields**
  - `id` (PK)
  - `teamName`
  - `league_id` (lookup to a League)
- **Lookups**
  - `roster` → all Players on this team
- **Aggregations** *(examples)*
  - `wins` = COUNT(all Games where `winnerId = this.id`)
  - `losses` = COUNT(all Games where `loserId = this.id`)

- `winPercentage` = `wins / (wins + losses)` if any games played
- `rosterSize` = COUNT(roster)
- `totalTeamRuns` = SUM(of all runs scored by this team)

Player

- **Fields**
  - `id` (PK), `fullName`, `battingHand` (L/R/S), `throwingHand` (L/R)
- **Lookups**
  - belongs to one `Team` (via `team_id`)
- **Aggregations** *(examples)*
  - `careerAtBats` = COUNT(AtBat where `batterId = this.id`)
  - `careerHits` = COUNT(AtBat with outcomes like SINGLE/DOUBLE/etc.)
  - `careerBattingAverage` = `careerHits / careerAtBats` (if atBats>0)
  - `ops` = `onBasePercentage + sluggingPercentage`

Game

- **Fields**
  - `id` (PK), `homeTeamId`, `awayTeamId`, `status` (e.g. IN_PROGRESS/FINAL)
- **Lookups**
  - `innings` → collection of Inning records for this game
- **Aggregations** *(examples)*
  - `runsHome` = SUM(of runs in half-innings where `offensiveTeamId=homeTeamId`)
  - `runsAway` = SUM(of runs in half-innings where `offensiveTeamId=awayTeamId`)
  - `winnerId` = if final & runs differ, whichever team is higher.
  - `loserId` = symmetric aggregator.

Inning / InningHalf

- **Inning**: references `gameId`, `inningNumber`, typically has a top/bottom half.

- **InningHalf**: references `offensiveTeamId`, `defensiveTeamId`; purely declarative “outs” and “runsScored” come from events:
  - `outs` = COUNT(OutEvent where `inningHalfId=this.id`)
  - `runsScored` = SUM(RunEvent where `inningHalfId=this.id` → `runCount`)
  - `isComplete` = true if outs≥3 **or** walk-off condition triggered

AtBat

- **Fields**
  - `id`, `inningHalfId`, `batterId`, `pitcherId`, `result`, `rbi`
- **Aggregations**
  - `pitchCountInAtBat` = COUNT(Pitch where `atBatId = this.id`)
  - `batterHasStruckOut` = (strikeCount >= 3)
  - `wasWalk` = (result='WALK')

Pitch

- **Fields**
  - `id`, `atBatId`, `pitchResult` (BALL, STRIKE, etc.), `pitchVelocity`
- **Aggregations** *(example)*
  - `isStrike` = true if pitchResult ∈ {CALLED_STRIKE, SWINGING_STRIKE}

OutEvent

- **Description**

- A record that an out occurred. Ties to an `inningHalfId` and optionally an `atBatId`.
- **No "incrementOuts()"**
  - Simply create `OutEvent` → `InningHalf.outs` aggregator rises accordingly.

RunEvent

- **Description**
  - A record that one or more runs scored, referencing the half-inning (and possibly an at-bat).
- **No "scoreRun()"**
  - Summed by aggregator → `InningHalf.runsScored`.

Appendix C: A simple Cross Domain Synergy

C.1 Introduction

This appendix shows how **CMCC** unifies disparate domains—in this case, **Baseball** and **Economics**—under the same snapshot-consistent rulebook. By treating the two domains as sets of **Schema, Data, Lookups, Aggregations, and Lambda** fields, we demonstrate how an economic model can directly reference baseball facts (like a team's wins or attendance) to derive business or revenue metrics—all **without** writing imperative code.

C.2 Example: "TeamEconomics" Referencing the Baseball Domain

Entities from the Baseball Domain

- **Team**: Has fields (e.g., `id`, `teamName`) and aggregations like `winPercentage`, `currentWinStreak`, etc.
- **Game**: Tracks which team won or lost, attendance, etc.

New Entity: "TeamEconomics"
Let's define a minimal "TeamEconomics" entity that directly pulls aggregator values from the Baseball `Team` entity. This new entity can compute, for instance, ticket revenue as a function of the team's win percentage and attendance data.

json
```
{
  "id": "CMCC_ToEMM_Economics_Baseball_Example",
  "meta-model": {
    "name": "Economics Extension to Baseball Domain",
    "description": "Illustrates how an economic aggregator (e.g., ticket revenue) can depend on a
baseball team's performance metrics.",
    "schema": {
      "entities": [
        {
          "name": "TeamEconomics",
          "description": "Captures revenue and morale for a baseball team in a given city,
referencing the baseball domain's 'Team' entity.",
          "fields": [
            {
              "name": "id",
              "type": "scalar",
              "datatype": "string",
              "primary_key": true,
              "description": "Unique identifier for this economics record."
            },
```

```
          {
            "name": "team_id",
            "type": "lookup",
            "target_entity": "Team",
            "foreign_key": true,
            "description": "Which baseball team this economics record corresponds to."
          },
          {
            "name": "cityName",
            "type": "scalar",
            "datatype": "string",
            "description": "The city or market where this team plays."
          }
        ],
        "lookups": [],
        "aggregations": [
          {
            "name": "averageAttendance",
            "type": "rollup",
            "description": "Pulls the baseball domain's aggregator for average attendance across
home games if that is tracked. Conceptual example.",
            "formula": "LOOKUP(TeamAttendance where teamId = this.team_id =>
averageHomeAttendance)"
          },
          {
            "name": "winPct",
            "type": "rollup",
            "description": "Directly references the baseball domain’s aggregator 'winPercentage'
on the Team entity.",
            "formula": "this.team_id.winPercentage"
          },
          {
            "name": "projectedTicketRevenue",
            "type": "rollup",
            "description": "Estimates ticket revenue from averageAttendance * some function of
the team's winPercentage.",
            "formula": "IF (winPct > 0.5) THEN (averageAttendance * 75) ELSE (averageAttendance *
50)"
          },
          {
            "name": "cityMorale",
            "type": "rollup",
            "description": "Rough measure of local morale, which rises if the team’s
currentWinStreak is high. Conceptual aggregator referencing a baseball aggregator.",
            "formula": "IF (this.team_id.currentWinStreak >= 5) THEN 1.2 ELSE 1.0"
          }
        ],
        "lambdas": [
          {
            "name": "simulatePriceChange",
            "parameters": ["ticketPriceFactor"],
```

```
            "description": "A purely declarative constraint that sets an updated formula for
'projectedTicketRevenue' if the city decides to raise/lower ticket prices.",
            "formula": "this.projectedTicketRevenue = (this.projectedTicketRevenue *
ticketPriceFactor)"
          }
        ],
        "constraints": []
      }
    ],
    "data": {
      "TeamEconomics": [
        {
          "id": "ECON_001",
          "team_id": "TEAM_NYY",
          "cityName": "New York"
        },
        {
          "id": "ECON_002",
          "team_id": "TEAM_BOS",
          "cityName": "Boston"
        }
      ]
    }
  }
 }
}
```

C.3 Explanation of the Key Fields and Formulas

1. **`team_id`**
   A **lookup** that directly references the **Baseball** domain's `Team` entity. The aggregator formulas in `TeamEconomics` can thus pull any baseball field or aggregator exposed by `Team`.

2. **`averageAttendance`**
   Illustrates a hypothetical aggregator that looks up a "TeamAttendance" measure (or any other baseball domain aggregator). It might sum or average attendance from all of the team's home games.

3. **`winPct`**
   Directly references the baseball domain's aggregator `Team.winPercentage`. This is a single expression—no imperative "fetch or recalc" step required.

4. **`projectedTicketRevenue`**
   Combines the above (`averageAttendance` and `winPct`) into a single formula. If the team's `winPct` is above 0.5, we assume an increased ticket revenue (75 currency units per attendee vs. 50 otherwise).

5. **`cityMorale`**
   Similarly ties into `currentWinStreak` on the baseball `Team`, increasing city morale if the streak is high. No "callFunctionToUpdateMorale()` ever runs; the aggregator auto-updates whenever `currentWinStreak` changes.

6. **Lambda Example** – `simulatePriceChange`
   While not strictly necessary, it demonstrates how a purely declarative formula can express “update constraints” such as changing ticket pricing. Internally, it’s still a statement about “what is true if price factor changes,” rather than an imperative directive.

C.4 Snapshot Consistency Across Domains

- **No Special “Integration Step”**
  In a typical procedural system, you might need a separate script or API call to sync baseball performance data with an economics module. Under CMCC, once `Team.winPercentage` changes, everything referencing it (like `projectedTicketRevenue`) automatically sees the updated value in the next committed snapshot.

- **Scalability**
  This approach scales to any number of domains, as each domain’s schema and aggregations can read or reference aggregator outputs from the others. For instance, you could add “MerchandiseSales” or “SponsorDeals” entities that also incorporate `winPercentage` or `averageAttendance` with no new custom function calls required.

- **Clarity and Maintainability**
  Economic constraints and formulas now live in the same declarative environment as baseball’s runs, outs, or attendance. If you want to adjust how revenue is computed (e.g., weigh attendance more heavily), you simply revise the aggregator formula in `TeamEconomics`.

C.5 Takeaway

This short example demonstrates the **cross-domain synergy** that emerges naturally from CMCC. A single snapshot-consistent system can carry **baseball** logic (teams, runs, aggregator-driven outcomes) while simultaneously expressing **economic** logic (ticket revenue, city morale, pricing) in the same rulebook—no manual bridging code or “glue scripts” needed. As the team’s performance changes, so do the economic metrics, purely by virtue of shared aggregator fields and lookups.