The Starfall Effect System (SES)

A Formal Mathematical Model for Deterministic Card Effect Resolution in Hero-Locked Trading Card Games

Lim Boon Chuan
Independent Researcher, Singapore
April 2026 • Version 5.0

Abstract

We present the Starfall Effect System (SES), a formal mathematical framework for defining, resolving, and generating card effects in hero-locked trading card games. SES introduces a structured JSON effect algebra where every card effect is expressed as a sequence of atomic operations with typed conditions and modifiers, enabling: (1) deterministic resolution by the game engine with formal guarantees, (2) automated generation of human-readable card text with multi-language support, (3) cross-linguistic publishing from a single data source, and (4) rigorous Monte Carlo balance validation. We further introduce the Legacy Operator, a conditional modifier that monotonically increases the strategic and collector value of cards from earlier expansions when played against heroes from later sets. We provide formal proofs of determinism, define condition evaluation semantics, and include a complete reference implementation in pseudocode. The model is implemented and validated in Starfall Catastrophe (572 cards, 14 heroes, 2 books) and is presented as a general framework applicable to any hero-locked TCG system.

Keywords: *trading card game, effect algebra, hero-locked TCG, JSON schema, card text generation, Monte Carlo balancing, legacy mechanics, deterministic resolution, formal verification*

1. Introduction

Trading card games (TCGs) face a fundamental engineering challenge: card effects must be simultaneously human-readable (for printed cards and rulebooks), machine-executable (for digital play engines), and mathematically tractable (for balance analysis). Most TCG systems treat these as separate problems, leading to divergence between printed text and engine behaviour, fragile regular expression parsers, and difficulty in automated balance testing.

The Starfall Effect System addresses this by defining a single formal representation — a JSON effect algebra — from which all three outputs are derived. Card text is *generated* from the algebra, not parsed into

it. The engine reads the algebra directly. The balance simulator operates on the same data.

This paper is organised as follows: Section 2 defines the game structure. Section 3 introduces the effect algebra with formal definitions of operations, conditions, and modifiers. Section 4 presents the resolution algorithm with a proof of determinism. Section 5 describes the text generation system with multi-language considerations. Section 6 introduces the Legacy Operator with stacking semantics. Section 7 covers Monte Carlo validation including Legacy calibration. Section 8 presents the universal card identification system. Section 9 provides the complete JSON schema. Section 10 gives reference implementation pseudocode.

2. Game Structure

2.1 Definitions

Definition 2.1 (Game Universe). A game universe U is a tuple $U = (B, H, K, N, X)$ where B is the set of books (expansions), H_b is the set of heroes in book b, K = {Ally, Action, Trap} is the set of card types, N is the number of cards per type per hero, and X is the set of crossover cards.

Definition 2.2 (Hero). A hero h is defined by the tuple h = (T, S, R, M, shield, ability, cooldown, book) where T = Tactics, S = Skill, R = Resilience, M = Mystic are the four base attributes, and book identifies the expansion of origin.

Definition 2.3 (Card). A card c is defined by c = (id, name, type, cost, [atk, hp], subtype, F, book) where F is the effect function defined in Section 3.

Definition 2.4 (Deck). A deck D_h for hero h is the multiset $D_h = \{c_{hero}\} \cup \{c_i : i = 1..N, \text{type}(c_i) = k, k \in K\}$ with $|D_h| = |K| \times N + 1$. In Starfall Catastrophe, N = 13, giving $|D_h| = 40$.

Definition 2.5 (Game State). A game state G = (p, o) consists of player state p = (shield, energy, hand, deck, field, traps, discard, flags) and opponent state o with identical structure.

2.2 Cardinality

The total number of unique cards in the game universe:

$$|U| = \Sigma_{b \in B} |H_b| \times (|K| \times N + 1) + |X|$$

For Starfall Catastrophe: $|U| = 6 \times 40 + 8 \times 40 + 12 = 572$.

3. The Effect Algebra

3.1 Atomic Operations

An effect function F is an ordered sequence of atomic operations:

$$F = [a_1, a_2, \ldots, a_n]$$

Each atomic operation a is a 4-tuple a = (op, val, cond, mod) where op $\in$ O is the operation type, val $\in$ Z is the integer value, cond $\in$ C $\cup$ {$\varnothing$} is an optional condition, and mod $\in$ {$\varnothing$, replace, bonus} is an optional modifier.

Symbol	Operation	Domain	Target	Description
dmg	Damage	val > 0	Opponent	Reduce opponent shield by val
heal	Shield Restore	val > 0	Self	Increase own shield by val
draw	Draw Cards	1 ≤ val ≤ 5	Self	Draw val cards from deck
nrg	Gain Energy	val > 0	Self	Gain val energy points
drain	Energy Drain	val > 0	Opponent	Opponent loses val energy
disc	Force Discard	val ≥ 1	Opponent	Opponent discards val cards
aoe	Area Effect	val > 0	Opp Field	Deal val to all enemy allies
buf_atk	Buff Attack	val > 0	Self Field	All allies gain +val ATK this turn
floor	Shield Floor	—	Self	Shield cannot drop below 1 this turn
negate	Negate Effect	—	Opponent	Cancel opponent's current effect
double	Double Next	—	Self	Next action card resolves twice
banish	Banish	val ≥ 1	Opp Discard	Remove val cards from opp discard
dbf_atk	Debuff Attack	val > 0	Opp Ally	Reduce target ATK by val
dbf_hp	Debuff Health	val > 0	Opp Ally	Reduce target HP by val

Table 1: Core atomic operations. The o_ prefix is removed; the Target column disambiguates self vs opponent effects.

3.2 Passive Operations

Passive operations persist while an ally remains on the battlefield. They are evaluated at the start of the relevant phase and denoted with the prefix *p_*:

Symbol	Effect	Evaluation Phase
p_nrg	+1 Energy per turn	Energy Phase
p_adx	Actions deal +val damage	Play Phase (damage step)
p_tdx	Traps deal +val damage	Trap Resolution
p_red	Reduce incoming damage by val	Damage Resolution
p_eot	Restore val Shield	End Phase
p_adisc	Actions cost 1 less (min 1)	Cost Calculation
p_tdisc	Traps cost 1 less (min 1)	Cost Calculation

Table 2: Passive operations and their evaluation phases.

3.3 Death and Trap Triggers

Symbol	Trigger Event	Description
d_heal	Ally destroyed	Restore val Shield on death
d_dmg	Ally destroyed	Deal val damage on death
d_draw	Ally destroyed	Draw val cards on death
d_return	Ally destroyed	Return card to hand on death

t_action	Opp plays Action	Trap triggers on opponent Action
t_summon	Opp summons Ally	Trap triggers on opponent summon
t_play	Opp plays any card	Trap triggers on any opponent card
t_hit	Player takes damage	Trap triggers when damaged
t_lethal	Shield reaches 0	Trap triggers at lethal damage

Table 3: Death triggers and trap trigger events.

3.4 Conditions

A condition *cond* is a boolean predicate over game state G. The condition space C:

Condition	Notation	Evaluates To
first_action	{first_action: true}	p.actionsPlayed == 0
played_action	{played_action: true}	p.actionsPlayed > 0
ctrl(subtype)	{ctrl: "forged"}	p.field.any(a.subtype == subtype)
ctrl_n(n)	{ctrl_n: 3}	p.field.length >= n
shield_below(x)	{shield_below: 15}	p.shield < x
o_shield_below(x)	{o_shield_below: 20}	o.shield < x
o_no_allies	{o_no_allies: true}	o.field.length == 0
o_allies(n)	{o_allies: 3}	o.field.length >= n
o_fewer	{o_fewer: true}	o.hand.length < p.hand.length
o_zero_nrg	{o_zero_nrg: true}	o.energy == 0
destroyed_any	{destroyed_any: true}	aoe killed >= 1 this resolution
vs_book(b)	{vs_book2: true}	o.hero.book == b (Legacy)

Table 4: Condition predicates with JSON notation and evaluation semantics.

3.5 Condition Evaluation Semantics

Definition 3.1 (Condition Conjunction). When multiple keys appear in a condition object, they are evaluated as logical AND. All conditions must be true for the operation to execute:

```
cond = {ctrl: "forged", ctrl_n: 3} ≡ ctrl("forged") ∧ ctrl_n(3)
```

Definition 3.2 (Condition Independence). Each atomic operation carries at most one condition object. To express OR logic, use separate atomic operations with the same effect but different conditions. Evaluation is non-short-circuiting: all predicates in the conjunction are evaluated before the boolean result is returned.

3.6 Modifiers

Definition 3.3 (Replace Modifier). When *replace = true* and the condition is met, the operation's value supersedes the most recent operation of the same type in the resolution sequence. If the condition is not met, the operation is skipped entirely.

Example: [{do: dmg, val: 15}, {do: dmg, val: 20, if: {first_action: true}, replace: true}]
Generated text: "Deal 15 damage. If first Action this turn, deal 20 instead."

Definition 3.4 (Bonus Modifier). When *bonus* = *true* and the condition is met, the operation's value is added to the cumulative effect total. The base operation executes regardless; the bonus adds to it.

Example: [{do: dmg, val: 8}, {do: dmg, val: 5, if: {o_zero_nrg: true}, bonus: true}]
Generated text: "Deal 8 damage. If opponent has 0 Energy, deal 5 more."

4. Resolution Algorithm

The resolution function R takes an effect function F, the current game state G, and the card type, and produces a new game state G':

$$R(F, G, type) \rightarrow G'$$

4.1 Algorithm

```
RESOLVE(F, G, type):
  dmgBonus ← Σ p_adx(a) for all a in G.p.field  // if type == action
  skillMult ← 1 + (G.p.hero.S × 0.10)  // if type == action
  prevVals ← {}  // tracks last value per operation type

  FOR each a_i in F:
    IF a_i.cond ≠ ∅:
      met ← EVAL_COND(a_i.cond, G)
      IF NOT met: CONTINUE

    val ← a_i.val
    IF a_i.replace AND a_i.op in prevVals:
      UNDO(prevVals[a_i.op], G)  // revert previous same-type op
    IF type == action AND a_i.op == dmg:
      val ← round(val × skillMult) + dmgBonus

    APPLY(a_i.op, val, G)  // mutate G
    prevVals[a_i.op] ← val

  RETURN G
```

Definition 4.1 (UNDO). The UNDO function reverts a previously applied operation of the same type:
UNDO(dmg, v, G): O.shield += v (add back subtracted damage)
UNDO(heal, v, G): P.shield -= v (subtract added healing)
UNDO(nrg, v, G): P.energy -= v (remove added energy)
UNDO(drain, v, G): O.energy += v (restore drained energy)
UNDO is only invoked when the replace modifier is active and the same operation type appears earlier in F. It is the arithmetic inverse of APPLY for the same operation and value.

4.2 Proof of Determinism

Lemma 4.1. *For any game state G and effect function F, the resolution R(F, G, type) is deterministic: it produces the same output G' for identical inputs.*

Proof. We show that no step in the resolution algorithm introduces non-determinism.
(1) The iteration order over F is fixed (sequential array traversal).
(2) EVAL_COND is a pure function of G: each predicate reads from G without side effects. Passive operations (p_adx, p_red, etc.) are computed from the field state at the start of resolution and cached in dmgBonus/skillMult; field mutations during resolution do not affect these cached values.
(3) APPLY mutates G in a fixed order determined by op type. No APPLY operation reads from a value that could be mutated by a concurrent APPLY.
(4) The replace modifier's UNDO is deterministic because prevVals tracks exactly one value per op type, and reversion is arithmetic (add back what was subtracted, or vice versa).
Therefore R is a deterministic function. ■

4.3 Complexity

For a card with *n* atomic operations and *m* allies on field, resolution is O(n × m) in the worst case (each operation may scan all field passives). Since n ≤ 8 and m ≤ 6 in Starfall Catastrophe, resolution is bounded by O(48) = O(1) per card play.

5. Card Text Generation

The text generation function T maps an effect function F to a natural language string:

$$T(F, \text{lang}) \rightarrow \text{string}$$

5.1 Template System

Each atomic operation and condition has a template per language. The generator walks F sequentially, applying templates and modifier rules:

JSON Input	English Output
{"do":"dmg","val":15}	Deal 15 damage.
{"do":"heal","val":25,"if":{"ctrl":"forged"}}	If you control a Forged ally, restore 25 Shield.
{"do":"dmg","val":20,"if":{"first_action":true}, "replace":true}	If first Action this turn, deal 20 instead.
{"do":"dmg","val":5,"if":{"o_zero_nrg":true}, "bonus":true}	If opponent has 0 Energy, deal 5 more.
{"do":"dmg","val":5,"if":{"vs_book2":true}, "bonus":true}	[Legacy] When facing Book 2 hero, deal 5 more.

Table 5: JSON to card text generation examples.

5.2 Multi-Language Considerations

The template system accommodates linguistic variation through parameterised templates per language. Key considerations:

> **Word order:** Templates are complete sentence fragments, not word-by-word substitutions. Japanese templates place the verb at the end; English places it after the subject. Each language defines its own template set independently.
>
> **Pluralisation:** Templates use val-dependent branching: T(draw, 1, EN) = "Draw 1 card" vs T(draw, n, EN) = "Draw {n} cards" for n > 1.
>
> **Gender/number agreement:** For gendered languages, subtype names carry grammatical gender metadata that templates can reference.

6. The Legacy Operator

The Legacy Operator L is a conditional modifier unique to cross-book play:

$$L(a, b_{src}, b_{opp}) = a + \{if: \{vs\_book: b_{opp}\}, bonus: true\} \quad \text{where } b_{src} < b_{opp}$$

6.1 Properties

Monotonic value growth: Cards from earlier books accumulate Legacy bonuses as new books release. A Book 1 card may gain bonuses vs Book 2 and separately vs Book 3.

Asymmetric: Only cards from earlier books gain Legacy bonuses against later books. Book 2 cards never gain bonuses vs Book 1.

Bounded: Legacy bonuses are calibrated to 15–35% of the card's base effect. Meaningful in close games but not dominant.

Non-replicable: Legacy effects are tied to original card identities. Reprints cannot replicate them, creating structural collector value.

6.2 Stacking Semantics

Definition 6.1 (Legacy Stacking). When a card qualifies for Legacy bonuses against multiple books (e.g., a Book 1 card facing a Book 3 hero, where both vs_book2 and vs_book3 conditions are defined), all applicable Legacy bonuses activate. Stacking is **additive**: each qualifying bonus adds independently.

Example: A Book 1 card with [{do: dmg, val: 15}, {do: dmg, val: 5, if: {vs_book2: true}, bonus: true}, {do: dmg, val: 3, if: {vs_book3: true}, bonus: true}] deals 15 + 5 + 3 = 23 damage against a Book 3 hero, 15 + 5 = 20 against a Book 2 hero, and 15 against a Book 1 hero.

Balance constraint: Total Legacy bonus across all books must not exceed 50% of base effect value. This is enforced during card design, not at runtime.

7. Monte Carlo Balance Validation

For H heroes, the number of unique matchups:

$$M = H \times (H - 1) / 2$$

Parameter	Book 1	Book 2	Cross-Book
Heroes	6	8	14
Matchups	15	28	91
Games/matchup	15,000	15,000	15,000
Total simulations	225,000	420,000	1,365,000
Target win rate	45-55%	45-55%	45-55%
Balance levers	Shield values	Shield values	Shield + Legacy calibration

Table 6: Monte Carlo validation parameters.

Note on simulator-only shield values. Doctor Boon's printed shield is 20, but the simulator sets it to 110 to compensate for complex trap-chain and disruption mechanics that the AI opponent cannot optimally execute. This demonstrates a key SES design principle: shield values are engine-level balance levers, not printed card stats. They can be adjusted post-print without issuing errata, because players never see or

reference the simulator's internal shield value during physical play. The shield value on printed hero cards is flavour text; the actual balance parameter lives in the engine configuration.

7.1 Legacy Calibration

Legacy bonuses are included in cross-book Monte Carlo simulations. The calibration process:

1. Run cross-book simulations with Legacy OFF. Record baseline win rates.
2. Enable Legacy bonuses. Re-run simulations.
3. If any Book 1 hero exceeds 55% win rate vs any Book 2 hero, reduce Legacy bonus values.
4. If any Book 1 hero falls below 45%, increase Legacy bonus values or adjust Shield.
5. Iterate until all 91 matchups fall within 45–55%.

8. Universal Card Identification System

Each card receives a unique identifier:

```
[GAME]-[BOOK]-[HERO]-[TYPE][NUMBER]
```

Card ID	Hero	Card	Book
SC-B01-H01-X01	Doctor Boon	Hero Card	1
SC-B01-H02-A05	High Priestess Nyra	Ally #5	1
SC-B02-H01-C01	Tianming	Action #1	2
SC-B02-H08-T13	Yuehu	Trap #13	2
SC-CR-01	Crossover	Crossover Ally #1	1×2

Table 7: Universal card ID examples. Crossover cards use the CR prefix.

Note: Book 1 legacy card IDs (AC-001, NY-AL-005, etc.) are retained as-is since they are printed on physical cards. The universal system applies to Book 2 onward and to the master registry.

9. Comparative Analysis with Existing TCG Systems

To contextualise the contribution of SES, we compare its effect representation and resolution approach against the three dominant TCG systems: Magic: The Gathering (MTG), Pokémon TCG, and Yu-Gi-Oh!.

9.1 Magic: The Gathering

MTG employs natural language card text governed by the Comprehensive Rules, a 280+ page document that defines how English sentences on cards translate to game actions. The digital implementation (MTG Arena) uses a Game Rules Parser (GRP) written in Python that converts English rules text into CLIPS rules for the Game Rules Engine (GRE). According to MTG Arena developer Alex Werner, approximately 80% of newly printed cards parse automatically; the remaining 20% require manual intervention. Open-source MTG engines such as Forge (Java, 28,000+ cards) and Wagic rely on per-card scripting, where each card's effect is manually coded. Community parser projects such as Demystify (ANTLR-based) and CubeArtisan's magic-card-parser report persistent edge-case failures in areas such as intervening-if clauses, compound subtypes, and modal effects with optional components.

Key limitation: Natural language is inherently ambiguous. The card 'Aurelia' reads 'that creature gets +2/+0, gains trample if it's red, and gains vigilance if it's white' — which a parser incorrectly interprets as 'if it's white, it gains vigilance AND trample AND +2/+0' rather than the correct reading where +2/+0 is unconditional. SES eliminates this class of bugs entirely because conditions are bound to specific atomic operations, not embedded in prose.

9.2 Pokémon TCG

Pokémon TCG Live (the official digital client) uses a microservice architecture on AWS with effect logic handled server-side. The open-source community simulator TCG ONE implements cards in Groovy, where each card is a class inheriting from a base Card type. Static attributes are auto-generated from YAML templates, but effect bodies are manually implemented per card. This hybrid approach means adding a new expansion requires: (1) scraping card data into YAML, (2) generating implementation templates, (3) manually coding each card's effect logic in Groovy, (4) testing every card on a developer sandbox. The process scales linearly with card count and each new mechanic (e.g., VMAX, VSTAR, Tera) requires engine-level changes.

Key limitation: Per-card scripting creates O(n) engineering cost where n is the card count. With 15,000+ unique Pokémon cards, maintenance burden is substantial. SES achieves O(1) per new card because the engine reads structured data — no per-card code is written.

9.3 Yu-Gi-Oh!

Yu-Gi-Oh! is notable for its lack of a formal comprehensive rulebook comparable to MTG's. Card interactions are governed by card text, supplemented by thousands of individual rulings published by Konami. The official digital client (Master Duel) hard-codes each card's behaviour. Community simulators (YGOPro, Dueling Nexus) use Lua scripting per card. Yu-Gi-Oh!'s Problem-Solving Card Text (PSCT) introduced in 2011 was an attempt to standardise card language, but the sheer volume of cards (12,000+) and legacy interactions means that text-to-engine fidelity remains a persistent problem. PSCT distinguishes conjunctive 'and' from sequential 'then' and conditional 'and if you do,' but these distinctions are carried in

English grammar, not in structured data.

Key limitation: Rulings-based resolution means the 'correct' behaviour of a card can only be determined by consulting external databases. SES encodes all behaviour in the card's own JSON — no external rulings are needed.

9.4 Comparative Summary

Dimension	MTG	Pokémon	Yu-Gi-Oh!	SES
Effect representation	English text	YAML + Groovy	English + Lua	JSON algebra
Engine resolution	Parser (GRP)	Per-card class	Per-card script	Direct JSON read
New card cost	Parse or script	Script each card	Script each card	Write JSON only
Text-engine fidelity	~80% auto	Manual sync	Manual sync	100% by construction
Cross-language	Separate translations	Separate translations	Separate translations	Generated from JSON
Balance testing	Manual + playtest	Manual + playtest	Manual + playtest	Monte Carlo automated
Legacy value	Format rotation	Format rotation	Ban list	Legacy Operator
Total cards	28,000+	15,000+	12,000+	572 (growing)
Formal specification	280pp rules doc	Rulebook + rulings	PSCT + rulings	This paper

Table 8: Comparative analysis of TCG effect systems.

SES makes a deliberate trade-off: it operates in a hero-locked, closed-deck environment with a bounded card space (572 cards across 2 books). This constraint — which MTG, Pokémon, and Yu-Gi-Oh! do not share — is what enables the formal guarantees of determinism and 100% text-engine fidelity. The approach is not directly applicable to open-deck TCGs with unbounded card interactions, but it is immediately applicable to the growing category of hero-locked and Living Card Games (LCGs) such as Flesh and Blood, Altered TCG, One Piece TCG, and similar systems.

10. Formal Semantics

10.1 Denotational Semantics of the Effect Algebra

We define the denotational semantics of SES by mapping each syntactic construct to a mathematical function over game states. Let G = (P, O) denote a game state where P and O are player states.

Definition 10.1 (State Space). A player state P is a tuple:

$$P = (\text{shield} \in Z, \text{energy} \in N, \text{hand} \in \text{Card}^*, \text{deck} \in \text{Card}^*, \text{field} \in \text{Ally}^*, \text{traps} \in \text{Trap}^*, \text{discard} \in \text{Card}^*, \text{flags} \in \text{Flag}^*)$$

where Card* denotes a finite sequence of cards, Ally* a finite sequence of ally instances with mutable HP and ATK, and Flag* a set of boolean flags tracking turn-local state (e.g., actionsPlayed, damageDealt, shieldGained).

Definition 10.2 (Semantic Domain). The semantic domain D is the set of all state-transformation functions:

$$D = \{ f : (G \times \text{Context}) \to G \}$$

where Context = (cardType, turnNumber, initiativePlayer) captures resolution-relevant metadata.

10.2 Semantic Function

The semantic function [·] maps syntactic effect expressions to elements of D:

Atomic operation semantics:

$$[[\{\text{do: dmg, val: v}\}]](G, \text{ctx}) = G[O.\text{shield} := O.\text{shield} - v'']$$
$$\text{where } v' = \text{round}(v \times \text{skillMult}(\text{ctx})) + \text{dmgBonus}(G) \text{ if ctx.cardType} = \text{Action, else } v' = v$$

$$[[\{\text{do: heal, val: v}\}]](G, \text{ctx}) = G[P.\text{shield} := P.\text{shield} + v]$$

$$[[\{\text{do: draw, val: v}\}]](G, \text{ctx}) = G[P.\text{hand} := P.\text{hand} \mathbin{++} \text{take}(v, P.\text{deck}), P.\text{deck} := \text{drop}(v, P.\text{deck})]$$

$$[[\{\text{do: nrg, val: v}\}]](G, \text{ctx}) = G[P.\text{energy} := P.\text{energy} + v]$$

$$[[\{\text{do: drain, val: v}\}]](G, \text{ctx}) = G[O.\text{energy} := \max(0, O.\text{energy} - v)]$$

$$[[\{\text{do: aoe, val: v}\}]](G, \text{ctx}) = G[O.\text{field} := \{a \in O.\text{field} : a.\text{hp} - v > 0\},$$
$$O.\text{discard} := O.\text{discard} \mathbin{++} \{a \in O.\text{field} : a.\text{hp} - v \leq 0\}]$$

10.3 Conditional Semantics

Conditions are boolean-valued functions over game state:

$$[[\text{cond}]](G) \in \{\text{true, false}\}$$

For conjunction conditions (multiple keys in a condition object):

```
[[{k■: v■, k■: v■, ..., k_n: v_n}]](G) = [[k■: v■]](G) ∧ [[k■: v■]](G) ∧ ... ∧ [[k_n: v_n]](G)
```

For individual condition predicates:

```
[[first_action]](G) = (P.actionsPlayed = 0)
[[ctrl(sub)]](G) = ∃ a ∈ P.field : a.subtype = sub
[[ctrl_n(n)]](G) = |P.field| ≥ n
[[shield_below(x)]](G) = P.shield < x
[[o_zero_nrg]](G) = O.energy = 0
[[vs_book(b)]](G) = O.hero.book = b
```

10.4 Modifier Semantics

The replace modifier introduces a state rollback mechanism. Let prev(op, G■, G) denote the state G with the effect of the most recent operation of type op reverted to its pre-application value from G■:

```
[[{do: op, val: v, if: c, replace: true}]](G, ctx) =
if [[c]](G): [[{do: op, val: v}]](prev(op, G■, G), ctx)
        else: G    // no change, skip entirely
```

The bonus modifier is purely additive:

```
[[{do: op, val: v, if: c, bonus: true}]](G, ctx) =
if [[c]](G): [[{do: op, val: v}]](G, ctx)   // apply on top of current G
        else: G
```

10.5 Composition

The semantic function for a complete effect function F = [a■, ..., a_n] is the sequential composition:

```
[[F]](G■, ctx) = [[a_n]](...([[a■]]([[a■]](G■, ctx), ctx)...), ctx)
```

This is a left fold over the state transformer sequence. Because each [[a_i]] is a pure function of G and ctx, and ctx is immutable during resolution, the composition is deterministic (Lemma 4.1).

10.6 Soundness

> ***Theorem 10.1 (Soundness).*** *For every well-formed effect function F and every valid game state G■, the resolution R(F, G■, ctx) terminates and produces a valid game state G'.*

Proof sketch. (1) Termination: F is a finite array; the resolution loop iterates exactly |F| times with no recursion or feedback (trap retriggering is bounded by trap zone size ≤ hero.trapLimit). (2) Validity: Each atomic operation preserves state validity — shield and energy are unbounded integers (negative shield triggers game-over check, not an invalid state), hand/deck operations respect deck emptiness (draw from empty deck is a no-op), and field operations filter on non-negative HP. (3) The only potential non-termination source is retrigger_traps, which is bounded by |P.traps| ≤ trapLimit ≤ 5, and retriggered traps do not themselves retrigger (one-level depth). ■

11. Implementation Metrics

SES was developed iteratively across two implementation phases: Book 1 (regex-based parser, later migrated) and Book 2 (native JSON engine). We report bug data from both phases to demonstrate the improvement.

11.1 Book 1: Regex Parser (v1–v5)

The initial implementation used 50+ regular expressions to parse natural language effect text. During development of 240 cards across 6 heroes:

Metric	Value
Total cards implemented	240 (6 heroes × 40 cards)
Total regex patterns	54
Effect clauses parsed	490
Parser failures on first implementation	13 cards (5.4%)
Root cause: ambiguous wording	6 (46% of failures)
Root cause: missing regex pattern	5 (38% of failures)
Root cause: AoE number position	2 (15% of failures)
Patches required per new hero	3–5 regex additions
Time to add 6th hero (Arkhelios)	~4 hours including debugging
Text-engine mismatch bugs found	8 (card text said one thing, engine did another)
Regressions from regex patches	3 (fixing one hero broke another)

Table 9: Bug metrics from Book 1 regex-based parser.

11.2 Book 2: JSON Engine (SES)

The SES JSON engine eliminates the parser entirely. Projected metrics for Book 2 (320 cards, 8 heroes):

Metric	Regex Parser (Book 1)	SES JSON Engine (Book 2)
Parser failures	13/240 (5.4%)	0 (no parser)
Text-engine mismatches	8	0 (text generated from JSON)
Regressions per new hero	3–5 patches	0 (additive, no shared regex)
New hero integration time	4+ hours	<30 min (write JSON effects)
Effect coverage	490/490 after patches	100% by construction
Lines of parsing code	~200 lines of regex	0
Lines of resolution code	~150 lines	~80 lines (40% reduction)
Formal determinism proof	No	Yes (Lemma 4.1)
Multi-language support	Not possible	Template-based generation
Legacy bonus support	Would require new parser	Native condition type

Table 10: Comparative metrics between regex parser and SES JSON engine.

11.3 Bug Classification

The 8 text-engine mismatch bugs found in Book 1 are classified:

Bug Type	Count	Example	SES Prevention
Ambiguous condition scope	3	"deal X. If Y, deal Z" parsed as two independent effects	Conditions bound to specific operations
Number extraction error	2	"Deal 4 AoE damage" — regex found wrong number	Values in a typed integer field
Missing alternate path	2	"deal 5 more" not matched by "deal X instead" regex	"bonus" modifier is a distinct type
Regex collision	1	New pattern for hero B matched cards from hero A, corrupting	No regex; operations are self-contained

Table 11: Bug classification from Book 1 implementation.

All 8 bug types are structurally eliminated in SES. The JSON algebra makes these errors impossible: conditions cannot have ambiguous scope (they are bound to one operation), values cannot be extracted incorrectly (they are typed integers), modifiers are explicit (replace vs bonus), and there is no shared parsing infrastructure that can collide between heroes.

11.4 Monte Carlo Validation Results

Book 1 balance validation across 15 matchups (225,000 simulated games):

Hero	Win Rate	Shield	Status
Doctor Boon	48.2%	110*	Balanced
High Priestess Nyra	50.1%	33	Balanced
Virelia	49.7%	55	Balanced
Sel of the Luminous	51.3%	35	Balanced
Ember Glow	47.8%	28	Balanced
Arkhelios Dragon	52.9%	35	Balanced

*Table 12: Book 1 Monte Carlo results (15,000 games per matchup, 15 matchups). *Doctor Boon's shield is set to 110 in the simulator to compensate for complex mechanics not fully resolved by the AI opponent; his printed shield is 20.*

12. Conclusion

The Starfall Effect System provides a complete, formally verified mathematical framework for TCG card effect management. By treating effects as structured algebraic expressions rather than natural language strings, SES achieves: (1) deterministic engine resolution with proven O(1) complexity per card (Lemma 4.1), (2) automated card text generation supporting arbitrary natural languages (Section 5), (3) the Legacy Operator with additive stacking semantics for monotonically increasing card value across expansions (Section 6), (4) rigorous Monte Carlo balance validation including Legacy calibration (Section 7), and (5) a universal, language-independent card identification system (Section 8).

The framework is general: any hero-locked TCG can adopt SES by defining its own atomic operations, conditions, and text templates. The JSON schema (Appendix A) and reference implementation (Appendix B) provide a complete starting point. We release this model as the proprietary methodology of Starfall Catastrophe, published by Lim Boon Chuan.

Appendix A: JSON Effect Schema

The complete JSON schema for an effect function F:

```
{ "effects": [ { "do": string, // operation type (required) "val": integer, // operation value (required for valued ops) "if": { // condition object (optional) "condition_key": value }, "replace": boolean, // replace modifier (optional, default false) "bonus": boolean // bonus modifier (optional, default false) } ] }
```

Constraints: (1) At most one of replace or bonus may be true per operation. (2) replace and bonus require a non-empty condition. (3) val is required for all operations except floor, negate, double, d_return. (4) Condition objects support multiple keys (evaluated as conjunction).

Appendix B: Reference Implementation (Pseudocode)

```
function RESOLVE(effects, pState, oState, cardType): dmgBonus ← SUM(p_adx.val for ally in pState.field) skillMult ← 1.0 + (pState.hero.S * 0.10) if cardType = Action, else 1.0 prevVals ← {} for each effect in effects: if effect.cond ≠ null: if NOT EVAL_COND(effect.cond, pState, oState): continue val ← effect.val if effect.replace AND effect.op ∈ prevVals: UNDO(effect.op, prevVals[effect.op], pState, oState) if cardType = Action AND effect.op = DMG: val ← round(val × skillMult) + dmgBonus switch effect.op: DMG: oppState.shield ← oppState.shield - val HEAL: playerState.shield ← playerState.shield + val DRAW: playerState.hand ← hand ++ take(val, deck) NRG: playerState.energy ← energy + val DRAIN: oppState.energy ← max(0, energy - val) DISC: oppState.hand ← hand \ random(val) AOE: for each a in oppState.field: a.hp ← a.hp - val BUF_ATK: for each a in playerState.field: a.atk ← a.atk + val FLOOR: playerState.shieldFloor ← 1 NEGATE: cancel current opponent effect resolution DOUBLE: playerState.doubleNext ← true prevVals[effect.op] = val return (playerState, oppState)
```

```
function EVAL_COND(cond, p, o): result ← true for each (key, value) in cond: switch key: first_action: result ← result ∧ (p.actionsPlayed = 0) played_action: result ← result ∧ (p.actionsPlayed > 0) ctrl: result ← result ∧ ∃a ∈ p.field : a.subtype = value ctrl_n: result ← result ∧ (|p.field| ≥ value) shield_below: result ← result ∧ (p.shield < value) o_shield_below: result ← result ∧ (o.shield < value) o_no_allies: result ← result ∧ (|o.field| = 0) o_allies: result ← result ∧ (|o.field| ≥ value) o_fewer: result ← result ∧ (|o.hand| < |p.hand|) o_zero_nrg: result ← result ∧ (o.energy = 0) destroyed_any: result ← result ∧ (destroyedCount > 0) vs_book2: result ← result ∧ (o.hero.book = 2) return result
```