Formal Verification of a Token Sale Launchpad: A Compositional Approach in Dafny

Evgeny Ukhanov Aurora Labs

August 2025

Formal Verification of a Token Sale Launchpad: A Compositional Approach in Dafny

Abstract

The proliferation of decentralized financial (DeFi) systems and smart contracts has underscored the critical need for software correctness. Bugs in such systems can lead to catastrophic financial losses. Formal verification offers a path to achieving mathematical certainty about software behavior [1]. This paper presents the formal verification of the core logic for a token sale launchpad, implemented and proven correct using the Dafny programming language and verification system. We detail a compositional, bottom-up verification strategy, beginning with the proof of fundamental non-linear integer arithmetic properties, and building upon them to verify complex business logic, including asset conversion, time-based discounts, and capped-sale refund mechanics. The principal contributions are the formal proofs of critical safety and lifecycle properties. Most notably, we prove that **refunds in a capped sale can never exceed the user's original deposit amount**, and that the precision loss in round-trip financial calculations is strictly bounded. Furthermore, we verify the complete lifecycle logic, including user withdrawals under various sale mechanics and the correctness of post-sale token allocation, vesting, and claiming. This work serves as a comprehensive case study in applying rigorous verification techniques to build high-assurance financial software [2].

1. Introduction

The domain of financial software, particularly in the context of blockchain and smart contracts, operates under a unique and unforgiving paradigm: deployed code is often immutable, and flaws can be exploited for immediate and irreversible financial gain [2], [3]. Traditional software testing, while essential, is inherently incomplete as it can only validate a finite set of execution paths. Formal verification addresses this limitation by using mathematical logic to prove properties about a program's behavior across *all* possible inputs that satisfy its preconditions [4], [5].

This paper focuses on the formal verification of a token sale launchpad contract. The core challenge lies in reasoning about complex interactions between multiple components: price-based asset conversions, application of percentage-based bonuses (discounts), and state transitions governed by sale mechanics (e.g., fixed-price vs. price discovery), all while handling the subtleties of integer arithmetic.

Our tool of choice is **Dafny**, a verification-aware programming language that integrates specification and implementation [6]. Dafny allows programmers to annotate their code with formal contracts, such as preconditions (`requires`), postconditions (`ensures`), and loop invariants (`invariant`). These annotations, along with the program code, are translated by the Dafny verifier into logical formulas, which are then dispatched to an automated Satisfiability Modulo Theories (SMT) solver, typically Z3 [7]. If the solver can prove all formulas, the program is deemed correct with respect to its specification.

The primary objective of this work is to construct a fully verified model of the launchpad's core logic. We demonstrate how a carefully layered architecture enables the verification of a complex system by decomposing the proof effort into manageable, reusable components. We will present the key modules, the mathematical properties they guarantee, and the overarching safety lemmas that emerge from their composition.

2. System Architecture and Verification Strategy

The verification effort is structured around a **compositional, bottom-up approach**, which is crucial for managing complexity [8], [9]. The system is decomposed into a hierarchy of modules, where each higher-level module relies on the proven correctness of the modules below it. This isolates reasoning and makes the overall verification problem tractable.

The architecture consists of the following layers:

1. `MathLemmas:` The foundational layer. It provides proofs for fundamental, non-trivial properties of non-linear integer arithmetic (multiplication and division), which are not natively understood by SMT solvers.
2. `AssetCalculations & Discounts:` The business logic primitives layer. These modules define the core financial calculations (asset conversion, discount application) and use lemmas from `MathLemmas` to prove their essential properties, such as monotonicity and round-trip safety.
3. `Config, Investments:` The data modeling layer. These modules define the primary data structures of the system, including the main `Config` datatype which encapsulates all sale parameters and rules.
4. `Deposit:` The workflow specification layer. This module composes the primitives from lower layers to define the complete, pure specification of a user deposit, including complex refund logic.
5. `Deposit, Withdraw, Claim, Distribution:` The workflow specification layer. These modules compose primitives from lower layers to define the complete, pure specifications for all user and administrative interactions, including deposits with refund logic, withdrawals, post-sale token claims with vesting, and stakeholder distributions.
6. `Launchpad:` The top-level state machine. This module defines the complete contract state and models all lifecycle transitions by orchestrating the verified workflows from the layer below.

A key pattern employed throughout the codebase is the **Specification-Implementation Separation** [10]. For most critical operations, a `function` ending in `...Spec` defines the pure mathematical contract. This allows us to reason about the system's logic at an abstract, mathematical level.

3. Foundational Layer: Verification of Non-Linear Integer Arithmetic

Reasoning about the multiplication and division of integers is a well-known challenge in automated verification [11], [12]. SMT solvers are highly effective for linear integer arithmetic, but non-linear properties often require explicit proof guidance. The `MathLemmas` module provides this guidance by establishing a set of trusted, reusable axioms for the rest of the system.

The core of financial calculations in this system is scaling a value x by a rational factor y/k, implemented using integer arithmetic as $(x \cdot y)/k$. The following key lemmas were proven from first principles:

- **Monotonicity with Favorable Scaling (`Lemma_MulDivGreater_From_Scratch`):** This lemma proves that if the scaling factor is greater than or equal to 1, the result is no less than the original amount.
  - **Property 1.** $\forall x, y, k \in \mathbb{N}$ where $x > 0$, $k > 0$, and $y \geq k$

$$\frac{x \cdot y}{k} \geq x$$

    This is crucial for proving that conversions at a stable or favorable price do not result in a loss of principal.

- **Strict Monotonicity with Highly Favorable Scaling (`Lemma_MulDivStrictlyGreater_From_Scratch`):** Due to integer division truncation, $y > k$ is insufficient to guarantee $(x \cdot y)/k > x$. This lemma establishes a stronger precondition to ensure a strict increase.
  - **Property 2.** $\forall x, y, k \in \mathbb{N}$ where $x > 0$, $k > 0$, and $y \geq 2k$:

$$\frac{x \cdot y}{k} > x$$

    This is used to prove that a significantly favorable price or a large bonus yields a tangible gain.

- **Round-trip Truncation and Bounded Loss (`Lemma_DivMul_Bounds`):** This lemma formalizes the fundamental property of integer division: information may be lost, but this loss is strictly bounded.

– **Property 3.** $\forall x, y \in \mathbb{N}$ where $y > 0$:

$$\left(\frac{x}{y}\right) \cdot y \leq x \quad \wedge \quad x - \left(\frac{x}{y}\right) \cdot y < y$$

This property is the cornerstone for proving the safety of round-trip calculations. It not only guarantees that a reverse operation cannot create value, but also establishes a precise upper bound on the precision loss, which cannot exceed the value of the divisor `y`.

These foundational lemmas abstract away the complexities of integer arithmetic, allowing higher-level modules to reason about calculations in terms of simple inequalities.

4. Core Business Logic Verification

Building upon the `MathLemmas` foundation, we verify the core business logic components.

4.1. Asset Conversion (`AssetCalculations`)

This module defines the logic for converting a base `amount` into assets based on a price fraction `saleToken / depositToken`. The specification is: `CalculateAssetsSpec(amount, dT, sT) := (amount * sT) / dT`

The module provides lemmas that instantiate the generic mathematical properties for this specific context. For instance, `Lemma_CalculateAssets_IsGreaterOrEqual` proves that `CalculateAssetsSpec(...) >= amount` if `sT >= dT`, by directly invoking `Lemma_MulDivGreater_From_Scratch`.

A critical property for refund safety is the **round-trip inequality with bounded loss**, proven in `Lemma_AssetsRevert_RoundTrip_bounds` [13]. It states that converting an amount to assets and then back cannot result in a gain, and furthermore, that any loss due to truncation is strictly bounded.

- **Property 4 (Asset Conversion Round-Trip Safety and Bounded Loss).** Let $Assets(w) := \text{CalculateAssetsSpec}(w, dT, sT)$ and $Revert(a) := \text{CalculateAssetsRevertSpec}(a, dT, sT)$. Then for $w > 0$:

$$\text{Assets}(w) > 0 \implies \text{Revert}(\text{Assets}(w)) \leq w$$

Moreover, the lemma proves a stronger property: the scaled difference between the original and reverted amounts will never exceed the sum of the price fraction's terms:

$$(w - \text{Revert}(\text{Assets}(w))) \cdot sT < dT + sT$$

This guarantee is crucial as it proves that financial loss from rounding errors is predictable and has a hard ceiling.

4.2. Time-Based Discounts (`Discounts`)

This module implements percentage-based bonuses. It uses fixed-point arithmetic with a `MULTIPLIER` of 10000 to represent percentages with four decimal places. It also verifies a critical business rule: discount periods must not overlap.

- **Property 5 (Discount Non-Overlap).** For a sequence of discounts D, the following predicate holds:

$$\forall i, j.(0 \leq i < j < |D|) \implies (D_i.\text{endDate} \leq D_j.\text{startDate} \vee D_j.\text{endDate} \leq D_i.\text{startDate})$$

Dafny successfully proves that this property implies the uniqueness of any active discount at a given time (`Lemma_UniqueActiveDiscount`), which is essential for ensuring that deposit calculations are deterministic and unambiguous [14]. Similar to asset conversions, the module also proves the round-trip safety for applying and reverting a discount (`Lemma_WeightOriginal_RoundTrip_lte`).

5. Top-Level Specification and State Machine Verification

The verified components are composed at the top layers to model the complete system behavior.

5.1. The Deposit Workflow and Refund Safety (`Deposit` module)

This module specifies the end-to-end logic for a user deposit. The main function, `DepositSpec`, branches based on the sale mechanic. The most complex case is `DepositFixedPriceSpec`, which handles deposits into a sale with a hard cap (`saleAmount`). If a deposit would cause the total sold tokens to exceed this cap, a partial refund must be calculated.

The paramount safety property for this entire system is ensuring that this refund never exceeds the user's initial deposit. This is formally stated and proven in `Lemma_RefundIsSafe`.

- **Property 6 (Ultimate Refund Safety).** For a valid configuration and any deposit `amount`, the calculated `refund` adheres to the following inequality:

$$\text{refund} \leq \text{amount}$$

The proof of this high-level property is a testament to the compositional strategy. It is not proven from first principles but by orchestrating a chain of previously-proven, and now stronger, lemmas:

1. `Lemma_CalculateAssetsRevertSpec_Monotonic` is used to show that the reverted value of the *excess* assets is less than or equal to the reverted value of the *total* assets.
2. `Lemma_AssetsRevert_RoundTrip_bounds` proves not only that the reverted value of the total assets is less than or equal to the user's initial weighted amount (`<= w`), but also that any precision loss from this round-trip conversion is strictly bounded.
3. `Lemma_CalculateOriginalAmountSpec_Monotonic` shows that reverting the discount on a smaller amount yields a smaller result.
4. `Lemma_WeightOriginal_RoundTrip_bounds` proves that the final original amount is less than or equal to the user's initial deposit amount, and more powerfully, that this round-trip operation can result in a loss of at most one minimal unit.

By chaining these proven inequalities, Dafny confirms that `refund <= amount`, providing a mathematical guarantee against a critical class of financial bugs. This guarantee is now built upon a foundation that provides even stronger assurances about bounded precision loss at each step of the calculation.

5.2. The Withdrawal Workflow (`Withdraw` module)

The `Withdraw` module provides the formal specification for users to retrieve their funds under specific circumstances, such as a failed sale or during the `PriceDiscovery` phase. The logic is bifurcated based on the sale mechanic:

- **Fixed-Price Withdrawals (`WithdrawFixedPriceSpec`):** In a failed sale, this models an "all-or-nothing" withdrawal. The user's entire investment is returned, and their corresponding weight is removed from the `totalSoldTokens`.
- **Price-Discovery Withdrawals (`WithdrawPriceDiscoverySpec`):** This models a partial or full withdrawal where the user's contribution is re-evaluated. The specification guarantees that `totalSoldTokens` is correctly reduced by the precise difference in the user's weight, ensuring the integrity of the final price calculation.

The verification of this module ensures that state changes related to withdrawals are handled safely, preventing accounting errors.

5.3. Token Claim and Vesting Logic (`Claim` module)

The `Claim` module formalizes the post-sale logic for users to claim their purchased tokens. Its verification provides mathematical guarantees about the correctness of token allocation and vesting schedules. Key verified components include:

- **User Allocation (`UserAllocationSpec`):** This function specifies the total tokens a user is entitled to based on their final `weight` and the total `totalSoldTokens`. The lemma `Lemma_UserAllocationSpec` proves its core mathematical properties, ensuring allocations are fair and predictable.
- **Vesting Calculation (`CalculateVestingSpec`):** This function models a standard vesting schedule with a cliff and linear release. The core safety property, proven in `Lemma_CalculateVestingSpec_Monotonic`, guarantees that the vested amount never decreases as time moves forward.
- **Available to Claim (`AvailableForClaimSpec`):** This function composes the allocation and vesting logic to determine the exact amount a user can claim at a specific time.

The verification of this module is critical for ensuring that the final distribution of tokens strictly adheres to the sale's rules and vesting commitments.

5.4. Post-Sale Distribution (`Distribution` module)

The `Distribution` module specifies the administrative function of distributing tokens to project stakeholders (e.g., the team, partners, and the solver) after a successful sale. The core function, `GetFilteredDistributionsSpec`, formally defines the logic for identifying which stakeholders are eligible for the next distribution round by filtering out those who have already received their tokens. The verification ensures this process is deterministic and complete, preventing accounts from being either skipped or paid multiple times.

5.5. The Contract State Machine (`Launchpad` module)

The `Launchpad` module represents the apex of the verification hierarchy. It defines the global state of the contract within the immutable `AuroraLaunchpadContract` datatype and models all of its lifecycle transitions [15]. The state representation is comprehensive, encapsulating not only the core financial tallies but also tracking the lifecycle of post-sale events, such as stakeholder distributions (`distributedAccounts`) and individual vesting claims (`individualVestingClaimed`).

The `GetStatus` function provides a pure, verifiable definition of the contract's status (e.g., `NotStarted`, `Ongoing`, `Success`), which serves as the basis for enforcing state-dependent business rules. This module includes critical lemmas that prove the logical integrity of the state machine itself:

- **Mutual Exclusion (`Lemma_StatusIsMutuallyExclusive`):** The contract cannot be in two different states simultaneously.
- **Temporal Progression (`Lemma_StatusTimeMovesForward`):** The contract progresses logically through its lifecycle as time advances.
- **Terminal States (`Lemma_StatusFinalStatesAreTerminal`):** Once a final state (`Success`, `Failed`, `Locked`) is reached, it cannot be exited.

The core of this module is the set of functions that model the contract's state transitions. Each transition is a pure function that takes the previous state Σ and returns a new state Σ', thereby providing a complete and auditable specification of the contract's behavior. Key verified transitions include:

- **`DepositSpec`:** Models the full state transition upon a user deposit. It enforces that deposits are only possible during the `Ongoing` state and delegates all complex financial logic (including refund calculations) to the pre-verified `Deposit.DepositSpec` function.
- **`WithdrawSpec`:** Specifies the logic for users to withdraw their funds. Its preconditions ensure this action is only permissible in valid states (e.g., `Failed`, `Locked`, or during an `Ongoing` price discovery sale). The function orchestrates the state change by invoking the verified `Withdraw` module.
- **`ClaimSpec` and `ClaimIndividualVestingSpec`:** These functions model the post-sale token claim process for public participants and private stakeholders, respectively. They enforce that claims can only occur after a `Success` state is reached and correctly update the user's `claimed` balance by delegating the complex allocation and vesting calculations to the `Claim` module.
- **`DistributeTokensSpec`:** Defines the administrative state transition for distributing tokens to project stakeholders. This action is guarded to ensure it only happens post-success and orchestrates the update of the `distributedAccounts` list by calling the verified `Distribution` module.

The verification of these top-level transitions is a powerful demonstration of the compositional strategy. The proofs at this layer do not re-verify the complex financial safety properties (like refund safety or vesting curve monotonicity). Instead, they focus solely on proving that the global state fields are updated correctly based on the outputs from the already-proven workflow functions. This separation of concerns reduces the safety of the entire system to the correctness of its orchestration logic, given the proven correctness of its parts [16].

6. Limitations and Future Work

While the compositional verification in Dafny provides a high degree of assurance regarding the internal consistency of the launchpad's business logic, it is crucial to acknowledge the inherent limitations of this approach and outline avenues for future research. The current formal model serves as a powerful mathematical specification, but its relationship to the production code and the execution environment warrants further discussion.

6.1. The Gap Between Specification and Production Code

A significant limitation of the current methodology is the separation between the formally verified Dafny code and the production-level Rust code, which is the artifact ultimately deployed to the blockchain. The Dafny model is a pure, mathematical representation of the logic. For the proofs to apply to the production system, there is a critical, implicit step: a human expert must manually audit the Rust implementation to ensure it is a faithful and precise translation of the verified Dafny specification.

This manual verification step, while standard in many formal methods applications, introduces a potential point of failure. Future work could focus on bridging this "specification-implementation gap" to achieve machine-checkable correspondence. Two promising directions emerge:

1. **Verified Code Generation:** One approach is to generate the production code directly from the verified specification. A trusted "Dafny-to-Rust" compiler could translate the proven Dafny logic into a Rust module, ensuring by construction that the deployed code adheres to the formal model. However, this approach faces significant practical hurdles. At present, no production-ready and trusted Dafny-to-Rust generator exists. Moreover, such a tool would need to be highly specialized to support the specific features of the NEAR blockchain, including its state management and asynchronous contract model, which represents a substantial engineering challenge in its own right.
2. **Integrated Specification and Implementation:** An alternative and more modern approach involves tools that allow formal specifications and proofs to be written directly within the production code. Languages and tools like **Verus** enable annotating Rust code with preconditions, postconditions, and invariants, which are then verified in place [17]. This unifies the specification and implementation into a single artifact, eliminating the need for manual correspondence checks and bringing formal verification closer to the production code. Nevertheless, this approach also has its drawbacks. The Verus ecosystem, while promising, is at an earlier stage of development compared to Dafny, and its toolset for formal verification is currently less mature. A significant practical issue is the limited IDE support, which can make debugging complex formal specification rules a considerably more challenging task.

6.2. Abstraction from the NEAR Execution Environment

The current model is a purposeful abstraction away from the complexities of the NEAR blockchain's execution environment. This was a necessary simplification to make the verification of the complex financial logic tractable. However, this abstraction inherently limits the scope of properties that can be proven.

The model does not account for crucial aspects of the on-chain environment, such as:

- The asynchronous, message-passing nature of cross-contract calls.
- Gas mechanics and the possibility of out-of-gas failures.
- Potential reentrancy vulnerabilities arising from complex call patterns.

These factors can introduce complications that the current, purely functional model cannot address. For instance, the withdrawal reentrancy issue discovered during development highlighted how interactions with the execution environment could affect security in ways not captured by the abstract logic. A significant, albeit highly challenging, area for future work would be to formalize the semantics of the NEAR execution environment itself. This would enable proving properties that hold not just in theory but also within the concrete operational context of the blockchain.

6.3. Expanding the Scope of Verified Properties

The current verification focuses primarily on critical safety properties, such as the correctness of refund calculations and adherence to the sale cap. In an ideal world, a fully verified contract would guarantee a broader spectrum of properties. The following categories represent a long-term roadmap for expanding the scope of verification:

- **Validity:** A global invariant stating that if the contract begins in a valid state, any possible sequence of transactions will keep it in a valid state. This is a generalization of the state machine integrity proofs, ensuring holistic system consistency.
- **Liveness:** Guarantees that certain desirable states are always eventually reachable. A critical special case, often called **Liquidity**, is the property that a user can always, under some sequence of valid actions, withdraw

their entitled funds from the contract. Proving liveness is notoriously difficult as it must account for potential interference from the external environment (e.g., frontrunning or other adversarial actions).
- **Fidelity:** This property ensures that the contract's internal representation of assets (e.g., token balances in its ledger) is always equal to the actual amount of assets cryptographically controlled by the contract. This would formally prove that funds cannot be lost or become permanently inaccessible due to bugs in the accounting logic.

The verification of these broader liveness and fidelity properties constitutes a long-term research objective for the field. The preceding discussion serves to delineate the precise scope of the guarantees provided by the present work, positioning it as a foundational step focused on core safety invariants, from which more comprehensive verification efforts may proceed.

7. Conclusion

This paper has detailed the formal verification of a token sale launchpad's core logic using Dafny. We have demonstrated that by adopting a **compositional, bottom-up verification strategy**, it is possible to formally reason about a system with complex, interacting components and non-linear arithmetic [9].

The key achievements of this work include:

1. **A Layered Proof Architecture:** Decomposing the problem from foundational mathematical lemmas to top-level state transitions, making a complex proof tractable.
2. **Verification of Non-Linear Arithmetic:** Proving and reusing a core set of lemmas for integer multiplication and division, which are essential for financial calculations.
3. **Proof of Critical Business Rules:** Formalizing and verifying rules such as the non-overlapping nature of discount periods.
4. **Mathematical Guarantee of Financial Safety:** The cornerstone of this work is the formal proof of `Lemma_RefundIsSafe` and `Lemma_AssetsRevert_RoundTrip_bounds`. Together, they demonstrate not only that refunds never exceed deposits, but also that precision loss from round-trip calculations is strictly and predictably bounded.
5. **Verified State Machine Lifecycle:** Proving the integrity of the contract's entire lifecycle, including user deposits, withdrawals, token claims with vesting, and post-sale distributions, ensuring predictable and correct state transitions over time.

This work provides strong evidence that formal methods are not merely an academic exercise but a practical and powerful tool for engineering high-assurance financial systems, providing mathematical certainty where traditional testing can only provide statistical confidence [4], [18].

Appendix A: Formal Proofs of Foundational Integer Arithmetic Properties

The `MathLemmas` module constitutes the axiomatic foundation upon which the entire verification hierarchy is constructed. Automated theorem provers, including the Z3 SMT solver employed by Dafny, possess comprehensive theories for linear arithmetic [19]. However, reasoning about non-linear expressions involving multiplication and division often necessitates explicit, programmer-provided proofs. This module furnishes these proofs, creating a trusted library of fundamental mathematical properties. This approach abstracts the intricacies of integer arithmetic, thereby enabling the verification of higher-level business logic in a more declarative and computationally tractable manner.

Lemma 1: Monotonicity of Integer Division

This lemma formally establishes that the integer division operation ($\lfloor a/b \rfloor$) preserves the non-strict inequality relation ($\geq$).

Formal Specification (`Lemma_Div_Maintains_GTE`)

$$\forall x, y, k \in \mathbb{N} : (k > 0 \land x \geq y) \implies \lfloor x/k \rfloor \geq \lfloor y/k \rfloor$$

Description and Verification Strategy

The lemma asserts that for any two natural numbers x and y where x is greater than or equal to y, dividing both by a positive integer k will preserve this order relation. The proof implemented in Dafny is a classic *reductio ad absurdum*.

1. **Hypothesis:** The proof begins by positing the negation of the consequent: $\lfloor x/k \rfloor < \lfloor y/k \rfloor$. Within the domain of integers, this is equivalent to $\lfloor x/k \rfloor + 1 \leq \lfloor y/k \rfloor$.
2. **Derivation:** Leveraging the definition of Euclidean division, $a = \lfloor a/b \rfloor \cdot b + (a \pmod b)$, the proof constructs a lower bound for y [20].. By substituting the hypothesis, we obtain: $y \geq \lfloor y/k \rfloor \cdot k \geq (\lfloor x/k \rfloor + 1) \cdot k = (\lfloor x/k \rfloor \cdot k) + k$
3. **Contradiction:** It is a known property that $k > (x \pmod k)$. Therefore, we can deduce that $(\lfloor x/k \rfloor \cdot k) + k > (\lfloor x/k \rfloor \cdot k) + (x \pmod k) = x$. This establishes the inequality $y > x$, which is a direct contradiction of the lemma's precondition $x \geq y$.
4. **Conclusion:** As the initial hypothesis leads to a logical contradiction, it must be false. Consequently, the original consequent, $\lfloor x/k \rfloor \geq \lfloor y/k \rfloor$, is proven to be true for all inputs satisfying the preconditions.

Verification Effectiveness: By formalizing this property as a standalone lemma, we provide the verifier with a powerful and reusable inference rule. For any subsequent proof involving inequalities and division, a simple invocation of this lemma suffices. This obviates the need for the SMT solver to rediscover this non-trivial, non-linear property within more complex logical contexts, thereby significantly enhancing the automation, performance, and predictability of the overall verification process.

Lemma 2: Scaling by a Rational Factor ≥ 1 (`Lemma_MulDivGreater_From_Scratch`)

This lemma proves that scaling an integer by a rational factor y/k (where $y \geq k$) results in a value no less than the original.

Formal Specification

$$\forall x, y, k \in \mathbb{N} : (x > 0 \land k > 0 \land y \geq k) \implies \lfloor (x \cdot y)/k \rfloor \geq x$$

Description and Verification Strategy

This lemma is instrumental in verifying that financial conversions at stable or favorable prices do not lead to a loss of principal value. The verification strategy is **compositional**, demonstrating the elegance of building proofs upon previously established theorems.

1. **Intermediate Premise:** The preconditions $y \geq k$ and $x > 0$ directly imply the inequality $x \cdot y \geq x \cdot k$.
2. **Compositional Invocation:** The proof then applies the previously proven `Lemma_Div_Maintains_GTE` to this intermediate inequality, substituting $x \cdot y$ for its first parameter and $x \cdot k$ for its second.
3. **Logical Deduction:** This invocation yields the statement $\lfloor (x \cdot y)/k \rfloor \geq \lfloor (x \cdot k)/k \rfloor$.
4. **Simplification:** Given $k > 0$, the term $\lfloor (x \cdot k)/k \rfloor$ is definitionally equivalent to x. This leads directly to the desired postcondition.

Verification Effectiveness: This exemplifies an efficient, layered verification approach. The proof reduces a complex, non-linear problem to a straightforward application of a known monotonicity property. This modularity not only enhances human comprehension but also simplifies the task for the SMT solver, making the verification near-instantaneous.

Lemma 3: Strict Scaling by a Rational Factor ≥ 2 (`Lemma_MulDivStrictlyGreater_From_Scratch`)

This lemma establishes a sufficient condition to guarantee a *strict* increase in value after scaling, providing a robust guard against value loss due to integer division's truncating nature.

Formal Specification

$$\forall x, y, k \in \mathbb{N} : (x > 0 \land k > 0 \land y \geq 2k) \implies \lfloor (x \cdot y)/k \rfloor > x$$

Description and Verification Strategy

The proof recognizes that the precondition $y > k$ is insufficient to guarantee strict inequality. It employs the stronger condition $y \geq 2k$.

1. **Strengthened Premise:** The proof establishes that $y \geq 2k$ implies $x \cdot y \geq x \cdot (2k) = x \cdot k + x \cdot k$. As $x > 0$ and $k > 0$, it follows that $x \cdot k \geq k$. This allows the derivation of the crucial inequality $x \cdot y \geq x \cdot k + k$.
2. **Compositional Invocation:** This inequality is the exact premise required by a stricter variant of the monotonicity lemma (`Lemma_Div_Maintains_GT`), which proves $a \geq b + k \implies \lfloor a/k \rfloor > \lfloor b/k \rfloor$. Applying this specialized lemma yields $\lfloor (x \cdot y)/k \rfloor > \lfloor (x \cdot k)/k \rfloor$.
3. **Conclusion:** The term $\lfloor (x \cdot k)/k \rfloor$ simplifies to x, thus proving the postcondition.

Verification Effectiveness: This lemma showcases a critical aspect of formal methods: identifying the precise and sufficiently strong preconditions required to guarantee a desired property. By encapsulating this logic, we create a tool for reasoning about scenarios where a tangible gain must be proven, such as the application of a significant bonus.

Lemmas 4 & 5: Scaling by a Rational Factor ≤ 1

These lemmas are the logical duals to the preceding two, addressing scaling by factors less than or equal to one.

Formal Specification

1. `Lemma_MulDivLess_From_Scratch`:

   $\forall x, y, k \in \mathbb{N} : (x > 0 \land y > 0 \land k \geq y) \implies \lfloor (x \cdot y)/k \rfloor \leq x$

2. `Lemma_MulDivStrictlyLess_From_Scratch`:

   $\forall x, y, k \in \mathbb{N} : (x > 0 \land y > 0 \land k > y) \implies \lfloor (x \cdot y)/k \rfloor < x$

Description and Verification Strategy

The proofs demonstrate both elegance and efficiency through reuse and contradiction.

- The proof for the non-strict case (`...Less...`) is achieved by a clever reuse of

  `Lemma_MulDivGreater_From_Scratch`. Given $k \geq y$, it invokes the greater-than lemma with the roles of k and y interchanged.

- The proof for the strict case (`...StrictlyLess...`) proceeds by contradiction. It assumes $\lfloor (x \cdot y)/k \rfloor \geq x$, which implies $x \cdot y \geq x \cdot k$, and for $x > 0$, implies $y \geq k$. This directly contradicts the lemma's precondition $k > y$.

Verification Effectiveness: These proofs highlight the power of a well-curated lemma library. Reusing existing proofs minimizes redundant effort, while the declarative nature of the proof by contradiction allows the SMT solver to efficiently explore the logical space and confirm the inconsistency.

Lemma 6: The Bounded Property of Integer Division Truncation (`Lemma_DivMul_Bounds`)

This lemma formalizes the fundamental property that integer division is a truncating operation, which is the root cause of potential precision loss in round-trip calculations. It proves not only that the result does not exceed the original value but also that the "loss" is strictly bounded.

Formal Specification

$$\forall x, y \in \mathbb{N} : (y > 0) \implies (\lfloor x/y \rfloor \cdot y \leq x) \land (0 \leq x - (\lfloor x/y \rfloor \cdot y) < y)$$

Description and Verification Strategy The proof of this lemma is a testament to the synergy between the programmer and the underlying verification engine. It is established by asserting the **Euclidean Division Theorem**, a core axiom within the SMT solver's theory of integers: `assert x == (x / y) * y + (x % y);`

From this axiom, both postconditions follow immediately. Since the remainder `(x % y)` is definitionally non-negative, `x` must be greater than or equal to the term `(x / y) * y`. The second property, `x - (x / y) * y < y`, follows from the fact that `x % y` is definitionally less than the divisor `y`.

Verification Effectiveness This is a paradigmatic example of effective formal verification. The programmer's role is not to re-prove foundational mathematics but to strategically invoke known axioms to guide the verifier's reasoning. By stating this single, axiomatic assertion, we provide the solver with the necessary fact to prove the safety and bounded loss of all round-trip financial calculations throughout the system.

Lemma 7: Lower Bound of Division from Strict Multiplication (`Lemma_DivLowerBound_from_StrictMul`)

This lemma proves a subtle but powerful property of non-linear integer arithmetic that is often non-trivial for SMT solvers to deduce on their own. It establishes a lower bound for a division's result based on a strict inequality involving a product.

Formal Specification

$$\forall a, b, c \in \mathbb{N} : (c > 0 \land a > b \cdot c) \implies \lfloor a/c \rfloor \geq b$$

Description and Verification Strategy

The lemma asserts that if a number `a` is strictly greater than a product `b * c`, then dividing `a` by `c` must yield a result of at least `b`. The proof is a classic *reductio ad absurdum*.

1. **Hypothesis:** The proof begins by assuming the negation of the consequent: $\lfloor a/c \rfloor < b$. For integers, this is equivalent to $\lfloor a/c \rfloor \leq b - 1$.
2. **Derivation:** Using the definition of Euclidean division, $a = \lfloor a/c \rfloor \cdot c + (a \pmod c)$, the proof constructs an upper bound for `a`. By substituting the hypothesis, we obtain: $a \leq (b-1) \cdot c + (a \pmod c) = b \cdot c - c + (a \pmod c)$
3. **Contradiction:** We know that the remainder $(a \pmod c)$ is strictly less than the divisor `c`. This allows us to establish a strict inequality: $b \cdot c - c + (a \pmod c) < b \cdot c - c + c = b \cdot c$. This establishes that $a < b \cdot c$, which is a direct contradiction of the lemma's precondition $a > b \cdot c$.
4. **Conclusion:** As the initial hypothesis leads to a logical contradiction, it must be false. Consequently, the original consequent, $\lfloor a/c \rfloor \geq b$, is proven to be true.

Verification Effectiveness

This lemma serves as a crucial piece of guidance for the verifier. By proving this non-linear property explicitly, we equip the SMT solver with a ready-made inference rule. This is particularly vital in proofs of round-trip calculations with tight bounds, such as `Lemma_WeightOriginal_RoundTrip_bounds`, where proving that a value is greater than or equal to `amount - 1` requires exactly this kind of reasoning. Encapsulating this logic prevents the solver from getting stuck or timing out while trying to rediscover this relationship in a more complex context, thereby improving the robustness and performance of the overall verification.

Appendix B: Formal Verification of Asset Conversion Logic

The `AssetCalculations` module represents the first layer of application-specific business logic, constructed upon the axiomatic foundation established in `MathLemmas`. Its purpose is to translate the abstract mathematical properties of integer arithmetic into concrete, provable guarantees for financial asset conversion operations. This module defines the pure mathematical specifications for conversion and provides a comprehensive suite of lemmas that formally prove their key properties, such as monotonicity, predictable behavior under various price conditions, and, most critically, round-trip safety.

B.1. Core Specification Functions

At the heart of the module lie two pure functions defining the mathematical essence of forward and reverse conversion. For clarity, let $w \in \mathbb{N}$ represent the input amount (weight or principal), $d_T \in \mathbb{N}^+$ be the denominator of the price fraction (e.g., the deposit token amount), and $s_T \in \mathbb{N}^+$ be the numerator (e.g., the sale token amount). Let C denote the direct conversion (`CalculateAssetsSpec`) and R denote the reverse conversion (`CalculateAssetsRevertSpec`).

1. **Direct Conversion (C):** This function maps a principal amount into a quantity of target assets.

$$C(w, d_T, s_T) := \lfloor (w \cdot s_T)/d_T \rfloor$$

2. **Reverse Conversion (R):** This function performs the inverse operation, calculating the principal amount from a quantity of assets.

$$R(w, d_T, s_T) := \lfloor (w \cdot d_T)/s_T \rfloor$$

B.2. Verification of Direct Conversion Properties (`CalculateAssets`)

This group of lemmas proves intuitive economic properties of the function C by directly mapping them to the foundational lemmas from Appendix A.

Lemma B.2.1: Conversion at a Non-Disadvantageous Price (`Lemma_CalculateAssets_IsGreaterOrEqual`)

- **Formal Specification:**

$$\forall w, d_T, s_T \in \mathbb{N}^+ : (s_T \geq d_T) \implies C(w, d_T, s_T) \geq w$$

- **Description and Verification Strategy:** This lemma guarantees that if the exchange rate is stable or favorable ($s_T \geq d_T$), the resulting asset quantity has a nominal value no less than the original principal. The proof is a direct **instantiation** of `Lemma_MulDivGreater_From_Scratch`. The parameters are mapped as follows: $x \to w$, $y \to s_T$, $k \to d_T$. The lemma's precondition $s_T \geq d_T$ precisely matches the required precondition $y \geq k$ from `MathLemmas`.
- **Verification Effectiveness:** This demonstrates the power of compositional reasoning. A complex financial guarantee is proven with a single invocation of a previously verified, general-purpose lemma, making the proof trivial for the SMT solver and transparent to a human auditor.

Lemma B.2.2: Conversion at a Highly Advantageous Price (`Lemma_CalculateAssets_IsGreater`)

- **Formal Specification:**

$$\forall w, d_T, s_T \in \mathbb{N}^+ : (s_T \geq 2 \cdot d_T) \implies C(w, d_T, s_T) > w$$

- **Description and Verification Strategy:** This guarantees a *strict* increase in nominal value when the exchange rate is significantly favorable. The precondition $s_T \geq 2 \cdot d_T$ is necessary to overcome the truncating effect of integer division. The proof is a direct instantiation of `Lemma_MulDivStrictlyGreater_From_Scratch`.
- **Verification Effectiveness:** This highlights the importance of identifying precise preconditions to obtain strict guarantees. The lemma is crucial for proving scenarios where not just non-loss, but a tangible gain, must be formally assured.

Lemma B.2.3: Conversion at an Unfavorable Price (`Lemma_CalculateAssets_IsLess`)

- **Formal Specification:**

  $\forall w, d_T, s_T \in \mathbb{N}^+ : (s_T < d_T) \implies C(w, d_T, s_T) < w$

- **Description and Verification Strategy:** This guarantees that if the exchange rate is unfavorable, the resulting asset quantity has a nominal value strictly less than the original principal. The proof is a direct instantiation of `Lemma_MulDivStrictlyLess_From_Scratch`.
- **Verification Effectiveness:** This completes the suite of behavioral guarantees for the C function, covering all three possible price relationships ($\geq$, $=$, $<$) and ensuring the function's behavior is fully specified and proven.

B.3. Verification of Reverse Conversion Properties (`CalculateAssetsRevert`)

This set of lemmas proves symmetric properties for the reverse function R. The verification strategy is analogous: instantiation of foundational lemmas. The key observation is that `R(w, d_T, s_T)` is mathematically equivalent to `C(w, s_T, d_T)`, meaning a reverse conversion is simply a direct conversion with the roles of the price fraction's numerator and denominator exchanged.

Lemma B.3.1: Reversion from an Originally Unfavorable Price

(`Lemma_CalculateAssetsRevert_IsGreaterOrEqual`)

- **Formal Specification:**
$$\forall w, d_T, s_T \in \mathbb{N}^+ : (d_T \geq s_T) \implies R(w, d_T, s_T) \geq w$$
- **Description and Verification Strategy:** If the original price was unfavorable or stable for the user ($s_T \leq d_T$), then converting the assets back will yield a principal amount no less than the asset amount being converted. The proof invokes `Lemma_MulDivGreater_From_Scratch` with the parameter mapping $x \to w$, $y \to d_T$, $k \to s_T$. The precondition $d_T \geq s_T$ correctly satisfies the required $y \geq k$.
- **Verification Effectiveness:** This demonstrates the elegance of symmetric arguments in formal proofs. Instead of constructing a new complex proof, we reuse an existing lemma by simply permuting its arguments, which serves to validate the generality and correctness of the foundational axioms.

B.4. Verification of Composite and Crucial Safety Properties

These lemmas establish higher-order properties that are critical for the overall safety and integrity of the financial logic.

Lemma B.4.1: Monotonicity of Reverse Conversion (`Lemma_CalculateAssetsRevertSpec_Monotonic`)

- **Formal Specification:**

  $$\forall w_1, w_2, d_T, s_T \in \mathbb{N}^+ : (w_1 \leq w_2) \implies R(w_1, d_T, s_T) \leq R(w_2, d_T, s_T)$$
- **Description and Verification Strategy:** This lemma proves that the reverse conversion function R is monotonic. That is, converting a smaller quantity of assets back to the principal cannot yield a larger result than converting a larger quantity. This property is an absolute prerequisite for proving the safety of partial refund calculations. The proof is based on `Lemma_Div_Maintains_GTE`. From $w_1 \leq w_2$, it follows that $w_1 \cdot d_T \leq w_2 \cdot d_T$. Applying `Lemma_Div_Maintains_GTE` to this inequality with divisor s_T directly yields the desired consequent.
- **Verification Effectiveness:** This shows how foundational lemmas are used to prove higher-order properties (monotonicity), which in turn serve as essential building blocks for even more complex safety proofs, such as refund correctness.

B.4.2: The Algebraic Equation for Round-Trip Loss (`Lemma_RoundTripLossEquation`)

- **Formal Specification:** Let $w, d_T, s_T \in \mathbb{N}^+$. Let:
  - $assets := \lfloor (w \cdot s_T)/d_T \rfloor$
  - $reverted := \lfloor (assets \cdot d_T)/s_T \rfloor$
  - $rem_1 := (w \cdot s_T) \pmod{d_T}$
  - $rem_2 := (assets \cdot d_T) \pmod{s_T}$ Then the following equality holds:

  $$(w - reverted) \cdot s_T = rem_1 + rem_2$$
- **Description and Verification Strategy:** This lemma provides the algebraic foundation for proving the bounded loss property. It isolates the complex arithmetic into a single, elegant equation. It proves that the "loss" from a round-trip conversion, when scaled up by `sT`, is precisely equal to the sum of the remainders from the two integer division operations involved. The proof in Dafny is a straightforward algebraic manipulation using the `calc` statement, which makes the reasoning explicit and easy for the verifier to follow:
  1. Start with the expression `(w - reverted) * sT`.
  2. Apply the Euclidean division theorem to `w * sT`, substituting it with `(assets * dT + rem1)`.
  3. Similarly, apply the theorem to `assets * dT`, substituting it with `(reverted * sT + rem2)`.
  4. The expression becomes `((reverted * sT + rem2) + rem1) - (reverted * sT)`.
  5. Simplifying this expression directly yields `rem1 + rem2`, completing the proof.
- **Verification Effectiveness:** This lemma is a prime example of effective proof engineering. By isolating this non-trivial algebraic identity into a standalone proof, we greatly simplify the main safety proof of `Lemma_AssetsRevert_RoundTrip_bounds`. Instead of forcing the SMT solver to re-derive this equality from first principles within a more complex logical context, we provide it as a trusted, reusable theorem. This makes the final proof more readable, robust, and computationally efficient.

Lemma B.4.3: Round-Trip Calculation Safety and Bounded Loss (`Lemma_AssetsRevert_RoundTrip_bounds`)

- **Formal Specification:**

  $\forall w, d_T, s_T \in \mathbb{N}^+ : C(w, d_T, s_T) > 0 \implies R(C(w, d_T, s_T), d_T, s_T) \leq w \wedge (w - R(C(w, d_T, s_T), d_T, s_T)) \cdot s_T < d_T + s_T$

- **Description and Verification Strategy:** This is the **central safety guarantee** of this module. It formally proves that the sequential application of a direct conversion and a reverse conversion cannot create value *ex nihilo*. Furthermore, it proves the stronger property that the value loss from integer truncation is strictly bounded. The proof is a composition of several established facts:

  1. Let `assets := C(w, d_T, s_T)` and `reverted := R(assets, d_T, s_T)`.
  2. The `reverted <= w` inequality is proven as before, using `Lemma_DivMul_Bounds` and `Lemma_Div_Maintains_GTE`.
  3. The stronger bounded loss property is proven using a dedicated helper lemma, `Lemma_RoundTripLossEquation`. This lemma algebraically demonstrates that the scaled loss, `(w - reverted) * sT`, is exactly equal to the sum of the remainders from the two division operations: `(w * sT) % dT + (assets * dT) % sT`.
  4. Since a remainder from division by `k` is always strictly less than `k`, we know that `(w * sT) % dT < dT` and `(assets * dT) % sT < sT`.
  5. Summing these two inequalities gives `rem1 + rem2 < dT + sT`, which completes the proof.

- **Verification Effectiveness:** This lemma is the culmination of the `AssetCalculations` module. It demonstrates how multiple simple, proven properties can be chained to prove a complex, critically important safety property. It provides not just a guarantee against value creation, but a strict, provable upper bound for any truncation-related losses.

Appendix C: Formal Verification of Time-Based Discount Logic

The `Discounts` module formalizes the logic for applying time-sensitive percentage-based bonuses. It employs fixed-point arithmetic to handle percentages with precision and establishes a rigorous framework to ensure that discount rules are applied consistently and unambiguously. The verification effort for this module guarantees not only the correctness of the core financial calculations but also the logical integrity of collections of discounts, preventing common business logic flaws such as applying multiple bonuses simultaneously.

C.1. Foundational Definitions and Predicates

The module is built upon a set of core definitions representing the properties of a single discount. Let the constant M denote the `MULTIPLIER` (e.g., 10000 for four decimal places of precision), which serves as the basis for fixed-point arithmetic. A `Discount`, d, is a tuple (s, e, p) where $s, e, p \in \mathbb{N}$, representing `startDate`, `endDate`, and `percentage` respectively.

Predicate C.1.1: Validity of a Discount (`ValidDiscount`)

- **Formal Specification:** A discount $d = (s, e, p)$ is considered valid if its parameters are self-consistent.

$$\text{ValidDiscount}(d) \iff (p \in (0, M] \wedge s < e)$$

- **Description:** This predicate enforces two fundamental business rules: the discount percentage p must be positive and not exceed 100% (represented by M), and the time interval must be logical (the start date must precede the end date). This predicate forms the base assumption for all operations on a discount.

Predicate C.1.2: Activity of a Discount (`IsActive`)

- **Formal Specification:** A discount $d = (s, e, p)$ is active at a given time $t \in \mathbb{N}$ if t falls within its effective time range.

$$\text{IsActive}(d, t) \iff s \leq t < e$$

- **Description:** This defines the discount's active period as a half-open interval $[s, e)$. This is a common and unambiguous convention in time-based systems, ensuring that `endDate` is the first moment in time when the discount is no longer active.

C.2. Verification of Discount Application Logic

This section formalizes the application of a discount to a principal amount and proves its mathematical properties. Let $W_A(a, p)$ denote the `CalculateWeightedAmount` function, where $a \in \mathbb{N}^+$ is the amount and p is the percentage from a valid discount.

Function C.2.1: Weighted Amount Calculation (`CalculateWeightedAmount`)

- **Formal Specification:**
$$W_A(a, p) := \lfloor (a \cdot (M + p))/M \rfloor$$
- **Description:** This function calculates the new "weighted" amount by scaling the original amount a by a factor of $(1 + p/M)$. The formula is implemented using integer arithmetic to avoid floating-point numbers.

Lemma C.2.2: Non-Decreasing Property of Discount Application

(`Lemma_CalculateWeightedAmount_IsGreaterOrEqual`)

- **Formal Specification:**
$$\forall a, p \in \mathbb{N}^+ : W_A(a, p) \geq a$$
- **Description and Verification Strategy:** This lemma guarantees that applying any valid discount will never decrease the principal amount. The proof is a direct instantiation of `Lemma_MulDivGreater_From_Scratch` from Appendix A. Since $p > 0$, it holds that $M + p \geq M$. This satisfies the $y \geq k$ precondition, making the proof trivial.

C.3. Verification of Discount Reversion Logic

This section handles the inverse operation: calculating the original amount from a weighted amount. Let $O_A(w_a, p)$ denote `CalculateOriginalAmount`, where $w_a \in \mathbb{N}^+$ is the weighted amount.

Function C.3.1: Original Amount Calculation (`CalculateOriginalAmount`)

- **Formal Specification:**
$$O_A(w_a, p) := \lfloor (w_a \cdot M)/(M + p) \rfloor$$
- **Description:** This function reverts the discount application, effectively scaling the weighted amount w_a by a factor of $M/(M + p)$.

Lemma C.3.2: Non-Increasing Property of Discount Reversion

(`Lemma_CalculateOriginalAmount_IsLessOrEqual`)

- **Formal Specification:**
$$\forall w_a, p \in \mathbb{N}^+ : O_A(w_a, p) \leq w_a$$
- **Description and Verification Strategy:** This guarantees that reverting a discount cannot result in a value greater than the weighted amount it was derived from. The proof instantiates `Lemma_MulDivLess_From_Scratch`. Since $p > 0$, it holds that $M \leq M + p$, which satisfies the $k \geq y$ precondition.

C.4. Verification of Collection Consistency Properties

These properties are critical as they govern the behavior of a set of discounts, ensuring logical integrity at the system level. Let $D = (d_0, d_1, ..., d_{n-1})$ be a sequence of discounts.

Predicate C.4.1: Non-Overlapping Discounts (`DiscountsDoNotOverlap`)

- **Formal Specification:** A sequence of discounts D is non-overlapping if for any two distinct discounts, their active time intervals are disjoint. Let $d_i = (s_i, e_i, p_i)$.

  $\text{DiscountsDoNotOverlap}(D) \iff \forall i, j \in [0, n-1] : (i < j \implies e_i \leq s_j \vee e_j \leq s_i)$
- **Description:** This is a crucial business rule that prevents ambiguity. It ensures that no two discount periods can be active at the same time, which is fundamental for deterministic calculations.

Lemma C.4.2: Uniqueness of Active Discount (`Lemma_UniqueActiveDiscount`)

- **Formal Specification:** If a sequence of discounts D is non-overlapping, then at any given time t, at most one discount in the sequence can be active.

  $\text{DiscountsDoNotOverlap}(D) \implies \forall i, j \in [0, n-1], \forall t \in \mathbb{N} : (\text{IsActive}(d_i, t) \land \text{IsActive}(d_j, t) \implies i = j)$

- **Description and Verification Strategy:** This is the most important safety property for the collection of discounts. It guarantees that any search for an active discount will yield an unambiguous result. The proof proceeds by contradiction. Assume $i \neq j$ and both d_i and d_j are active at time t.

  1. $\text{IsActive}(d_i, t) \implies s_i \leq t < e_i$
  2. $\text{IsActive}(d_j, t) \implies s_j \leq t < e_j$
  3. From these, it follows that $s_i < e_j$ and $s_j < e_i$.
  4. This contradicts the `DiscountsDoNotOverlap(D)` predicate, which requires $e_i \leq s_j$ or $e_j \leq s_i$.
  5. Therefore, the initial assumption ($i \neq j$) must be false, proving that $i = j$.

- **Verification Effectiveness:** This lemma is a prime example of proving a high-level system property as a direct logical consequence of a lower-level data invariant. By verifying this, Dafny provides a mathematical guarantee that the core business logic for finding and applying bonuses is free from race conditions or ambiguity related to time, which is a common and critical failure mode in financial systems [21].

Appendix D: Formal Verification of System Configuration and Composite Logic

The `Config` module serves as the central nervous system of the launchpad specification. It aggregates all system parameters, business rules, and component configurations into a single, immutable data structure. This module's primary function is to compose the verified primitives from lower-level modules (such as `Discounts`) into higher-level, context-aware specifications. Its verification ensures that these composite operations maintain the safety properties established by their constituent parts and that the system's overall parameterization is logically sound [8].

D.1. The `Config` Datatype and Core Invariants

The state of the system's static configuration is captured by the `Config` datatype, denoted here as Γ. It is a tuple comprising various parameters, including the sale mechanics, dates, and sequences of sub-structures like discounts.

Predicate D.1.1: System-Wide Validity (`ValidConfig`)

The `ValidConfig` predicate is the root invariant for the entire system. It asserts that the configuration Γ is well-formed by taking a logical conjunction of numerous component-level validity predicates, ensuring holistic integrity before any transaction is processed.

- **Formal Specification:** Let Γ be a configuration instance. $\text{ValidConfig}(\Gamma) \iff P_{\text{dates}} \land P_{\text{mechanics}} \land P_{\text{discounts}} \land P_{\text{vesting}} \land P_{\text{stakeholders}} \land P_{\text{accounting}}$ where each component predicate is defined as:
  - **Date Consistency (P_{dates}):** $\Gamma.startDate < \Gamma.endDate$
  - **Mechanics Consistency ($P_{\text{mechanics}}$):** $\Gamma.mechanic.FixedPrice? \implies (\Gamma.mechanic.depositTokenAmount > 0 \land \Gamma.mechanic.saleTokenAmount > 0)$
  - **Discounts Consistency ($P_{\text{discounts}}$):** $DiscountsDoNotOverlap(\Gamma.discount) \land (\forall d \in \Gamma.discount : ValidDiscount(d))$
  - **Global Vesting Consistency (P_{vesting}):** $\Gamma.vestingSchedule.Some? \implies ValidVestingSchedule(\Gamma.vestingSchedul$
  - **Stakeholder Consistency ($P_{\text{stakeholders}}$):** $\text{IsUnique}(\Gamma.distributionProportions) \land (\forall p \in \Gamma.distributionProportions$ $p.Valid())$
  - **Accounting Consistency ($P_{\text{accounting}}$):** $\Gamma.totalSaleAmount = \Gamma.saleAmount + \text{SumOfStakeholderAllocations}(\Gamma.di$
- **Description:** This predicate establishes a comprehensive baseline of sanity for the system's parameters. In addition to basic checks on dates and sale mechanics, it now enforces several critical invariants:
  - **Accounting Integrity:** It guarantees that the `totalSaleAmount` is precisely the sum of the public sale amount and all private stakeholder allocations. This prevents configuration-level bugs that could lead to token supply inflation or deflation.
  - **Stakeholder Uniqueness:** It ensures that all stakeholder accounts (including the solver) are unique, preventing ambiguous or incorrect distributions.
  - **Recursive Validity:** It recursively validates each individual stakeholder's configuration, including any private vesting schedules they may have.

`ValidConfig` serves as a crucial precondition for all functions that operate on the configuration, ensuring they are never invoked with inconsistent or illogical data.

D.2. High-Level Specification of Composite Calculations

This section analyzes the core functions within `Config` that combine the system's state (time) with financial primitives (discount application) to produce context-dependent results.

Function D.2.1: Specification for Weighted Amount Calculation (`CalculateWeightedAmountSpec`)

Let $W_S(a, t, \Gamma)$ denote this specification, which computes the weighted amount for a principal a at time t under configuration Γ. Let $F(D, t)$ be the `FindActiveDiscountSpec` function, which returns `Some(d)` if an active discount d exists in sequence D at time t, and `None` otherwise.

- **Formal Specification:** For $a > 0$:

$$W_S(a, t, \Gamma) := \begin{cases} a & \text{if } F(\Gamma.\text{discount}, t) = \text{None} \\ W_A(a, d.p) & \text{if } F(\Gamma.\text{discount}, t) = \text{Some}(d) \end{cases}$$

  where $W_A(a, p)$ is the `CalculateWeightedAmount` function from Appendix C.
- **Description:** This function acts as a logical switch. It models the behavior of applying a discount if and only if one is active at the specified time. It encapsulates the search-and-apply logic into a single pure function.

Lemma D.2.2: Monotonicity of Weighted Amount Calculation (`Lemma_CalculateWeightedAmountSpec_Monotonic`)

- **Formal Specification:**

$$\forall a_1, a_2, t \in \mathbb{N}, \forall \Gamma : (\text{ValidConfig}(\Gamma) \land a_1 \leq a_2) \implies W_S(a_1, t, \Gamma) \leq W_S(a_2, t, \Gamma)$$

- **Description and Verification Strategy:** This critical lemma proves that the system-level weighting function is monotonic. The proof proceeds by case analysis on the result of $F(\Gamma.\mathit{discount}, t)$:
  1. **Case `None`:** $W_S(a, t, \Gamma) = a$. The property reduces to $a_1 \leq a_2$, which is true by the precondition.
  2. **Case `Some(d)`:** The property becomes $W_A(a_1, d.p) \leq W_A(a_2, d.p)$. This is equivalent to proving $\lfloor (a_1 \cdot (M+p))/M \rfloor \leq \lfloor (a_2 \cdot (M+p))/M \rfloor$. Given $a_1 \leq a_2$, it follows that $a_1 \cdot (M+p) \leq a_2 \cdot (M+p)$. Applying `Lemma_Div_Maintains_GTE` from Appendix A completes the proof for this case.
- **Verification Effectiveness:** This lemma is essential for reasoning about aggregate values in the system, such as total deposits. It provides a formal guarantee that larger initial contributions will always result in equal or larger weighted contributions, a fundamental property for fairness.

D.3. Ultimate Round-Trip Safety for Composite Logic

This section culminates in proving the round-trip safety for the entire chain of time-dependent bonus calculations.

Function D.3.1: Specification for Original Amount Calculation (`CalculateOriginalAmountSpec`)

Let $O_S(w_a, t, \Gamma)$ denote this specification for a weighted amount w_a.

- **Formal Specification:** For $w_a > 0$:

$$O_S(w_a, t, \Gamma) := \begin{cases} w_a & \text{if } F(\Gamma.\text{discount}, t) = \text{None} \\ O_A(w_a, d.p) & \text{if } F(\Gamma.\text{discount}, t) = \text{Some}(d) \end{cases}$$

  where O_A is the `CalculateOriginalAmount` function from Appendix C.

Lemma D.3.2: Monotonicity of Original Amount Calculation (`Lemma_CalculateOriginalAmountSpec_Monotonic`)

- **Formal Specification:**

$$\forall r_1, r_2, t \in \mathbb{N}, \forall \Gamma : (\text{ValidConfig}(\Gamma) \land r_1 \leq r_2) \implies O_S(r_1, t, \Gamma) \leq O_S(r_2, t, \Gamma)$$

- **Description and Verification Strategy:** This lemma proves that the system-level discount reversion function is monotonic. It is the logical dual to `Lemma_CalculateWeightedAmountSpec_Monotonic` and guarantees that reverting a smaller weighted amount cannot yield a larger original amount than reverting a larger weighted amount. This property is an indispensable link in the chain of reasoning for proving refund safety. The proof proceeds by a case analysis on the presence of an active discount:

1. **Case `None`:** The function is an identity, so the property reduces to the precondition $r_1 \leq r_2$.
2. **Case `Some(d)`:** The property reduces to proving $\lfloor (r_1 \cdot M)/(M+p) \rfloor \leq \lfloor (r_2 \cdot M)/(M+p) \rfloor$. Given $r_1 \leq r_2$, it follows that $r_1 \cdot M \leq r_2 \cdot M$. Applying the foundational `Lemma_Div_Maintains_GTE` from Appendix A directly completes the proof.

- **Verification Effectiveness:** This lemma is essential for composing the high-level safety proof of `Lemma_RefundIsSafe`. It allows the verifier to transitively reason about inequalities. Specifically, it enables the proof to carry the bound established on the intermediate `remain` variable (which is a weighted amount) back to the final `refund` variable (which is an original amount), thus completing the deductive chain.

Lemma D.3.3: System-Level Round-Trip Safety with Bounded Loss of At Most One (`Lemma_WeightOriginal_RoundT`

This is a paramount safety property proven within the `Config` module. It ensures that the composite operation of applying and then reverting a time-based discount is non-value-creating and, more importantly, has an extremely small and strictly bounded precision loss.

- **Formal Specification:**

  $\forall a \in \mathbb{N}^+, \forall t \in \mathbb{N}, \forall \Gamma : \text{ValidConfig}(\Gamma) \implies a - 1 \leq O_S(W_S(a, t, \Gamma), t, \Gamma) \leq a$

- **Description and Verification Strategy:** This proof provides one of the strongest guarantees in the system. It confirms that after applying a bonus and then reverting it, the final amount can be at most **one single minimal unit of currency** less than the original. This is achieved by a precise analysis of integer division truncation. The proof again proceeds by case analysis on `F(Γ.discount, t)`:

  1. **Case `None`:** The expression simplifies to `a <= a`, which is trivially true.
  2. **Case `Some(d)`:** The problem is reduced to the round-trip safety of the underlying discount arithmetic primitives. The proof leverages `Lemma_DivMul_Bounds` and `Lemma_DivLowerBound_from_StrictMul` to analyze the expression `floor((floor((a * (M+p))/M) * M) / (M+p))`. It shows that due to the two sequential truncations, the final result can never deviate from `a` by more than 1.

- **Verification Effectiveness:** This lemma represents a significant milestone. It proves an extremely strong and practical property: the financial logic for bonuses is safe and almost perfectly reversible. This guarantee is a critical prerequisite for proving the ultimate refund safety in the `Deposit` module, as it ensures the bonus mechanism itself cannot be a source of value inflation or significant loss in refund calculations.

D.4. Helper Function for Stakeholder Lookup

To support the new functionality of individual vesting claims, a verified helper function for retrieving stakeholder-specific data from the configuration was introduced.

Function D.4.1: Specification for Stakeholder Lookup (`GetStakeholderProportion`)

- **Formal Specification:** `result := GetStakeholderProportion(proportions, account)` The function's postconditions guarantee that:
  1. If the result is `Some(p)`, then `p` is an element of the input `proportions` sequence and `p.account` matches the queried `account`.
  2. If the result is `None`, then no proportion `p` in the `proportions` sequence has `p.account` equal to the queried `account`.
- **Description and Verification Strategy:** This function provides a pure, verifiable specification for searching the list of stakeholder proportions within the configuration. It is implemented as a standard recursive search over a sequence. Dafny's verifier is able to prove its correctness by induction on the length of the `proportions` sequence, confirming that the search is both sound (only returns correct data) and complete (finds the data if it exists).
- **Verification Effectiveness:** By formalizing this lookup, we eliminate a potential source of error in the high-level state transition logic. The `Launchpad` module's `ClaimIndividualVestingSpec` function can now rely on this proven specification to unambiguously retrieve the correct allocation and vesting schedule for a given stakeholder. This ensures that the claim logic is always based on the verifiably correct parameters, preventing one stakeholder from accidentally being assigned another's vesting terms.

Appendix E: Formal Verification of the Deposit State Transition Logic

The `Deposit` module represents the compositional apex of the launchpad's core financial logic. It integrates the verified primitives from `AssetCalculations`, `Discounts`, and `Config` to define a complete, end-to-end specification for the state transition resulting from a user deposit. This module's primary contribution is the formal proof of complex, emergent properties of this integrated workflow, most notably the safety of the refund mechanism in a capped sale. It serves as a testament to the power of layered verification, where the safety of a complex system is derived from the proven safety of its individual components.

E.1. High-Level Specification Functions

The module orchestrates the deposit logic through a hierarchy of specification functions. Let Γ denote a valid configuration (`Config`), $a \in \mathbb{N}^+$ be the deposit amount, $t \in \mathbb{N}$ be the current time, $D_T \in \mathbb{N}$ be the total amount deposited in the contract, and $S_T \in \mathbb{N}$ be the total tokens sold (or total weight).

Function E.1.1: The Deposit Specification Dispatcher (`DepositSpec`)

This function, denoted D_S, acts as a dispatcher based on the sale mechanic defined in the configuration. It returns a tuple (a', w', D'_T, S'_T, r) representing the net amount added to the investment, the weight/assets added, the new total deposited, the new total sold, and the refund amount.

- **Formal Specification:**

$$D_S(\Gamma, a, D_T, S_T, t) := \begin{cases} D_{FP}(\Gamma, a, D_T, S_T, t) & \text{if } \Gamma.\text{mechanic.FixedPrice?} \\ D_{PD}(\Gamma, a, D_T, S_T, t) & \text{if } \Gamma.\text{mechanic.PriceDiscovery?} \end{cases}$$

  where D_{FP} and D_{PD} are the specifications for fixed-price and price-discovery deposits, respectively.

E.2. Verification of the Fixed-Price Deposit Workflow

The most complex logic resides in the fixed-price sale scenario, which involves a hard cap on the number of tokens to be sold ($\Gamma.saleAmount$).

Function E.2.1: The Fixed-Price Deposit Specification (`DepositFixedPriceSpec`)

Let this function be denoted D_{FP}. It models the entire workflow, including potential refunds. Let d_T and s_T be $\Gamma.mechanic.depositTokenAmount$ and $\Gamma.mechanic.saleTokenAmount$.

1. **Weighted Amount Calculation:** First, the initial deposit a is adjusted for any active time-based discounts. $w := W_S(a, t, \Gamma)$ (using the weighted amount spec from Appendix D).
2. **Asset Conversion:** The weighted amount w is converted into sale assets. $assets := C(w, d_T, s_T)$ (using the asset conversion spec from Appendix B).
3. **Cap Check:** The potential new total of sold tokens is calculated: $S'_{T,\text{potential}} := S_T + assets$.
4. **State Transition Logic:** The final state is determined by comparing this potential total to the sale cap.

- **Formal Specification:**

$$\begin{gathered} D_{FP}(\Gamma, a, D_T, S_T, t) := \\ \text{if } (S_T + C(W_S(a, t, \Gamma), d_T, s_T) \leq \Gamma.\text{saleAmount}) \text{ then} \\ (a, C(W_S(a, t, \Gamma), d_T, s_T), D_T + a, S_T + C(W_S(a, t, \Gamma), d_T, s_T), 0) \\ \text{else} \\ (a - r, \Gamma.\text{saleAmount} - S_T, D_T + (a - r), \Gamma.\text{saleAmount}, r) \\ \text{where } r = R_F(\Gamma, a, S_T, t, d_T, s_T) \end{gathered}$$

Function E.2.2: The Refund Calculation Specification (`CalculateRefundSpec`)

This helper function, R_F, isolates the complex refund calculation logic.

- **Formal Specification:** Let $w := W_S(a, t, \Gamma)$ and $assets := C(w, d_T, s_T)$. Let $assets_{excess} := (S_T + assets) - \Gamma.\text{saleAmount}$. Let $remain := R(assets_{excess}, d_T, s_T)$ (reverse conversion of the excess).

$$R_F(...) := O_S(remain, t, \Gamma)$$

  (original amount of the reverted excess, from Appendix D).

E.3. Verification of Amount Conservation (`Lemma_DepositFixedPrice_AmountConservation`)

While `Lemma_RefundIsSafe` provides the crucial upper bound on the refund, this lemma proves a different, but equally important property: the exact conservation of funds from the user's perspective during a deposit that triggers a refund.

- **Formal Specification:** Let $(a', w', D'_T, S'_T, r) := D_{FP}(\Gamma, a, D_T, S_T, t, d_T, s_T)$.

  $\forall \Gamma, a, D_T, S_T, t, d_T, s_T : (\text{ValidConfig}(\Gamma) \land a > 0 \land d_T > 0 \land s_T > 0 \land S_T < \Gamma.\text{saleAmount}) \implies a' + r = a$

- **Description and Verification Strategy:** This lemma formally proves that the user's initial deposit (`amount`) is perfectly accounted for, being split exactly between the portion retained by the contract (`newAmount`, denoted `a'`) and the portion returned to the user (`refund`, denoted `r`). This is a stronger guarantee than `refund <= amount`, as it proves that no funds are created or destroyed in the transaction. The proof is a direct consequence of the specification of `DepositFixedPriceSpec`. In the case where a refund is issued, the new amount retained by the contract is explicitly defined as `amount - refund`. The verifier can therefore trivially prove that `(amount - refund) + refund == amount`.

- **Verification Effectiveness:** This lemma provides a formal guarantee against "dust" funds being lost or unaccounted for due to off-by-one errors or incorrect arithmetic in the deposit logic. It ensures that the accounting for each deposit transaction is perfectly balanced.

E.4. The Ultimate Safety Property: `Lemma_RefundIsSafe`

This is the most critical safety property of the entire financial system. It provides a mathematical guarantee that the calculated refund amount can never exceed the user's original deposit amount, preventing a catastrophic class of bugs where the contract could be drained of funds.

- **Formal Specification:**

$$\begin{aligned} &\forall \Gamma, a, w, \text{assets}, \text{assets}_{excess}, t, d_T, s_T : \\ &\text{ValidConfig}(\Gamma) \land a > 0 \land w > 0 \land d_T > 0 \land s_T > 0 \land \\ &\Big(\begin{array}{l} w = W_S(a, t, \Gamma) \land \\ assets = C(w, d_T, s_T) \land \\ assets_{excess} \le assets \end{array} \Big) \implies \\ &\qquad O_S(R(assets_{excess}, d_T, s_T), t, \Gamma) \le a \end{aligned}$$

- **Description and Verification Strategy:** The proof of this lemma is a masterful demonstration of compositional verification. It does not attempt to prove the property from first principles but instead constructs a deductive chain using previously verified lemmas from other modules. The chain of reasoning within the Dafny proof directly mirrors the following steps:

  1. **Define `remain`:** Let $remain := R(assets_{excess}, d_T, s_T)$. The goal is to prove $O_S(remain, t, \Gamma) \le a$.
  2. **Bound `remain` using Asset Reversion Monotonicity:** From the precondition $assets_{excess} \le assets$, the proof applies `Lemma_CalculateAssetsRevertSpec_Monotonic` (from Appendix B). This yields the inequality: $R(assets_{excess}, d_T, s_T) \le R(assets, d_T, s_T)$, and therefore $remain \le R(assets, d_T, s_T)$.
  3. **Bound `R(assets, ...)` using Asset Round-Trip Safety:** `assets` is defined as $C(w, d_T, s_T)$. The proof then invokes `Lemma_AssetsRevert_RoundTrip_bounds` (from Appendix B), which guarantees that $R(C(w, d_T, s_T), d_T, s_T) \le w$. This gives us the crucial intermediate bound: $R(assets, d_T, s_T) \le w$.
  4. **Establish Transitive Bound on `remain`:** By combining steps 2 and 3, the proof establishes the transitive inequality: $remain \le R(assets, d_T, s_T) \le w \implies remain \le w$.
  5. **Apply Monotonicity of Original Amount Calculation:** With the inequality $remain \le w$ established, the proof applies `Lemma_CalculateOriginalAmountSpec_Monotonic` (from Appendix D). This allows the reasoning to be lifted from the domain of "weighted" amounts to "original" amounts, yielding: $O_S(remain, t, \Gamma) \le O_S(w, t, \Gamma)$.
  6. **Bound `O_S(w, ...)` using Discount Round-Trip Safety:** `w` is defined as $W_S(a, t, \Gamma)$. The proof invokes the powerful `Lemma_WeightOriginal_RoundTrip_bounds` (from Appendix D), which states that $O_S(W_S(a, t, \Gamma), t, \Gamma) \le a$. This provides the final link in the chain: $O_S(w, t, \Gamma) \le a$.

7. **Final Conclusion:** By combining steps 5 and 6, the proof arrives at the final transitive inequality, completing the demonstration: $O_S(remain, t, \Gamma) \leq O_S(w, t, \Gamma) \leq a \implies O_S(remain, t, \Gamma) \leq a$.

- **Verification Effectiveness:** The proof of `Lemma_RefundIsSafe` is the capstone of this verification effort. It demonstrates that the system is safe from a critical financial vulnerability *by construction*. The safety is not an accidental property but an inevitable consequence of composing components, each of which has been independently proven to be safe (monotonic and non-value-creating on round-trips with bounded loss). This layered, compositional approach provides an exceptionally high degree of confidence in the correctness of the entire deposit workflow [22].

Appendix F: Verification of the Global State Machine and System Synthesis

The `Launchpad` module represents the final and outermost layer of the system's formal specification. It encapsulates the entire state of the smart contract within a single immutable data structure and defines the valid state transitions that govern its lifecycle. This module does not introduce new financial primitives; instead, its critical function is to **orchestrate** the verified components from the lower-level modules (`Deposit`, `Config`, etc.). The verification at this level ensures that the global state is managed correctly and that the complex, pre-verified workflows are integrated into the state machine in a sound and secure manner.

F.1. The Global State Representation

The complete state of the contract at any point in time is represented by the datatype `AuroraLaunchpadContract`, denoted here by the symbol Σ.

- **Formal Specification:** The state Σ is a tuple containing all dynamic and static data of the contract:

$$\Sigma := (\Gamma, D_T, S_T, f_{set}, f_{lock}, \mathcal{A}, N_p, \mathcal{J})$$

  where:
  - Γ: The `Config` structure, containing all static sale parameters (as defined in Appendix D).
  - $D_T \in \mathbb{N}$: `totalDeposited`, the aggregate principal deposited by all participants.
  - $S_T \in \mathbb{N}$: `totalSoldTokens`, the aggregate tokens sold or total weight accumulated.
  - $f_{set} \in \{\text{true}, \text{false}\}$: `isSaleTokenSet`, a flag indicating contract initialization.
  - $f_{lock} \in \{\text{true}, \text{false}\}$: `isLocked`, a flag indicating if the contract is administratively locked.
  - $\mathcal{A}$: A map AccountId $\to$ IntentAccount, linking external account identifiers to internal ones.
  - $N_p \in \mathbb{N}$: `participantsCount`, the number of unique investors.
  - $\mathcal{J}$: The map IntentAccount $\to$ InvestmentAmount, storing the detailed investment record for each participant.
- **Top-Level Invariant (`Valid`):** The fundamental invariant of the global state is that its embedded configuration must be valid.

$$\text{Valid}(\Sigma) \iff \text{ValidConfig}(\Gamma)$$

F.2. The State Machine Logic: Observing the State

The `GetStatus` function provides a pure, observable interpretation of the contract's state Σ at a given time t. Let $S(\Sigma, t)$ denote the status function.

- **Formal Specification:**

$$S(\Sigma, t) := \begin{cases} \text{NotInitialized} & \text{if } \neg\Sigma.f_{set} \\ \text{Locked} & \text{if } \Sigma.f_{lock} \\ \text{NotStarted} & \text{if } t < \Gamma.\text{startDate} \\ \text{Ongoing} & \text{if } \Gamma.\text{startDate} \leq t < \Gamma.\text{endDate} \\ \text{Success} & \text{if } t \geq \Gamma.\text{endDate} \wedge \Sigma.D_T \geq \Gamma.\text{softCap} \\ \text{Failed} & \text{if } t \geq \Gamma.\text{endDate} \wedge \Sigma.D_T < \Gamma.\text{softCap} \end{cases}$$

- **Helper Predicates:** For clarity, we define helper predicates (e.g., $IsOngoing(\Sigma, t)$) as $S(\Sigma, t) == Ongoing$.

F.3. Properties of the State Machine

The verification of this module includes proofs about the logical integrity of the state machine itself, ensuring its behavior is predictable and consistent over time [23].

- **Lemma F.3.1: Temporal Progression (`Lemma_StatusTimeMovesForward`)** This lemma proves that the state machine cannot move backward in time.

$$\forall t_1, t_2 \in \mathbb{N}, \forall \Sigma : (\text{Valid}(\Sigma) \land t_1 \leq t_2) \implies (\text{IsOngoing}(\Sigma, t_1) \land t_2 < \Gamma.endDate \implies \text{IsOngoing}(\Sigma, t_2))$$

- **Lemma F.3.2: Mutual Exclusion of States (`Lemma_StatusIsMutuallyExclusive`)** This proves that the contract cannot simultaneously be in two conflicting states.

$$\forall t \in \mathbb{N}, \forall \Sigma : \text{Valid}(\Sigma) \implies \neg(\text{IsOngoing}(\Sigma, t) \land \text{IsSuccess}(\Sigma, t))$$

- **Lemma F.3.3: Terminal Nature of Final States (`Lemma_StatusFinalStatesAreTerminal`)** This proves that once a final state (`Success`, `Failed`, `Locked`) is reached, it is permanent [24].

$$\forall t_1, t_2 \in \mathbb{N}, \forall \Sigma : (\text{Valid}(\Sigma) \land t_1 \leq t_2) \implies (\text{IsSuccess}(\Sigma, t_1) \implies \text{IsSuccess}(\Sigma, t_2))$$

F.4. The State Transition Functions

The heart of the module is the set of pure functions modeling the contract's dynamic behavior. Each function defines how the global state Σ transitions to a new state Σ' in response to an action.

F.4.1. Deposit Transition (`DepositSpec`) This function defines the state transition for a user deposit.

- **Formal Specification:** $(\Sigma', a', w', r) := T_{deposit}(\Sigma, \text{accId}, a, \text{intAcc}, t)$
  - **Preconditions (`requires`):** The function requires that the contract's global state is valid (`Valid()`). For user deposits, it strictly enforces that the transaction must occur during the `Ongoing` phase (`IsOngoing(t)`).
  - **Postconditions (`ensures`):** The specification guarantees that the new state Σ' is constructed by correctly updating the old state Σ with the results from the pre-verified sub-workflow:
    - The returned values (a', w', r) must exactly match the output of `Deposit.DepositSpec(\Gamma, a, \Sigma.D_T, \Sigma.S_T, t)`.
    - The new global totals must be updated correctly: $\Sigma'.D_T = \Sigma.D_T + a'$ and $\Sigma'.S_T = \Sigma.S_T + w'$.
    - The user's individual investment record in the `investments` map is updated by adding `a'` and `w'` to their previous balance.
- **Description and Verification Strategy:** This function acts as a verified gatekeeper and orchestrator for deposits. It uses the `GetStatus` function to enforce the time-based business rule for when deposits are allowed. It then delegates all complex financial calculations to the `Deposit` module, whose correctness (including refund safety) is already established. The verification at this layer focuses on proving that the global state is updated consistently with the results of this delegation.
- **Verification Effectiveness:** This compositional proof is remarkably efficient. It ensures that the safe, low-level deposit logic is correctly integrated into the global state machine, preventing state corruption or the bypassing of business rules (e.g., depositing before the sale starts).

F.4.2. Withdrawal Transition (`WithdrawSpec`) This function specifies the state transition for a user withdrawal.

- **Formal Specification:** $\Sigma' := T_{withdraw}(\Sigma, \text{intAcc}, a, t)$
  - **Preconditions (`requires`):** The function's preconditions are strict. A withdrawal is only permitted in specific contract states: `Failed`, `Locked`, or `Ongoing` for a `PriceDiscovery` sale. It also requires that the `intentAccount` has an existing investment and that the withdrawal `amount` is valid for the given sale mechanic (e.g., must be the full amount for a `FixedPrice` withdrawal).
  - **Postconditions (`ensures`):** The specification guarantees that the new state Σ' is the result of applying the changes computed by the `Withdraw` module. Let $(inv', sold') := W.WithdrawSpec(\Gamma, \Sigma.\mathcal{J}[\text{intAcc}], a, \Sigma.S_T, t)$. Then:
    - $\Sigma'.D_T = \Sigma.D_T - a$

  - $\Sigma'.S_T = sold'$
  - $\Sigma'.\mathcal{J}[\text{intAcc}] = inv'$
- **Description and Verification Strategy:** This function orchestrates the withdrawal process. It first acts as a guard, using `GetStatus` to ensure the contract is in a state that permits withdrawals. It then invokes the verified `Withdraw.WithdrawSpec` function to compute the new state of the user's investment and the new global total of sold tokens. Finally, it constructs the new global state Σ' by applying these computed changes.
- **Verification Effectiveness:** This proof formally guarantees that withdrawals are handled safely and correctly at the contract level. It prevents invalid withdrawal attempts (e.g., a user trying to withdraw from a successful sale) and ensures that the contract's global accounting (`totalDeposited`, `totalSoldTokens`) remains perfectly consistent with the changes in individual user investments.

F.4.3. Public Sale Claim Transition (`ClaimSpec`) This function defines the state transition for a public participant claiming their vested tokens.

- **Formal Specification:** $\Sigma' := T_{claim}(\Sigma, \text{intAcc}, t)$
  - **Preconditions (`requires`):** This action is heavily guarded. It requires that the sale has concluded successfully (`IsSuccess(t)`). Crucially, it also requires that the amount available to claim is strictly greater than the amount already claimed (`available > investment.claimed`), ensuring the transaction is meaningful.
  - **Postconditions (`ensures`):** The specification guarantees that the user's `InvestmentAmount` record is updated correctly. Let $assets_{claim} :=$ Claim.AvailableForClaimSpec(...) $-\ \Sigma.\mathcal{J}[\text{intAcc}]$.claimed. Then the new investment record is $\Sigma'.\mathcal{J}[\text{intAcc}] = \Sigma.\mathcal{J}[\text{intAcc}]$.AddToClaimed($assets_{claim}$). All other parts of the state remain unchanged.
- **Description and Verification Strategy:** The function orchestrates the claim process by first using `GetStatus` to enforce the "successful sale" business rule. It delegates the complex calculation of vested assets to the pre-verified `Claim.AvailableForClaimSpec` function. The core of the state transition is the update to the user's `claimed` balance in the `investments` map.
- **Verification Effectiveness:** This proof ensures that the token claim mechanism is robust and secure. It formally proves that users can only claim what they are entitled to based on the verified allocation and vesting logic. It prevents critical bugs such as claiming tokens before the sale has ended, claiming more tokens than allocated, or re-claiming tokens that have already been distributed.

F.4.4. Individual Vesting Claim Transition (`ClaimIndividualVestingSpec`) This function specifies the claim process for private stakeholders with individual vesting schedules.

- **Formal Specification:** $\Sigma' := T_{claim\_indiv}(\Sigma, \text{intAcc}, t)$
  - **Preconditions (`requires`):** Similar to the public claim, this requires `IsSuccess(t)`. It also requires that the `intentAccount` corresponds to a valid stakeholder (provably found via `Config.GetStakeholderProportion`). Finally, it enforces that the available amount is greater than what has been claimed.
  - **Postconditions (`ensures`):** The specification guarantees that the `individualVestingClaimed` map is correctly updated for the given `intentAccount` with the new total claimed amount, calculated by delegating to `Claim.AvailableForIndividualVestingClaimSpec`.
- **Description and Verification Strategy:** This function mirrors the logic of the public claim but operates on a separate part of the state (`individualVestingClaimed` map) and uses a different set of configuration parameters (the stakeholder's private vesting schedule). It relies on the verified `GetStakeholderProportion` helper to safely retrieve the correct parameters.
- **Verification Effectiveness:** This proof guarantees the correct and secure operation of the private stakeholder claim process. It ensures a clean separation of concerns, preventing a public participant from interfering with a stakeholder's allocation. It formally proves that each stakeholder's unique vesting schedule is applied correctly and unambiguously.

F.4.5. Token Distribution Transition (`DistributeTokensSpec`) This function defines the administrative state transition for distributing tokens to stakeholders.

- **Formal Specification:** $\Sigma' := T_{distribute}(\Sigma, t)$

  - **Preconditions (`requires`):** This administrative action requires that the sale is in a `Success` state and that there are stakeholders pending distribution (`|Distribution.GetFilteredDistributionsSpec(...) > 0`).
  - **Postconditions (`ensures`):** The specification guarantees that the new list of paid accounts is the old list appended with the list of newly eligible accounts: Σ'.distributedAccounts = Σ.distributedAccounts + + Distribution.GetFilteredDistributionsSpec(Γ, Σ.distributedAccounts).
- **Description and Verification Strategy:** This function models the batch processing of stakeholder payments. It uses preconditions as safety gates for the administrative action. The core logic is delegated to the pre-verified `Distribution` module to compute the list of stakeholders to be paid in the current batch. The state transition is a simple append operation on the `distributedAccounts` sequence.
- **Verification Effectiveness:** This proof provides a formal guarantee that the administrative distribution process is sound and complete. It relies on the pre-verified properties of the `Distribution` module to ensure no stakeholder is paid twice or omitted, and it enforces the high-level business rule that this action can only occur after a successful sale.

F.5. Grand Synthesis and Overall Analysis

The formal verification of the `Launchpad` module completes a hierarchical proof structure, providing end-to-end formal assurance for the system's entire lifecycle. The layers of this structure can be summarized as follows:

1. **Layer 1: Axiomatic Foundation (Appendix A - `MathLemmas`)**: Established the fundamental, non-linear properties of integer arithmetic, including strict bounds on truncation loss.
2. **Layer 2: Financial Primitives (Appendix B, C - `AssetCalculations`, `Discounts`)**: Built upon the axioms to prove the safety and correctness of isolated financial operations. Key properties included monotonicity and bounded round-trip safety.
3. **Layer 3: Composite Workflows (Appendix D, E, G, H, I - `Config`, `Deposit`, `Withdraw`, `Claim`, `Distribution`)**: Formalized the complete business logic for all user- and system-level interactions. This layer establishes end-to-end safety guarantees for every phase of the contract's lifecycle: from the atomicity and fund conservation of initial deposits** (`Lemma_RefundIsSafe`), through the accounting integrity of **withdrawals**, to the temporal correctness of post-sale **token claims** (vesting monotonicity) and the logical soundness of administrative **distributions**.
4. **Layer 4: Global State Machine (Appendix F - `Launchpad`)**: Integrated all verified workflows into a global state machine, proving that the orchestration logic correctly and safely manages the system's overall state across its entire lifecycle.

This hierarchical decomposition provides a robust and scalable methodology for verifying mission-critical systems. The final safety properties of the `Launchpad` contract are a logical and inevitable consequence of the verified properties of its constituent parts, culminating in a system with the highest possible degree of formal assurance against a wide class of vulnerabilities, spanning low-level financial arithmetic, complex business rule interactions, and the complete state machine lifecycle.

Appendix G: Formal Verification of the Withdrawal Workflow

The `Withdraw` module provides the formal specification for the user withdrawal workflow, serving as a logical counterpart to the `Deposit` module. It defines the pure, mathematical behavior for withdrawals, guaranteeing that state changes—such as decrementing `totalDeposited` and `totalSoldTokens`—are handled safely and predictably under different sale mechanics. Its verification is critical for ensuring that funds can be safely returned to users in edge-case scenarios like a failed sale, without compromising the contract's accounting integrity.

G.1. Specification Dispatcher (`WithdrawSpec`)

The top-level function `WithdrawSpec` acts as a verified router, dispatching the withdrawal logic to the appropriate sub-specification based on the sale mechanic defined in the configuration.

- **Formal Specification:** Let `inv` be the user's `InvestmentAmount`.

$$(\text{inv}', \text{sold}') := \text{WithdrawSpec}(\Gamma, \text{inv}, a, S_T, t)$$

  The postconditions guarantee that the result tuple is exactly equal to the result of either `WithdrawFixedPriceSpec` or `WithdrawPriceDiscoverySpec`, depending on `Γ.mechanic`.

- **Description and Verification Strategy:** This function enforces the top-level preconditions for any withdrawal, such as ensuring the withdrawal amount is positive and that the requested amount is valid for the given sale type (e.g., `amount == inv.amount` for Fixed Price). By acting as a simple, pure dispatcher, its correctness is straightforward for the verifier to confirm, ensuring that the complex logic is always routed to the correct, formally verified implementation.

G.2. Fixed-Price Withdrawal (`WithdrawFixedPriceSpec`)

This function models a complete, "all-or-nothing" withdrawal, which is the required behavior if a sale fails to meet its `softCap` and must be cancelled.

- **Formal Specification:** Let `inv` be the user's `InvestmentAmount`.

$$(\text{inv}', \text{sold}') := \text{WithdrawFixedPriceSpec}(\text{inv}, a, S_T)$$

  The specification guarantees:
  - $\text{inv}'.\text{amount} = 0 \land \text{inv}'.\text{weight} = 0$
  - $\text{inv}'.\text{claimed} = \text{inv.claimed}$
  - $\text{sold}' = S_T - \text{inv.weight}$
- **Description and Verification Strategy:** The logic enforces that the user must withdraw their entire deposit (`amount == inv.amount`). The postconditions guarantee a clean and complete removal of the user's record: their `amount` and `weight` are zeroed out, and the global `totalSoldTokens` is decremented by their full original `weight`.
- **Verification Effectiveness:** This proof provides a formal guarantee of atomicity for cancellation events. It ensures that a withdrawing user's contribution is completely erased from the contract's financial state, preventing scenarios where a user could withdraw their principal but leave behind "ghost" weight that would incorrectly affect the allocations for remaining participants.

G.3. Price-Discovery Withdrawal (`WithdrawPriceDiscoverySpec`)

This function models a more complex partial or full withdrawal during an ongoing `PriceDiscovery` sale, where a user's relative share of the pool is dynamic.

- **Formal Specification:** Let `inv` be the user's `InvestmentAmount` and $a' = \text{inv.amount} - a$. Let $w_{recalc} = W_S(a', t, \Gamma)$.

$$(\text{inv}', \text{sold}') := \text{WithdrawPriceDiscoverySpec}(\Gamma, \text{inv}, a, S_T, t)$$

  The specification guarantees:
  - $\text{inv}'.\text{amount} = a'$
  - $\text{inv}'.\text{weight} = \min(\text{inv.weight}, w_{recalc})$
  - $\text{sold}' = S_T - (\text{inv.weight} - \text{inv}'.\text{weight})$
- **Description and Verification Strategy:** This workflow is significantly more complex because a partial withdrawal requires re-evaluating the user's contribution. The logic is as follows:
  1. The new principal amount (`newAmount`) is calculated.
  2. This `newAmount` is passed to `CalculateWeightedAmountSpec` to determine the user's `recalculatedWeight` at the current `time` (as active discounts may have changed since their last deposit).
  3. The global `totalSoldTokens` is then reduced by the precise *difference* between the user's old and new weight.
- **Verification Effectiveness:** The verification of this specification is critical for the integrity of a price discovery sale. It formally proves that the `totalSoldTokens`, which serves as the denominator in the final price calculation, is always an accurate reflection of the total weighted contributions currently in the contract. This prevents exploits where a user could deposit during a high-bonus period, then withdraw their principal after the bonus expires, while leaving an inflated weight in the system, unfairly diluting other participants.

Appendix H: Formal Verification of Token Claim and Vesting Logic

The `Claim` module formalizes the entire post-sale workflow, defining the logic for calculating users' final token allocations and managing their release according to vesting schedules. The verification of this module provides mathematical guarantees that the final distribution of tokens is fair, predictable, and strictly adheres to the sale's predefined rules.

H.1. User Token Allocation Logic

This section details the specification and verification of how a user's final token entitlement is calculated.

H.1.1. Specification of Total Allocation (`UserAllocationSpec`) This function is the single source of truth for determining a user's total token entitlement based on the sale's outcome.

- **Formal Specification:**

$$\text{UserAllocationSpec}(w, S_T, \Gamma) := \begin{cases} w & \text{if } \Gamma.\text{mechanic.FixedPrice?} \\ \lfloor (w \cdot \Gamma.\text{saleAmount})/S_T \rfloor & \text{if } \Gamma.\text{mechanic.PriceDiscovery?} \end{cases}$$

- **Description and Verification Strategy:** The function's behavior is defined by the sale mechanic. For a `FixedPrice` sale, the `weight` a user accumulates is their final token allocation. For a `PriceDiscovery` sale, the allocation is calculated proportionally based on the user's share of the final `totalSoldTokens`. The preconditions `S_T > 0` and `w <= S_T` ensure the calculation is well-defined and prevents division-by-zero errors.

H.1.2. Verification of Allocation Properties (`Lemma_UserAllocationSpec`) This lemma proves key mathematical properties of the `UserAllocationSpec` function, providing the SMT solver with essential, non-trivial insights into the non-linear arithmetic involved.

- **Formal Specification:** The lemma proves several properties, including:
  - $w \leq S_T \implies \text{UserAllocationSpec}(w, S_T, \Gamma) \leq \Gamma.\text{saleAmount}$
  - $\Gamma.\text{saleAmount} \leq S_T \implies \text{UserAllocationSpec}(w, S_T, \Gamma) \leq w$
- **Description and Verification Strategy:** The proof relies on direct instantiation of the foundational lemmas from `MathLemmas`. For instance, to prove that a user's allocation cannot exceed the total `saleAmount`, the proof invokes `Lemma_MulDivLess_From_Scratch`, as the precondition `w <= S_T` satisfies the lemma's requirements.
- **Verification Effectiveness:** This lemma is essential for any higher-level proof that reasons about aggregate allocations. It provides the verifier with trusted "axioms" about the `UserAllocationSpec` formula, guaranteeing that the sum of all allocations will not exceed the sale cap and that the allocation behaves predictably relative to the user's contribution.

H.2. Vesting Calculation Logic

This section formalizes the shared logic for time-based token release.

H.2.1. Specification of the Vesting Curve (`CalculateVestingSpec`) This function specifies the vesting logic, which is used for both the main public sale and individual stakeholder schedules.

- **Formal Specification:** Let t_s be `vestingStart` and v_s be `vestingSchedule`.

$$\text{CalculateVestingSpec}(A, t_s, t, v_s) := \begin{cases} 0 & \text{if } t < t_s + v_s.\text{cliffPeriod} \\ A & \text{if } t \geq t_s + v_s.\text{vestingPeriod} \\ \lfloor (A \cdot (t - t_s))/v_s.\text{vestingPeriod} \rfloor & \text{otherwise} \end{cases}$$

  where A is the total number of assets to be vested.
- **Description and Verification Strategy:** The specification models a standard piecewise vesting curve: 0 tokens are released before the cliff, the full amount is released after the vesting period, and a linear interpolation is used in between. The accompanying `Lemma_CalculateVestingSpec_Properties` lemma formally proves that the result of this function never exceeds the total assets `A`, a safety property derived by instantiating `Lemma_MulDivLess_From_Scratch`.

H.2.2. The Monotonicity of Vesting (`Lemma_CalculateVestingSpec_Monotonic`) This is the most critical safety and liveness property of the vesting logic.

- **Formal Specification:**

$$\forall A, t_s, v_s, t_1, t_2 : (t_1 \leq t_2) \implies \text{CalculateVestingSpec}(A, t_s, t_1, v_s) \leq \text{CalculateVestingSpec}(A, t_s, t_2, v_s)$$

- **Description and Verification Strategy:** This lemma guarantees that as time moves forward, a user's vested (and therefore claimable) amount can only increase or stay the same; it can never decrease. The proof proceeds by a comprehensive case analysis on the positions of `t1` and `t2` relative to the `cliffEnd` and `vestingEnd` timestamps. In the most complex case (where both `t1` and `t2` are within the linear vesting period), the proof is completed by applying `Lemma_Div_Maintains_GTE`.
- **Verification Effectiveness:** This lemma provides a formal guarantee against a critical class of bugs where a user could lose access to tokens they were previously entitled to. It ensures the vesting process is predictable and fair, which is essential for user trust in the system.

H.3. Composite Claim Logic (`AvailableForClaimSpec`)

Finally, the `AvailableForClaimSpec` function composes the verified allocation and vesting components to define the end-to-end logic for determining a user's claimable balance at any given time. Its correctness is not proven from first principles but is a direct and inevitable consequence of the proven properties of the functions it orchestrates.

Appendix I: Formal Verification of Post-Sale Distribution Logic

The `Distribution` module formalizes the administrative task of calculating the ordered list of project stakeholders eligible for token distribution after a successful sale. Unlike modules focused on financial calculations, this module's primary concern is the correctness of list and set manipulations. Its verification ensures that the process for identifying who to pay next is deterministic, auditable, and free from logical errors such as paying a stakeholder twice or omitting them entirely.

I.1. The Core Filtering Logic (`FilterDistributedStakeholders`)

This function provides the core mechanism for identifying which stakeholders are still pending payment. It implements a verified set difference operation on ordered sequences.

- **Formal Specification:** Let $\mathcal{P}$ be the sequence of `StakeholderProportion` from the configuration and $\mathcal{D}_{acc}$ be the sequence of already distributed `IntentAccount`s. Let $\mathcal{P}_{acc}$ be the sequence of accounts extracted from $\mathcal{P}$. The function `FilterDistributedStakeholders` produces a result sequence $\mathcal{R}$ with the following formally proven properties:
  1. **Correctness (Set Difference):** The set of accounts in $\mathcal{R}$ is exactly the set of accounts in $\mathcal{P}_{acc}$ minus the set of accounts in $\mathcal{D}_{acc}$.
  $$\{a \mid a \in \mathcal{R}\} = \{p.\text{account} \mid p \in \mathcal{P}\} \setminus \{a \mid a \in \mathcal{D}_{acc}\}$$
  2. **Soundness:** Every account in the result list $\mathcal{R}$ is guaranteed to be a valid stakeholder who has not yet been paid.
  $$\forall a \in \mathcal{R} \implies (\exists p \in \mathcal{P} : p.\text{account} = a) \land (a \notin \mathcal{D}_{acc})$$
  3. **Completeness:** Every valid stakeholder who has not yet been paid is guaranteed to be in the result list $\mathcal{R}$.
  $$(\forall p \in \mathcal{P} : p.\text{account} \notin \mathcal{D}_{acc}) \implies p.\text{account} \in \mathcal{R}$$
  4. **Uniqueness Preservation:** If the initial list of stakeholder accounts $\mathcal{P}_{acc}$ is unique, the resulting list $\mathcal{R}$ is also guaranteed to be unique.
- **Description and Verification Strategy:** The function is implemented using recursion on the `proportions` sequence. At each step, it checks if the head of the list is present in the `distributed` sequence. If it is not, the account is prepended to the result of the recursive call on the tail of the list. Dafny proves the extensive postconditions for this function by induction. The set-based specification is particularly powerful, as it abstracts away the details of the sequence implementation and proves the function's behavior at a higher, more intuitive mathematical level.
- **Verification Effectiveness:** This verification provides a rock-solid guarantee against common and critical bugs in administrative processes. It formally proves that **no stakeholder will ever be paid twice** (due to the soundness property) and that **no eligible stakeholder will ever be accidentally omitted** (due to the completeness property). This ensures the operational integrity of the distribution phase.

I.2. Composing the Final Distribution List (`GetFilteredDistributionsSpec`)

This top-level function composes the core filtering logic with the business rule that the `solver` account has a distinct identity and priority in the distribution list.

- **Formal Specification:** Let Γ be the configuration, and $\mathcal{D}_{acc}$ be the sequence of distributed accounts.

$$\text{GetFilteredDistributionsSpec}(\Gamma, \mathcal{D}_{acc}) :=$$

$$\begin{cases} \text{FilterDistributedStakeholders}(\Gamma.\text{props}, \mathcal{D}_{acc}) & \text{if } \Gamma.\text{solver} \in \mathcal{D}_{acc} \\ [\Gamma.\text{solver}] \ +\!+ \ \text{FilterDistributedStakeholders}(\Gamma.\text{props}, \mathcal{D}_{acc}) & \text{if } \Gamma.\text{solver} \notin \mathcal{D}_{acc} \end{cases}$$

  where `++` denotes sequence concatenation.

- **Description and Verification Strategy:** This function acts as a pure specification that orchestrates the final list construction. It first checks if the solver has been paid. If not, the solver's account is placed at the head of the distribution queue. It then invokes the pre-verified `FilterDistributedStakeholders` function to compute the remainder of the queue. The verification at this level is compositional: given the proven contract of `FilterDistributedStakeholders`, Dafny simply proves that this `if/then/else` composition correctly implements the intended logic.

- **Verification Effectiveness:** This demonstrates the power of layered verification. We do not need to re-prove the complex set-difference properties. We trust the verified specification of the lower-level function and only prove the correctness of the orchestration logic. This provides a formal guarantee that the business rule regarding the solver's priority is always correctly and safely applied, ensuring the distribution order is predictable and auditable.

-

References

[1] M. Borkowski, M. Sidorowicz, and A. Szałas, "A Formal Methodology for Proving Security of Financial Systems," in *International conference on formal methods and software engineering (ICFEM)*, in Lecture notes in computer science, vol. 4260. Springer, 2006, pp. 409–424. doi: 10.1007/11901433_27.

[2] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *Principles of security and trust (POST)*, in Lecture notes in computer science, vol. 10204. Springer, 2017, pp. 164–186. doi: 10.1007/978-3-662-54455-6_8.

[3] C. E. Weir and M. Calder, "A Formal Model and Analysis of the DAO Exploit," in *International conference on formal engineering methods (ICFEM)*, in Lecture notes in computer science, vol. 11232. Springer, 2018, pp. 446–462. doi: 10.1007/978-3-030-02441-4_28.

[4] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal Methods: Practice and Experience," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, pp. 1–36, 2009, doi: 10.1145/1592434.1592436.

[5] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996, doi: 10.1145/242223.242257.

[6] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Logic for programming, artificial intelligence, and reasoning (LPAR-16)*, in Lecture notes in computer science, vol. 6355. Springer, 2010, pp. 348–370. doi: 10.1007/978-3-642-16242-8_25.

[7] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and algorithms for the construction and analysis of systems (TACAS)*, in Lecture notes in computer science, vol. 4963. Springer, 2008, pp. 337–340. doi: 10.1007/978-3-540-78800-3_24.

[8] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. de Sousa, "Compositional Verification of Security Protocols," *Formal Aspects of Computing*, vol. 19, no. 2, pp. 217–250, 2007, doi: 10.1007/s00165-006-0010-0.

[9] J. Chen, R. Gu, and J. Vautard, "Compositional Verification of Smart Contracts with Objects and Callbacks," in *Proceedings of the 42nd ACM SIGPLAN conference on programming language design and implementation (PLDI)*, ACM, 2021, pp. 883–898. doi: 10.1145/3453483.3454086.

[10] K. R. M. Leino and R. Monahan, "A Tutorial on the Verifying Compiler Pattern," *Formal Aspects of Computing*, vol. 29, no. 4, pp. 563–583, 2017, doi: 10.1007/s00165-016-0402-1.

[11] D. Monniaux, "The Pitfalls of Verifying Floating-Point Computations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 3, pp. 1–41, 2008, doi: 10.1145/1353445.1353446.

[12] G. Audemard and B. Dutertre, "Certified Verification of Integer Arithmetic," in *International conference on tools and algorithms for the construction and analysis of systems (TACAS)*, in Lecture notes in computer science, vol. 12652. Springer, 2021, pp. 22–38. doi: 10.1007/978-3-030-72013-1_2.

[13] K. Bhargavan *et al.*, "Formal Verification of Smart Contracts: Short Paper," in *Proceedings of the 2016 ACM workshop on programming languages and analysis for security (PLAS)*, ACM, 2016, pp. 91–96. doi: 10.1145/2993600.2993611.

[14] I. Grishchenko, M. Maffei, and C. Schneidewind, "A Semantic Framework for the Security Analysis of Ethereum Smart Contracts," in *Principles of security and trust (POST)*, in Lecture notes in computer science, vol. 10804. Springer, 2018, pp. 243–269. doi: 10.1007/978-3-319-89722-6_9.

[15] Y. Hirai, "Defining the Ethereum Virtual Machine for Interactive Theorem Provers," in *Financial cryptography and data security (FC)*, in Lecture notes in computer science, vol. 10323. Springer, 2017, pp. 520–535. doi: 10.1007/978-3-319-70278-0_33.

[16] L. Cohen, J. Eremondi, M. Sagiv, and J. R. Wilcox, "Certified Reasoning about Contract Updates," in *Proceedings of the 2017 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications (OOPSLA)*, ACM, 2017, pp. 1–28. doi: 10.1145/3133884.

[17] A. Mok, T. König, N. Kher, H.-B. Sim, and K. R. M. Leino, "Verus: A language and methodology for formal verification," in *Formal Methods in Computer-Aided Design (FMCAD)*, IEEE, 2022, pp. 252–263. doi: 10.1109/FMCAD55376.2022.9924300.

[18] P. Jovanovic, "Foundations and Practice of Formal Verification of Smart Contracts," *arXiv preprint*, 2021, Available: https://arxiv.org/abs/2102.04323

[19] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, 2nd ed. Springer, 2016.

[20] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley Professional, 1997.

[21] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (CCS)*, ACM, 2016, pp. 254–269. doi: 10.1145/2976749.2978309.

[22] R. Gu *et al.*, "CertiKOS: A Formally Verified OS Kernel," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, USENIX Association, 2016, pp. 635–651.

[23] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[24] B. Alpern and F. B. Schneider, "Defining Liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985, doi: 10.1016/0020-0190(85)90056-0.