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

Introduction: The Emergence of Truth

1.1 Purpose of This Paper

This paper is **not** a how-to guide for building domains in CMCC, nor is it a deep proof of Turing-completeness. Instead, it's a **narrative demonstration** of how purely declarative modeling—absent any imperative "update" steps—allows geometric theorems, baseball scores, and quantum measurement collapses to arise naturally from the same **universal** set of primitives.

In other words, **we won't** be walking through exhaustive code examples or domain-specific minutiae; those live in the accompanying GitHub repositories. Our focus here is **why** these five primitives (Schema, Data, Lookups, Aggregations, and Lambda Calculated Fields) can unify knowledge modeling across wildly diverse domains—and how "theorems" like Pythagoras or "three outs end an inning" are by-products of enumerated constraints.

1.2 Motivation: From "I think, therefore I am" to Geometry

Philosophical discussions on truth often start with René Descartes' foundational statement: "I think, therefore I am." By acknowledging **something** rather than **nothing**, we introduce the concept of existence—a ground-level distinction that opens the door to a cascade of richer facts. Notice there's no imperative step required; it's purely a statement of what *is*.

- As soon as you say, "There is something," you can label that "something" with a **1**, and its absence with a **0**.
- From bits (0 and 1), you can represent any number by enumerating them as sequences of bits.
- Once you have numbers, you can define coordinates, distances, and angles.
- A triangle then "emerges" if you declare three connected points.

At every step, no one calls a function like `CreateTriangle()`. One simply states, "These three points connect," and that constraint alone yields a triangular shape in the conceptual space. This approach leads directly to geometry's most famous result—the Pythagorean theorem—arising from constraints rather than commands.

1.3 Overview of the Incremental Approach

Our journey starts with the simplest binary statements—"something" vs. "nothing"—and shows how a few additional facts yield lines, angles, triangles, and finally the Pythagorean theorem. **But then**, we show how that same approach underpins two other domains you might never associate with geometry:

1. **Baseball** – A supposedly "procedural" sport with seemingly stepwise rules ("increment runs," "increment outs"). In the declarative model, these increments vanish; we simply declare that a run occurred or an out was made. Aggregator formulas handle the rest.

2. **Quantum Walk** – Where wavefunction superposition and measurement similarly reduce to enumerated facts (amplitudes, measurement events). No `collapseWavefunction()` is needed: aggregator constraints do the collapsing once a measurement data point is declared.

By comparing geometry, baseball, and quantum mechanics under one narrative, we spotlight the **syntax-free** nature of CMCC: You or your system enumerate facts (in JSON, for instance), and aggregator fields establish the relationships that produce "magical" results like $a^2 + b^2 = c^2$ or wavefunction collapse—all without specialized DSLs.

1.4 Avoiding Theorems & Proofs: Letting Facts Speak for Themselves

We don't **prove** classical results in the usual sense; we simply list the constraints that make them inevitable. For instance:

- We don't "prove" Pythagoras. We note that **once** you've declared three edges, an angle of 90°, and that these edges belong to the same polygon, $a^2 + b^2 = c^2$ is forced by the aggregator definitions.
- We don't code `incrementRuns()` in baseball; we define events (RunEvent, OutEvent) and aggregator formulas that yield the total runs or outs.
- We don't call `collapseWavefunction()` in quantum. A measurement aggregator triggers that outcome once relevant data appears (e.g., a "MeasurementEvent").

Hence, **the logic emerges** from enumerated constraints and aggregator fields. That is the essence of the **Conceptual Model Completeness Conjecture (CMCC)**: with just **five** building blocks (S, D, L, A, F), any domain's truths can be fully captured—no specialized imperative code required.

---

2. Starting at Zero: Something versus Nothing

We begin with the most elementary assertion of all existence: the statement "I think, therefore I am" implies at least one indisputable reality—there is **something** rather than **nothing**. By declaring the presence of something, we introduce the foundational concept of binary distinction:

- We define **0** to represent the absence of existence (nothingness).
- We define **1** to represent the presence of something.

From this simple binary distinction, we establish a foundational truth:

> "There is something (1), as opposed to nothing (0)."

2.1 Binary as the Declarative Foundation

Building upon these definitions, we now assert new facts:

- We call a collection of 0s and 1s a **binary representation**.
- This structured arrangement of bits enables us to represent numerical concepts clearly and unambiguously.

Thus emerges another inevitable truth:

> "Numbers are declaratively represented using binary."

2.2 Constructing a Conceptual Space: Numbers to Coordinates

Given that we can represent numbers declaratively, we naturally extend our concept by asserting:

> "Numbers can represent coordinates in a conceptual space."

This simple assertion allows us to represent precise positions, and we now add:

> "Two numbers together define a point."

2.3 Emergence of Geometric Structures

From two points, a new truth logically emerges **without** imperative steps:

> “Two points together define a line.”

With the existence of lines, certain emergent truths automatically appear:

- “Lines have lengths and directions, derived implicitly from their points.”
- “Lines connecting distinct points form measurable lengths and angles.”

By declaring yet another point, a profound structural shift occurs:

> “Three connected points define a triangle.”

Once triangles exist, certain properties become inevitable truths:

- “A triangle has exactly three internal angles.”
- “The sum of these angles is always exactly 180 degrees.”

2.4 The Emergence of the Right Triangle

When we specify further conditions, additional truths naturally follow. For instance, we declare:

> “If one internal angle is exactly 90 degrees, the triangle is a right triangle.”

This introduces a new inevitable classification and a new layer of relational structure:

- “The side opposite this 90-degree angle is called the hypotenuse.”
- “We commonly label the hypotenuse as ‘c’.”

These assertions now bring us tantalizingly close to a profound truth.

2.5 Inevitable Emergence of the Pythagorean Theorem

With the above declarative constraints in place, the following relationship becomes inevitable:

- “For any right triangle, the relationship between the sides a, b, and hypotenuse c will always satisfy the relationship $a^2 + b^2 = c^2$.”

Notably, this is not stated as a theorem or a rule imposed externally. Instead, it emerges organically as a logical consequence from the previously declared facts and constraints—purely declarative truths that made any contradictory outcome impossible. The Pythagorean theorem is therefore not simply a rule or instruction; it is an emergent fact woven inherently into the structure of geometric existence.

---

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

We concluded Section 2 by noting that binary-encoded numbers can represent arbitrary coordinates, thus forming the bedrock of a “conceptual space.” In this section, we explore how straightforward facts about such coordinates give rise to lines, angles, and polygons, all without any imperative “drawShape()” or “computeAngle()” calls. Instead, once we declare that certain points exist—and how they relate to one another—the entire world of Euclidean geometry unfolds inevitably.

3.1 Representing Numbers in Binary—Mapping to Coordinates

The path from bits to geometry begins by *interpreting* our previously defined binary numbers as **coordinates** in a Cartesian plane. That is, any pair (x,y)(x, y)(x,y) can be stored as two binary strings—one for xxx, one for yyy. From a declarative viewpoint:

1. **Numbers as Data**: We store each integer or floating-point value as a series of bits (0s and 1s).
2. **Coordinate as a Pair**: We declare a coordinate by grouping two numeric values, e.g. (x1,y1)(x_1, y_1)(x1,y1).
3. **No Computation**: We never instruct the system to "create a coordinate." We merely **state** that "this pair of numbers forms a point," and, logically, it now exists within our conceptual universe.

When consistently applied, these steps enable us to place points anywhere on a conceptual plane—no domain-specific geometry engine required.

3.2 Defining a "Point" from Two Numbers

Once we've said "a coordinate is two numbers," it becomes natural to **declare** an entity called a "Point." For instance:

- **Point**: An entity holding two numeric fields, `x` and `y`.

In a purely declarative approach, the moment we have `(x, y)` as factual data, we can call it a `Point`. This is not a procedure, but rather a direct relationship: "These two numeric values define a location in 2D space." The aggregator formulas in our schema will later help us derive distances, slopes, and angles, but for now, all we do is label the record as a `Point`.

3.3 Two Points Make a Line: Emergent Notions of Length & Angle

With *two* such points in place, new emergent properties arise automatically:

1. **Line Existence**: Declare that two points are "connected," and you have a line segment by definition.
2. **Lengths & Directions**: Because each point has numeric (x,y)(x, y)(x,y) coordinates, the **length** and **direction** of the segment come "for free" via aggregator formulas (e.g., a distance aggregator computing (x2−x1)2+(y2−y1)2\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}(x2−x1)2+(y2−y1)2).
3. **Angles**: If a second line references one of the same points, an **angle** aggregator can measure the turn between those two segments, also without any imperative geometry calls.

In short, "two points form a line" is not a coded function—it's a fact. Once stated, length and directional properties cannot be avoided; they simply exist by virtue of coordinate definitions and aggregator calculations.

3.4 Adding a Third Point: Triangleness Appears

Stepping from two points (one line) to three points (three lines) leads us to the **triangle**, arguably the most fundamental shape in Euclidean geometry. We **declare**:

> "Three points—each pair connected by a line segment—compose a polygon with three edges."

No function `CreateTriangle()` is invoked; the aggregator formulas that track "number of edges" or "internal angles" will reveal that we indeed have a *three*-edged shape. From here:

- **Angles**: The system can identify each internal angle by referencing line segments.
- **Edges**: A count aggregator yields “edge_count = 3,” labeling it a triangle.
- **Sum of Angles**: Another aggregator can sum these angles, inevitably reaching 180° in a Euclidean context.

This is how **triangleness** emerges from data, rather than from any direct command.

3.5 Fourth Point: Quadrilaterals & Expanding the Shape Universe

Adding a **fourth** point, or more, extends the same pattern:

1. **Quadrilaterals**: If four lines form a closed loop, you have a polygon with four edges (e.g., a quadrilateral).
2. **Classification**: Aggregator-driven logic can further classify shapes by their angles (rectangles, parallelograms, etc.)—again *without* imperatively switching from “triangle mode” to “quadrilateral mode.”
3. **Beyond 2D**: Similarly, you can define additional coordinates (like a zzz-value) for 3D shapes, or further fields for higher-dimensional objects.

Thus, from *binary numbers* to *3-edge polygons* to more elaborate shapes, the entire structural story is the same: once you declare enough data (points and their connections), aggregator formulas take over, revealing the geometry that was implicit all along.

---

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*.

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.

---

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: "Wavefunction 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 wavefunction plus a measurement outcome.

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."

5.3 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 wavefunction, 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."

6.1 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 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()`.

6.3 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.

---

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.

---

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 Positioning Relative to Known Approaches

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.

8.6 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 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 Reflection on “I Think, Therefore I Am” as the Only Imperative Statement

At the start, we recalled Descartes’s simplest declaration of existence: “I think, therefore I am.” This is arguably the sole imperative “statement” in the entire conceptual universe—merely a recognition that there is *something* rather than *nothing*. From that one acknowledgment, we cascaded into bits (0s and 1s), numerical coordinates, geometry, and beyond. This underscores a key philosophical point: **once we accept the existence of data (“I am”), all subsequent details can be declaratively inferred** by specifying relationships and constraints. No further “Do this. Now do that.” is required.

9.2 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.

9.3 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.

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.

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

Appendices

- **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):
    """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')

        # 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]
```

Putting it into practice:

Triangle and Polygon Demo

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.

Core Entities (Abbreviated)

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
CopyEdit

```
{
  "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.