OPTIMIZED DECENTRALIZED REWARD DISTRIBUTION(1)

Chun-Hu Cui
fleetpro@gmail.com

He-Song Cui
jsreputation@gmail.com

ABSTRACT

In DeFi (Decentralized Finance) applications, and in dApps (Decentralized Application) generally, it is common to periodically pay interest to users as an incentive, or periodically collect a penalty from them as a deterrent. If we view the penalty as a negative reward, both the interest and penalty problems come down to the problem of distributing rewards. Reward distribution is quite accomplishable in financial management where general computers are used, but on a blockchain, where computational resources are inherently expensive and the amount of computation per transaction is absolutely limited with a predefined, uniform quota, not only do the system administrators have to pay heavy gas fees if they handle rewards of many users one by one, but the transaction may also be terminated on the way. The computational quota makes it impossible to guarantee processing an unknown number of users.

We propose novel algorithms that solve Simple Interest, Simple Burn, Compound Interest, and Compound Burn tasks, which are typical components of DeFi applications. If we put numerical errors aside, these algorithms realize accurate distribution of rewards to an unknown number of users with no approximation, while adhering to the computational quota per transaction. For those who might already be using similar algorithms, we prove the algorithms rigorously so that they can be transparently presented to users. We also introduce reusable concepts and notations in decentralized reasoning, and demonstrate how they can be efficiently used. We demonstrate, through simulated tests spanning over 128 simulated years, that the numerical errors do not grow to a dangerous level.

***K*eywords** DeFi · dApp · staking reward · reward distribution · pendency tracker · activity tracker

1 Introduction

Saving computational resources is a general demand in all computing applications, but it has become a vital need in blockchains. Blockchains impose a unilateral, uniform, and unconditional quota on the amount of computation that can be used for a transaction. Administrators as well as users must adhere to the quota while also paying a significant amount of fees in proportion to the amount of computation consumed. We cannot trade off the integrity, consistency, and transparency requirements of applications to work around the quota problem, because the fact that blockchain is selected as the platform in the first place means we cannot compromise on the requirements.

The problem of overcoming the high computational cost and the computational quota per transaction arises especially when processing arrays of unknown length. Suppose, for example, there is a requirement to pay each user 0.01 % interest every day based on the amount they keep staked. On a general computer platform, it is possible and common to enumerate all users, calculate their interest amount, and send the amount to their accounts. However, the computational quota per transaction on blockchains make it a necessity to find fundamentally different solutions. An immediate naive solution is to restrict the number of users, which is unacceptable.

The next idea is to process a certain number of users in one transaction and repeat a similar transaction several times until covering all users. The flaw is that the repetition has to be initiated and controlled from the off-chain part by system administrators, or, equivalently, by their administration automation tools. This will prevent the achievement of the integrity, consistency, and transparency goal that called for choosing blockchain as the platform in the first place. If administrators and off-chain administration tools were such reliable, they wouldn't have chosen the expensive blockchain as the platform in the first place.

Some algorithms implement reward distribution to an unknown number of users without enumerating users. The earliest one is the prize distribution algorithm adopted in the MasterChef smart contract of the PancakeSwap DeFi application. Instead of calculating each and every users prize share each time a new total prize is collectively available to users, they maintain the accumulated prize dividend index for each unit of total share (so, of each users' share) each time before the total share (so a user's share) changes its value. The prize per share is used by all other users to calculate their pending prize. This allows administrators to pretend to have processed users prize without actually having processed them all. This *virtual* processing method has been a model for many applications. We also gain inspiration from this method.

We formulate our methodology in Section 2. We first identify reward distribution tasks that we handle in this paper, (see Section 2.1), in terms of firstly how much rewards should theoretically be available for the dApp to distribute to users, and secondly whether the rewards are added to the same account as their principal account or to a separate asset account. Simple Interest, Simple Burn, Compound Interest, and Compound Burn are the types of reward distribution tasks that we identify and aim to find algorithms for in this paper. See Table 1 for the task types and their algorithm names.

We then introduce the Consistency Criteria for a reward distribution algorithm, as well as Relative Errors as a derived concept. See Section 2.3 for the Consistency Criteria and Relative Errors. Relative Errors are then used to assess the accuracy and numerical errors of our algorithms.

We finally discuss several imaginary types of reward distribution tasks and identify the position of our aimed reward distribution tasks among those concepts. See Section 2.2 for more.

We present and prove our algorithms in Section 3. We depict them in UML State Machine diagrams for quick reference and comprehension. We find two alternative algorithms for each aimed reward distribution task: pendency tracker and activity tracker. See Table 1 for the classification of algorithms.

Task type	Algorithms that do not handle errors
Simple Interest	Simple Interest Pendency tracker Simple Interest Activity tracker
Simple Burn	Simple Burn Pendency tracker Simple Burn Activity tracker
Compound Interest	Compound Interest Pendency tracker Compound Interest Activity tracker
Compound Burn	Compound Burn Pendency tracker Compound Burn Activity tracker

Table 1. Classification of algorithms

The term $pendency$ and $activity$ are two alternative variables that represent the state of a reward distribution algorithm. They both allow tracking an aspect of reward distribution, but it turns out that they each have unique pros and cons. The activity tracker algorithms are proved symbolically by mathematical induction. We demonstrate that appropriate concepts and notations can help reasoning of decentralized processes.

Section 3.9 discusses numerical error sources of the algorithms. There are two types of numerical error sources theoretically. Errors coming from integer expression of the exponentiation of fractional numbers are exponentiation errors, while errors coming from integer expression of the division of integers are division errors. Both types of errors are small enough to ignore, if we don't have extensively many transactions in our applications, although we propose an idea of how to mitigate division errors.

Section 4 discusses our simulation tests and their results on our four reward distribution task types. Despite that the Relative Errors in some tasks increase linearly, or even exponentially, over block numbers, they are less than 10^{-11} in compound tasks and less than 10^{-22} in simple tasks in our particular simulated tests, which span 128 simulated years and assume quite steep interest/penalty rates and modest frequency of transactions.

2 Methodology

2.1 Tasks

Throughout this paper, we assume decentralized reward distribution applications where users' digital assets are stored on a *smart contract*, and all changes to the assets are only made through and by access functions defined by the smart contract. Without losing generality, we assume *solidity* language as our smart contract programming language. *Reward distribution* is defined, for the purpose of this paper, as *making the assets that are theoretically available to users physically available to them by crediting the assets to or debiting the assets from their destination accounts,* or by getting ready to fulfill any compatible asset transfer requests immediately.

We identify types of reward distribution tasks below, in terms of both the *amount* and textitdestination of rewards. We only handle numerical amount of assets and don't care of the nature and usage of the assets, which is up to particular applications. We note it is customary to burn up collected penalties and to call the penalty a burn in Decentralized Applications. We follow the custom and may call the penalty simply a burn. We also note that interest is a positive reward while burn is a negative reward.

For short notations, U denotes the set of all possible users, throughout this paper.

- Simple Interest

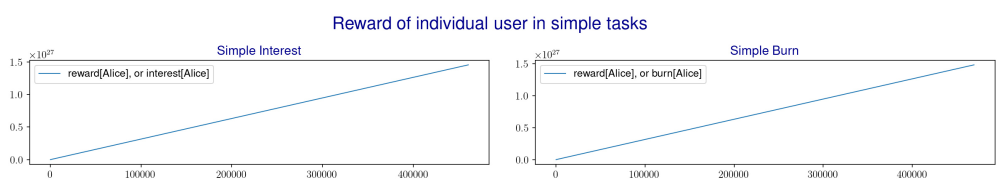

Figure 1. In a simple task, $reward(user)$ for any user $user$, as well as $\sum_{u \in U} reward(u)$, grows linearly over time, because simple tasks only collect rewards that grow linearly over time and the collected rewards have their own account different from the principal account.

Simple Interest is a type of reward distribution task where the amount and destination of rewards are defined by the following programming pseudocode:

$$reward[user] = reward[user] + principals[user] * rate * blocks/cycle \tag{1}$$

where $user$ is the user to whom the rewards are distributed, $reward[user]$ is the reward destination account that stores $user$'s rewards, $principals[user]$ is $user$'s amount of principal, $blocks$ is the number of blockchain blocks that elapses, $cycle$ is a certain positive integer, and $rate$ is the interest rate formulated: "the interest as much as $rate$ portion of $principal$ should be paid to the user every $cycle$ blocks that elapses." The meaning and notation of variables are the same throughout this paper, except that $rate$ refers to the penalty rate in burn tasks and is formulated: "the penalty as much as $rate$ portion of $principal$ should be collected from the user every $cycle$ blocks that elapses."

In this task, the user earns interest proportional to the interest rate and time that elapses on their principal. The interest is accumulated to its own account separate from that of the principal, and is *not* credited to the principal account. The rewards can even be a different type of asset from the principal. See Figure 1 for more. We choose block number, rather than block timestamp, as the measure of time for security reasons. The interest being simply additively accumulated in an account does not define how the interest is processed or used in particular applications.

- Simple Burn

Simple Burn is a type of reward distribution task where the amount and destination of rewards are defined by the following programming pseudocode:

$$reward[user] = reward[user] + principals[user] * rate * blocks/cycle \tag{2}$$

In other words, the user is charged with a penalty proportional to the penalty rate and time that elapses on their principal. The penalty has its own destination account different from that of principal, and is *not* debited from the principal amount. The penalty can even be a different type of asset from the principal. See Figure 1 for more. The penalties, which are negative rewards, are simply positively accumulated in the reward account, because our algorithms do not extend to taking care of how the penalty is exercised or materialized in particular applications.

- Compound Interest

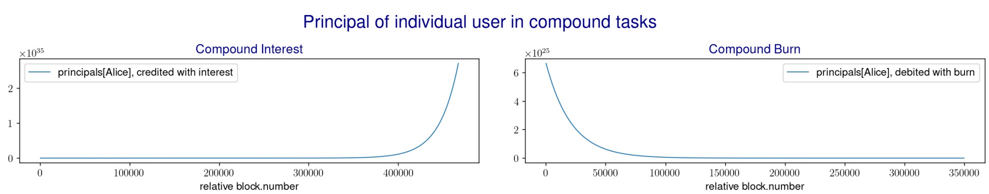

Figure 2. In a compound task, $balance(user)$ for any user $user$, which is $user$'s $principal$ plus/minus $user$'s interest/burn, as well as $\sum_{u \in U} balance(u)$, grows/shrinks exponentially from its initial value, because compound tasks only collect interest/burn that is exponential over time and the collected interest/burn is credited to/debited from the principal account.

Compound Interest is a type of reward distribution task where the amount and destination of rewards are defined by the following programming pseudocode:

$$principals[user] = principals[user] + principals[user] * ((1 + rate)^{blocks/cycle} - 1) \tag{3}$$

In other words, interest is created by the amount $principal[user]$ and credited to the $principal[user]$ account. Rigorously, the user continuously earns time-linear interest on their principal amount while the earned interest is continuously credited to the principal amount. The continuity is implemented by exponentiation. The interest and principal not only share the same asset type with each other but also share the same account. See Figure 2 for more.

- Compound Burn

Compound Burn is a type of reward distribution task where the amount and destination of rewards are defined by the following programming pseudocode:

$$principals[user] = principals[user] - principals[user] * (1 - (1 - rate)^{blocks/cycle}) \tag{4}$$

In other words, the burn (penalty) is created by the amount $principal[user]$ and debited from the $principal[user]$ account. Rigorously, the user continuously pays a time-linear burn on their principal while the paid burn is continuously debited from the principal amount. The continuity is implemented by exponentiation. The penalty and principal not only share the same asset type with each other but also share the same account. See Figure 2 for more.

- Simple Shared Prize
  Simple Shared Prize is a type of reward distribution task where the amount and destination of rewards are defined by the following programming pseudocode:

$$reward[user] = reward[user] + principals[user]/totalPrincipal * alpha * blocks/cycle, \tag{5}$$

  where $alpha$ is a constant that determines the total prize created for all users collectively over time. The user earns their share of the total prize $alpha * blocks/cycle$. The earned rewards have a separate account from that of the principal.

Simple tasks are simple because the rewards are destined to a separate account and they may even be of different asset type, whereas compound tasks are compound because the rewards are destined to the principal account, either by adding to or subtracting from the existing principal.

We propose algorithms that solve the former *four* tasks, under the conditions that the number of users is unknown and there is a computational quota. The 5th task is used as a reference to understand the former four tasks.

2.2 Position

We discuss the position of our aimed task types identified in Section 2.1 on the map of possible reward distribution policies.
We can identify several imaginable types of reward distribution task types that can be thought of but unreasonable practically, in order to clarify the position of our aimed types of reward distribution tasks, as follows:

- Simple Interest Exponential
  Simple Interest Exponential is an *unreasonable* type of reward distribution task where the amount and destination of rewards are defined by the following programming pseudocode:

$$reward[user] = reward[user] + principals[user] * ((1 + rate)^{blocks/cycle} - 1) \tag{6}$$

  The more frequently rewards are collected according to this formula, the less total reward the user will earn. Users will not move, unless. If this formula is intentionally used to discourage them from collecting their rewards frequently, the algorithm is likely found by tweaking our algorithms.

- Simple Burn Exponential
  Simple Burn Exponential is an *unreasonable* type of reward distribution task where the amount and destination of rewards are defined by the following programming pseudocode:

$$penalty[user] = penalty[user] + principals[user] * ((1 - rate)^{blocks/cycle} - 1) \tag{7}$$

  The more frequently penalty is paid according to this formula, the less total penalty the user will pay. Users will not rest, unless. If this formula is intentionally used to encourage them to pay penalty frequently, the algorithm is likely found by tweaking our algorithms.

- Compound Interest Linear
  Compound Interest Linear is an *unreasonable* type of reward distribution task where the amount and destination of rewards are defined by the following programming pseudocode:

$$principals[user] = principals[user] + principals[user] * (alpha * rate * blocks/cycle) \quad (8)$$

  The more frequently the reward is collected according to this formula, the more total reward the user will earn. Users will not rest, unless. The algorithm, nonetheless, is likely found by tweaking our algorithms.

- Compound Burn Linear
  Compound Burn Linear is an *unreasonable* type of reward distribution task where the amount and destination of rewards are defined by the following programming pseudocode:

$$principals[user] = principals[user] - principals[user] * (alpha * rate * blocks/cycle) \quad (9)$$

  The more frequently the penalty is paid according to this formula, the less total penalty the user will pay. Users will not rest, unless. The algorithm is likely found by tweaking our algorithms.

We exclude Simple Shared Prize type of tasks from our goal, because

- The algorithm created by the Pancakeswap dApp can correctly answer $pending(user)$ for Simple Shared Prize tasks. (See Equation 5 for the formula of Simple Shared Prize.)
- We can extend with ease the PancakeSwap algorithm to correctly answer $balance(user)$, $totalPending()$, and $totalBalance()$ for any Simple Shared Prize tasks, by using a similar logic as used for other task types in this paper. See Section 2.3 for $balance(user)$, $totalPending()$, and $totalBalance()$.

Our *goal* is, therefore, to find algorithms that distribute rewards that are generated continuously over time to users for Simple Interest, Simple Burn, Compound Interest, and Compound Burn tasks, for an unknown number of users and adhering to the computational quota.

To the best of our knowledge, virtual distribution was invented by the PancakeSwap DeFi application. Their virtual distribution method is often used as a model to work around the computational quota in subsequent Decentralized Applications. We confirm, however, that our goal can *not* be accomplished by tweaking the PancakeSwaps algorithm. In their algorithm, a user's rewards are generated by the *relative* amount of the user's principal, while on our tasks, a user's rewards are generated by the *absolute* amount of the user's principal. See Section 2.1 for their respective reward formulas. There have been frequent attempts to solve the Compound Interest and Compound Burn tasks, raising several concepts, for example, around Compound Interest: periodic compound, manual compound, continuous compound, and automatic compound. We should make it clear *which* of the concepts relate *how* to our algorithms. We discuss these concepts one by one, although they are *not* exclusive of each other.

To avoid confusion, compounding itself refers to adding interest to the principal that created the interest or subtracting a burn (penalty) from the principal that caused the burn. The amount of interest to be compounded can be either $principal * ((1 + rate)^{period} - 1)$ or $principal * rate * period$, which we call the time-exponential interest and the time-linear interest, respectively. As with burn tasks, they are $principal * ((1 - (1 - rate)^{period}))$ and $principal * rate * period$, and called time-exponential burn and time-linear burn, respectively.

- Periodic Compound
  This task literally compounds periodically, either regularly or irregularly. The compounding must be a part of a blockchain transaction and the transaction must be invoked either by administrators or users. Periodic compounding of time-linear interest by users' transactions may cause a meaningless competition or bank-run between users, because the more frequently they compound, the more interest they earn. *Periodic compounding by users is reasonable for time-exponential interest*, as frequency has no effect on compounding

time-exponential interest. Periodic compounding of time-linear interest by administrators' transactions, on the other hand, might hurt users if administrators or administration automation tools fail to call compounding in time, restricting the growth of users' interest. Periodic compounding of time-exponential interest by administrators' transactions might cause users to await, if they fail, for the next round of compounding to be unleashed. Administrators don't need to *unreasonably* take compounding over while users can *reasonably* take responsibility for that.

- Manual Compound

  If manual compounding means compounding with direct personal involvement of people rather than by off-chain automation tools, then we need to note that people are the most unreliable component in a Decentralized Application, unless the people are users compounding for themselves. Users compounding for themselves means users in need of their compounding call compounding at their discretion. *This will allow them to be responsible for their rewards*. If manual compounding means compounding by administrators' transactions, rather than by users', and if administrators fail, then compounding may stop while users are still using the system.

- Automatic Compound

  If automatic compounding means compounding with no direct personal involvement of people but with their automation tools, off-chain tools are second most unreliable component for a Decentralized Applications, unless the tools are operated by users for themselves and at their discretion. If automatic compounding means compounding with no direct involvement of administrators, then it is with the involvement of users and by users' transactions. *Compounding will not stop as long as users in need of compounding keep using the system*, if compounding is left to users.

- Continuous Compound

  We cannot compound continuously, as nobody wants to invoke compounding transactions every block. Continuous compounding can be viewed as periodic compounding of time-linear interest with an infinitely small compounding period. This does not necessarily mean calling compounding transactions in every block, which too is not enough to be continuous. *Continuous compounding can be implemented by, periodically or intermittently, regularly or irregularly, compounding time-exponential interest.* As mentioned above, frequency has no effect on compounding time-exponential interest. *Continuous compounding, or compounding time-exponential interest, is only reasonable if invoked by users for themselves.*

We observe above that the desirable compounding should be by users' transactions compounding time-exponential interest for users themselves at their discretion, whether it be manual or automatic. (Manual and automatic are not defined clearly.) A Decentralized Application, after all, should be in operation only while there are users using it, not while there are administrators. Users will, immediately or eventually, have to pay more for more gas, but gas fees cannot warrant compounding called by administrators.

We follow this observation and *choose, regular or irregular, compounding of time-exponential interest carried out by users' transactions for users themselves, whether it be manual or automatic.*

Carrying out compounding by users' transactions leads to getting compounding actions, which are part of our algorithms, *parasitic on users' transactions*. Technically, this is implemented by our algorithms hooking user transactions, working inside the transactions' context, and collecting pending rewards before the transactors perform their intended actions, although true dynamic hooking is not possible on a blockchain and in Solidity language. *Coincidentally, normal users' transactions work better if pending rewards have been collected before performing their intended actions*. For example, when a user transfers a portion of their net principal to someone else, the user wants to collect pending interest into the principal account before transferring. Generally, compounding actions should get parasitic on *every* principal-changing transaction and precede it. Any user transactions that require compounding to precede themselves may want compounding actions to get *parasitic* on themselves. The transactions include: mint, burn, transfer, stake, un-stake, harvest, etc. See List 2.2 for how to implement compounding actions parasitic on users' transactions.

```
function mint(user, amount) {
  changePrincipal(user, amount); # Collect user's pending rewards, and credit user's principal
}
function transfer(sender, recipient, amount) {
  changePrincipal(sender, -amount); # Collect sender's pending rewards, and debit their principal
  changePrincipal(recipient, amount); # Collect recipient's pending rewards, and credit their principal
}
```

Listing 1. Example of $changePrincipal(user, amount)$ function parasitic on transactions. The $changePrincipal(user)$ function in our algorithms acts as the compounding action that gets parasitic on users' transactions and precedes the transactions' intended actions. This function collects all pending rewards of the user and adds it to their destination account. *Furthermore*, the function takes over principal-chaining actions from the transactions, as compounding actions and principal-changing actions have high cohesion and should be in the same module, from the software engineering point of view.

One of the *thumb rules* that we learn from the previous and this section is that reasonable simple tasks handle linear rewards whereas reasonable compound tasks handle exponential rewards.

2.3 Consistency criteria

We clarify the following terms:

- If *a distribution is made actually*, all users' entitled rewards are moved to their respective destination accounts.
- If *a distribution is made actually and immediately*, the distribution is made actually, as soon as users are entitled to some rewards.

Algorithms that distribute rewards to an unknown number of users should act *as if all distributions were made actually and immediately*, while actual distributions are deferred until suitable moments of time, because immediate actual distribution to all users is not guaranteed to succeed due to the computational quota and possibly excessively large number of users. Acting this way is called *making virtual reward distribution*.

For a query into a reward distribution process, we clarify the term *return value* and *true value*:

- return value, for a query, is the value returned by an algorithm that is running in a particular program, in response to the query.
- true value, for a query, is the value that exists for the query purely by accounting principals and independently of algorithms and programs.

A reward distribution algorithm that is running in a particular program is said to be *consistent* at a moment if and only if the ***Consistency Criteria***, defined below, are satisfied at the moment:

- A query $pending(user)$'s return value $pending(user)$ equals its *true value* $\{pending(user)\}$ for any user $user$, which is the current amount of reward that $user$ is entitled to but is not yet actually distributed.
- A query $balance(user)$'s return value $balance(user)$ equals its *true value* $\{balance(user)\}$ for any user $user$, which is the balance of $user$'s reward destination account plus/minus $pending(user)$. (Minus is for burn tasks.) Equivalently, it answers to "what would the balance of $user$'s reward account be if all distributions were made actually and immediately." We note that in compound tasks the reward account is the same as the principal account, unlike in simple tasks. See Section 2.1 for more about task types.
- A query $totalPending()$'s return value $totalPending()$ equals its *true value* $\{totalPending()\}$, which is $\sum_{u \in U}\{pending(user)\}$. This query has its own significance, as it may be impossible to sum up across all users. Algorithms find this value indirectly.

- A query $totalBalance()$'s return value $totalBalance()$ equals its *true value*, $\{totalBalance()\}$, which is $\sum_{u \in U}\{balance(user)\}$. This query has also its own significance, as it may be impossible to sum up across all users. Algorithms find this value indirectly.
- The algorithm allows consistent transfers, meaning that the algorithm allows all transfers of any amount of asset from the reward destination account of any user $user$ if the amount is equal to or less than $balance(user)$, unless the application prohibits the transfers.

The Consistency Criteria can be simplified as:

- $pending(user) = \{pending(user)\}$ for any user $user$
- $balance(user) = \{balance(user)\}$ for any user $user$
- $totalPending() = \{totalPending()\}$
- $totalBalance() = \{totalBalance()\}$
- the algorithm allows consistent transfers

Algorithms may not be consistent, because they may

- have errors in their logic, unlike our algorithms
- have an approximation in their logic, unlike our algorithms
- suffer numerical errors of computer operation, *like* our algorithms

We prove that our algorithms have no errors in their logic by rigorously verifying that *our algorithms are consistent* at any moment, in the next section.

As for numerical errors, we discuss the errors theoretically and in simulated tests. We introduce the following *Consistency Errors* as performance measures of our algorithms running in a particular program:

$$TrueTotal = \{totalBalance()\} = \sum_{u \in U}\{balance(user)\} \tag{10}$$

$$Absolute\ Error\ A = |totalBalance() - TrueTotal| \tag{11}$$

$$Absolute\ Error\ B = |\sum_{u \in U} balance(u) - TrueTotal| \tag{12}$$

$$Relative\ Error\ A = |totalBalance() - TrueTotal|/TrueTotal \tag{13}$$

$$Relative\ Error\ B = |\sum_{u \in U} balance(u) - TrueTotal|/TrueTotal \tag{14}$$

Consistency Errors only takes care of $balance(user)$, and not of $pending(user)$, because $balance(user)$ is not independent of $pending(user)$ but is an accumulation of $pending(user)$. Absolute Error and Relative Error may be called Absolute Consistency Error and Relative Consistency Error, respectively.

3 Algorithms

In this section, the eight algorithms for our four task types are proved to be consistent at any moment, under the assumption that there are no computer numerical errors, as is not the case. See Table 1 for a classification of the eight algorithms. Numerical errors are handled in the last subsection 3.9, where random algorithms that mitigate some of numerical errors are proposed and discussed.

The algorithms presented below are in the form of UML State Machine diagram, without losing rigorosity. We choose verbal proof for pendency tracker algorithms, and choose symbolic proof for activity tracker algorithms, for comparison.

3.1 Simple Interest pendency tracker algorithm

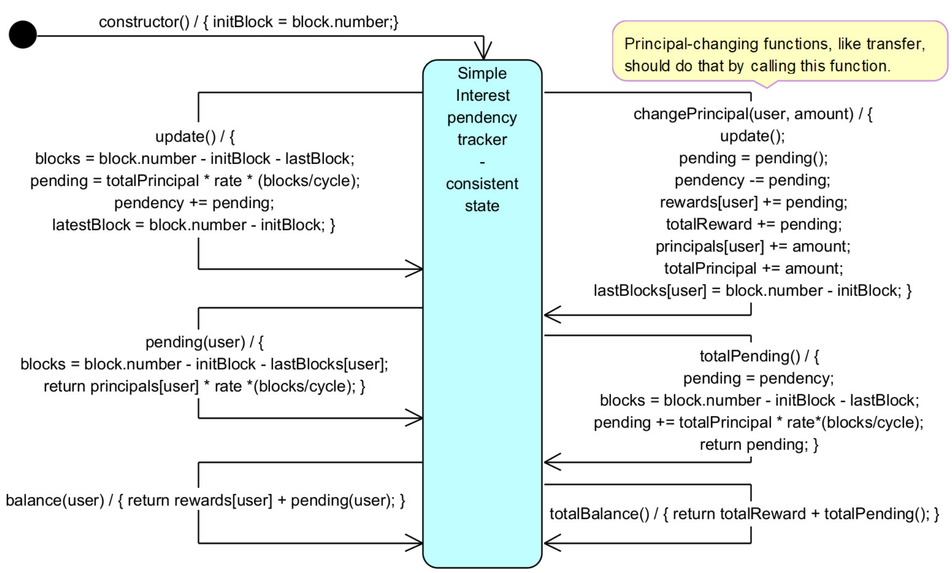

Figure 3. The UML State Machine of Simple Interest pendency tracker algorithm. The four functions corresponding to queries in the Consistency Criteria, as well as the $changePrincipal(user, amount)$ function, are represented as events of the state machine. When invoked, these events are supposed to let the state machine transition from the *consistent state* back to the same *consistent state*.

See Equation 1 for the formula of Simple Interest tasks. See Figure 3 for the state machine of the algorithm.

The algorithm can be proved as follows:

The $changePrincipal(user, amount)$ function collects and compounds all pending interest of a given user $user$, whenever *before* changing the variable $principals[user]$, because the interest formula Equation 1 is a function of a *constant* $principals[user]$ and the elapsed time period over which the $principals[user]$ remained that constant. After finishing the $changePrincipal(user, amount)$ function, the $user$'s pending interest becomes zero. This justifies the logic of the $pending(user)$ function, which simply returns the rewards created after the latest call on the $changePrincipal(user, amount)$ function.

As for the $totalPending()$ function, the variable $pendency$ is tracked by the $update()$ and $changePrincipal(user, amount)$ functions for the user $user$ who is currently calling a principal-changing transaction, which, in turn, calls the current instance of $changePrincipal(user, amount)$ function. The variable $pendency$ is added, in the $update()$ function, with the *total interest* newly created by the existing total principal $totalPrincipal$ during the period over which the total Principal was kept to the current constant, whenever before the total principal is changed, so, whenever before a user's principal is changed. That newly created total interest should represents *all* users' newly created interest for the same period. The variable $pendency$ is then subtracted with *the* user's pending interest and the user's reward account is credited with that pending interest, effectively distributing the user's pending interest to the user, via $pendency$. Therefore, the variable $pendency$ indicates the total pending interest, as of the latest $changePrincipal(user, amount)$ call, that is not yet actually distributed to individual users other than *the* very user. When asking the $totalPending()$ query, the additional interest created after the latest $changePrincipal(user, amount)$ call is returned together with the $pendency$.

The $balance(user)$ function is straightforward. The balance of a user is the sum of their rewards, which is collected and accumulated interest of the user, and their pending interest. The $totalBalance()$ is similar.

This algorithm is called a pendency tracker, because the $totalPending()$ function is calculated by using a representation of pending amount, $pendency$.

3.2 Simple Burn pendency tracker algorithm

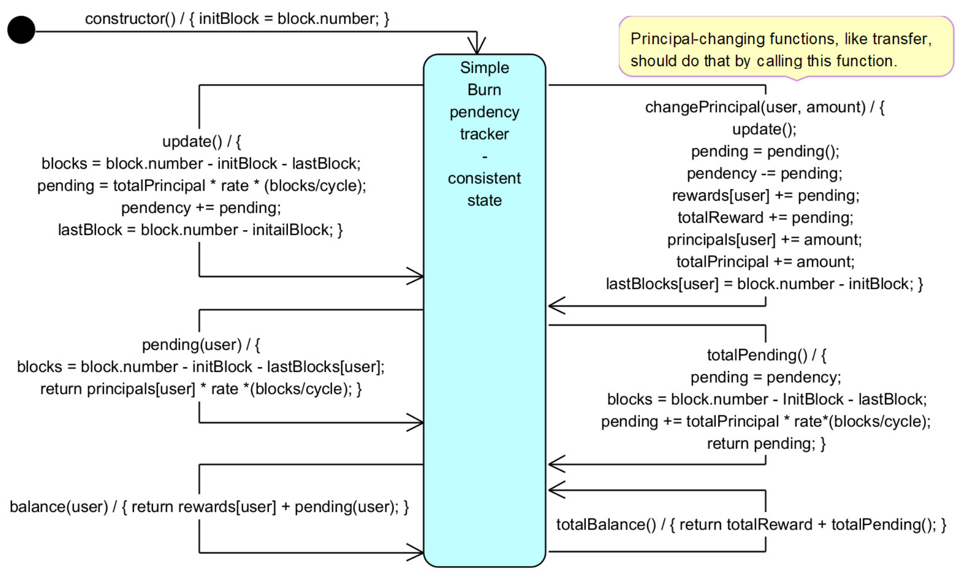

Figure 4. The UML State Machine of Simple Burn pendency tracker algorithm.

See Equation 2 for the formula of Simple Burn tasks. See Figure 4 for the state machine of the algorithm.
We note in the functions $balance(user)$ and $totalBalance()$, the pending penalty, which is the penalty theoretically charged but not yet actually distributed, is added to, and not subtracted from, the user's reward. This is because these algorithms solve only quantitative relationships and do not relate to how the assets are materialized, simply accumulating charged penalties into the negative reward account.

This algorithm can be proved similarly as in Simple Interest Pendency tracker.

3.3 Compound Interest pendency tracker algorithm

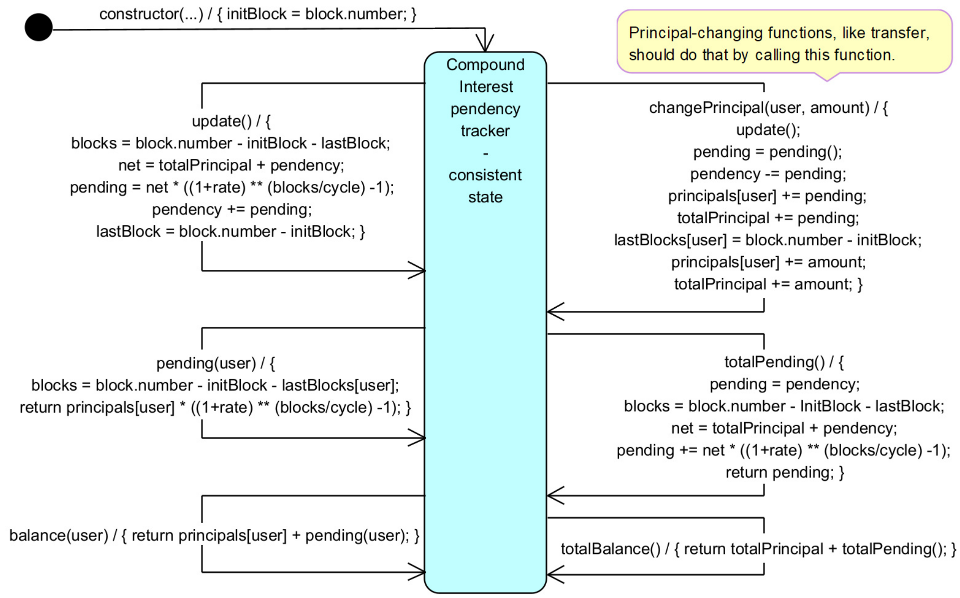

Figure 5. The UML State Machine of Compound Interest pendency tracker algorithm.

See Equation 3 for the formula of Compound Interest tasks. See Figure 5 for the state machine of the algorithm.

3.4 Compound Burn pendency tracker algorithm

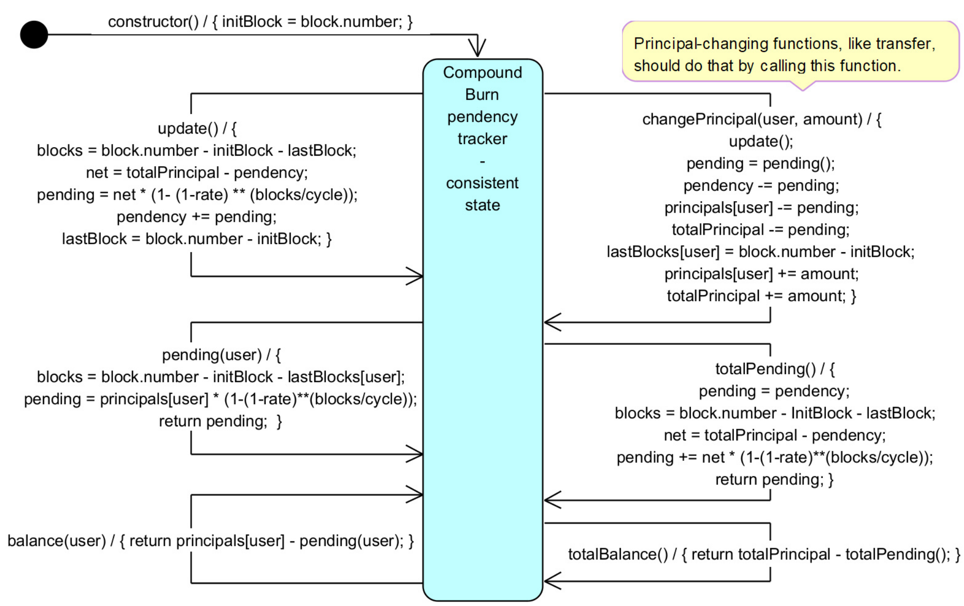

Figure 6. The UML State Machine of Compound Burn pendency tracker algorithm.

See Equation 4 for the formula of Compound Burn tasks. See Figure 6 for the state machine of the Compound Burn pendency tracker algorithm.

This algorithm can be proved Similarly as in Simple Interest Pendency tracker

3.5 Simple Interest activity tracker algorithm

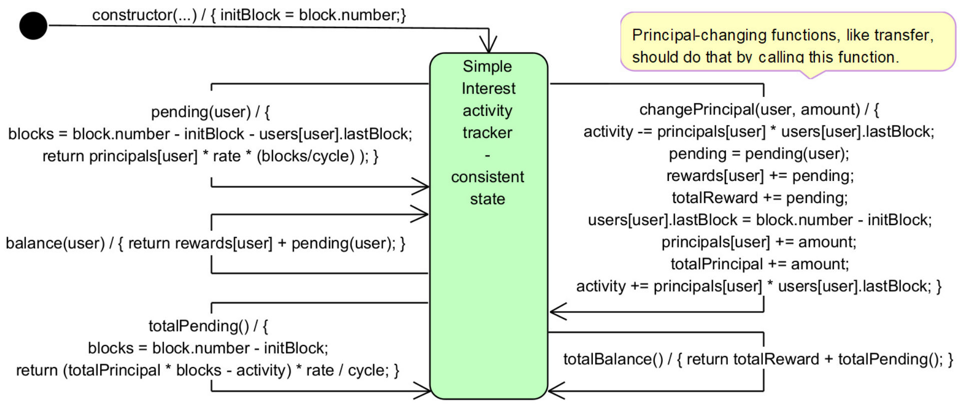

Figure 7. The UML State Machine of Simple Interest activity tracker algorithm.

See Equation 1 for the formula of Simple Interest tasks. See Figure 7 for the state machine of the Simple Interest *activity* tracker algorithm.

Unlike *pendency* tracker algorithms presented so far that track the amount of rewards that are not yet distributed, *activity* tracker algorithms track a representation of users' activity in terms of how long time users keep how much principal. Symmetric, they should be equivalent, but it is observed that the two performs not the same and not necessarily one is better than the other in all aspects. Pendency tracker and activity tracker algorithms are alternatives to each other and we can choose between them in practice according to specific task requirements. See Section 4 for more.

We introduce the following definitions and notations, which might be used in symbolic reasoning of general blockchain techniques:

- **History** is a set of identified events in a particular decentralized application program.
  - The identified events primarily may include all transactions in the program.
  - If the task has a transaction in a block, then the block can also be included in the identified events.
  - The initial block $initBlock$, where the decentralized application program is deployed on the blockchain, is included, in particular.
  - We assume there is an event $init$ where all variables are initialized from no value to zero at the beginning of $initBlock$.
  - If we need to identify the evaluation of some successive individual programming statements in a particular transaction, then those statements are also an event in the event history.
  - Different programs may have different history for the same application class, as each program relates to events of its own interest.
- **Left moment of event e**, denoted by $e-$, is the very start of the event e and has no duration. **Right moment of event e**, $e+$, is the very end of event e. For example, if e is a block, then $e-$ is the start of the block; if e is a transaction, then $e+$ is the end of the transaction; etc. We assume $init-$ is equal to $initBlock-$, in particular.
- **Moments of history** H, $M(H)$, means $\cup_{e \in H}\{e-, e+\}$
- **Ordered moments of history** H, $OM(H)$, is a sequence that consists of elements of the moments of history H and that is arranged in the order of taking place. The ordered moments of history exist uniquely for a given history, and is a finite-length sequence or has the same structure as natural numbers.

- **Block of moment m**, $B(m)$ for a moment m, is the block or block number where the moment takes place.
- **Quantity Q as of moment m**, $Q^{[m]}$ for a moment m, is the quantity of property Q that exists at the moment m.
- **All users**, U, is the set of all possible account addresses.

We also have the following definitions and notations specifically for this paper:

- **Last block of user u at moment m**, $u.lastBlock^{[m]}$, is the block or block number where the user u's principal changed latest before the given moment m.
- **Principal of user u**, $principals[u]$, is the amount of the principal of user u.
- **Interest/burn rate**, $rate$, is an interest rate or burn rate in the reward distribution application.
- **Virtual rewarding period**, $cycle$, is a certain positive integer such that the interest/burn rate is described as "an interest/burn as much as $rate$ portion of the principal is credited from/debited to its destination account every $cycle$ block(s) that elapses." This is called a virtual because we don't actually collect rewards every $cycle$ blocks.

The following abbreviations are used interchangeably for the remaining part of this paper:

- P for $principals$
- lB for $lastBlock$
- C for $cycle$
- R for $rate$

We introduce the following definitions:

- A reward distribution algorithm is said to be consistent for a principal-changing event e if and only if the algorithm is consistent for any moment m over the moment interval $[e+, en-]$ where en is the next coming Principal-changing event. (See Section 2.3 for Consistency Criteria.)
- A reward distribution algorithm is said to be consistent if and only if the algorithm is consistent for any principal-changing event e.

We prove that the Simple Interest activity tracker algorithm is consistent, by mathematical induction for principal-changing events, as follows:

- **The algorithm is consistent for the initial principal-changing event** $init$.

  We have to prove that the algorithm is consistent at any moment m over the moment interval $[init+, n-]$ where n is the next coming principal-chaining event. Firstly, all the four queries in the Consistency Criteria return zero; which is its true value, because the principals of all users remain zero over the interval $[init+, n-]$, as there were no principal-changing actions at all after all principals were initialized to zero by the event $init$.

  Secondly, the algorithm allows consistent transfers, because $balance(user)$ is a zero for any user $user$, and the algorithm can always transfer/debit a zero amount from the reward destination account of $user$.

- **If we assume that the algorithm is consistent for a principal-changing event e, then it is also consistent for the next coming principal-changing event** ne .

  See Section 2.3 for Consistency Criteria. Let $\dot{u}$ be the user whose principal is changed by the event ne, then this proposition is proved as follows.

☐ $pending(u)^{[(u.lastBlock^{[e+]})+]} = 0$ for any user u.

Because, $pending(u)^{[r]}$, where $r = (u.lastBlock^{[e+]})+$,

$= principals[u]^{[r]} * rate * (B(r) - u.lastBlock^{[r]})$, by the algorithm

$= P[u]^{[r]} * R * (B((u.lB^{[e+]})+) - u.lB^{[(u.lB^{[e+]})+]})$

$= P[u]^{[r]} * R * (u.lB^{[e+]} - u.lB^{[e+]})$

$= 0$

☐ $activity^{[e+]} = \sum_{u \in U} principals[u]^{[e+]} * u.lastBlock^{[e+]}$.
(See the algorithm state machine in Figure 7 for $activity$.)

Because, $totalPending^{[e+]}$

$= \sum_{u \in U} pending(u)^{[e+]}$, because the algorithm is consistent at $e+$,

$= \sum_{u \in U} \{pending(u)^{[(u.lB^{[e+]})+]} + P[u]^{[(u.lB^{[e+]})+]} * R * (B(e+) - u.lB^{[e+]})/C\}$

$= \sum_{u \in U} \{P[u]^{[e+]} * R * (B(e+) - u.lB^{[e+]})/C\}$,

because

- $pending(u)^{[(u.lB^{[e+]})+]} = 0$ for any user u;
- if $(B(e+) - u.lB^{[e+]}) = 0$ and, so, $P[u]^{[(u.lB^{[e+]})+]}$ does not yet exist at the moment $e+$, then we can replace the multiplier $P[u]^{[(u.lB^{[e+]})+]}$ with any value;
- if $(B(e+) - u.lB^{[e+]}) > 0$, then the user's principal didn't change since its latest change and

  $P[u]^{[(u.lB^{[e+]})+]} = P[u]^{[e+]}$.

$= (\sum_{u \in U} P[u]^{[e+]}) * B(e+) * R/C - (\sum_{u \in U} P[u]^{[e+]} * u.lB^{[e+]}) * R/C$

$= (totalPrincipal^{[e+]} * B(e+) - (\sum_{u \in U} P[u]^{[e+]} * u.lB^{[e+]})) * R/C$

On the other hand, the algorithm returns the following value to be $totalPending^{[e+]}$:

$(totalPrincipal^{[e+]} * B(e+) - activity^{[e+]}) * rate/cycle$.

Therefore, $activity^{[e+]} = \sum_{u \in U} P[u]^{[e+]} * u.lB^{[e+]}$.

☐ $activity^{[ne+]} = \sum_{u \in U} principals[u]^{[ne+]} * u.lastBlock^{[ne+]}$.

Because, $activity^{[ne+]}$

$= activity^{[ne-]} - principals[\dot{u}]^{[ne-]} * \dot{u}.lastBlock^{[ne-]} + principals[\dot{u}]^{[ne+]} * u.lastBlock^{[ne+]}$

$= activity^{[e+]} - P[\dot{u}]^{[e+]} * \dot{u}.lB^{[e+]} + P[\dot{u}]^{[ne+]} * u.lB^{[ne+]}$

$= \sum_{u \in U} P[u]^{[e+]} * u.lB^{[e+]} - P[\dot{u}]^{[e+]} * \dot{u}.lB^{[e+]} + P[\dot{u}]^{[ne+]} * u.lB^{[ne+]}$

$= \sum_{u \in U \setminus \{\dot{u}\}} P[u]^{[e+]} * u.lB^{[e+]} + P[\dot{u}]^{[ne+]} * u.lB^{[ne+]}$

$= \sum_{u \in U \setminus \{\dot{u}\}} P[u]^{[ne+]} * u.lB^{[ne+]} + P[\dot{u}]^{[ne+]} * u.lB^{[ne+]}$

$= \sum_{u \in U} P[u]^{[ne+]} * u.lB^{[ne+]}$

Below, we assume any moment r over the moment interval $[en+, o-]$ where o is the next coming principal-changing event after en, and prove that the algorithm is consistent at the moment r.

□ $pending(u)^{[r]}$ returns its true value for any user u.

Because, $pending(u)^{[r]}$

$= principals[u]^{[r]} * rate * (B(r) - u.lastBlock^{[r]})$

$= P[u]^{[e+]} * R * (B(r) - u.lB^{[r]})$, for $u \neq \dot{u}$

$= P[u]^{[e+]} * R * (B(e+) - u.lB^{[e+]} + B(r) - B(e+))$, for $u \neq \dot{u}$

$= pending(u)^{[e+]} + P[u]^{[e+]} * R * (B(r) - B(e+))$, for $u \neq \dot{u}$

where, from the assumption of induction, $pending(u)^{[e+]}$ returns its true value; and $P[u]^{[e+]} * R * (B(r_) - B(m+))$ is the true reward created after the moment $e+$. Therefore, $pending(u)^{[r]}$ returns its true value for $u \neq \dot{u}$.

For the user $\dot{u}$,

$principals[\dot{u}]^{[r]} * rate * (B(r) - u.lastBlock^{[r]})$

$= P[\dot{u}]^{[ne+]} * R(B(r) - B(ne+))$

$= pending(\dot{u})^{[e+]} - pending(\dot{u})^{[e+]} + P[\dot{u}]^{[ne+]} * R(B(r) - B(ne+))$

$= pending(\dot{u})^{[e+]} - pending(\dot{u})^{[ne-]} + P[\dot{u}]^{[ne+]} * R(B(r) - B(ne+))$

where $pending(\dot{u})^{[e+]}$ returns its true value, from the assumption of induction; $-pending(\dot{u})^{[ne-]}$ has occurred by the $changePrincipal(u, amount)$ function collecting the pending reward of the user $\dot{u}$ at the event ne; and $principals[\dot{u}]^{[ne+]} * R(B(r) - B(ne+))$ is the true reward created after the moment $ne+$. Therefore, we can say $pending(\dot{u})^{[r]}$ returns its true value.

□ $totalPending^{[r]}$ returns its true value.

Because, $totalPending^{[r]}$

$= (totalPrincipal^{[r]} * B(r) - activity^{[r]}) * rate/cycle$

$= totalPrincipal^{[ne+]} * B(r) * R/C - activity^{[ne+]} * R/C$

$= ((\sum_{u \in U} P[u]^{[ne+]} * B(r)) - (\sum_{u \in U} P[u]^{[ne+]} * u.lB^{[ne+]})) * R/C$

$= \sum_{u \in U} P[u]^{[u.lB^{[r]}+]} * (B(r) - u.lB^{[r]}) * R/C$

$= \sum_{u \in U} pending(u)[r],$

where $pending(u)[r]$ returns its true value for all users u. Therefore $totalPending^{[r]}$ also returns its true value.

- ☐ $balance(u)^{[r]}$ returns its true value for any user u.
  Because, according to the algorithm, $balance(u)^{[r]} = reward[u]^{[r]} + pending(u)^{[r]}$, where $reward[u]^{[r]}$ is its true value because it has been accumulated with historic true values $\{pending(u)\}$; and $pending(u)^{[r]}$ is also proved above to be its true value.
- ☐ $totalBalance^{[r]}$ returns its true value.
  Because, according to the algorithm, $totalBalance^{[r]} = totalReward^{[r]} + totalPending^{[r]}$, where $totalReward^{[r]}$ is its true value because it has been accumulated with historic true values $pending(user)$; and $totalPending^{[r]}$ is proved above to be its true value.
- ☐ The algorithm allows consistent transfers.
  Because the algorithm collects and adds $pending(user)$ to the user's actual balance so that the actual balance becomes the same amount as $balance(user)$ returns, before calling the requested transfer actions, which can now transfer/debit up to $balance(user)$ amount of asset from the actual balance.

Thus far, the Simple Interest pendency tracker algorithm is proved to be consistent, in the meaning defined in Section 3.5.

3.6 Simple Burn activity tracker algorithm

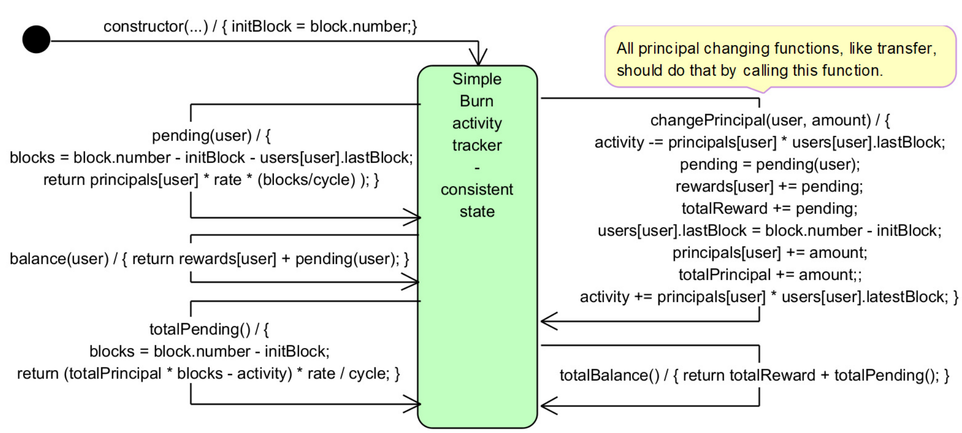

Figure 8. The UML State Machine of Simple Burn activity tracker algorithm.

See Equation 2 for the formula of Simple Burn tasks.

Figure 8 shows the state machine of the Simple Burn *activity* tracker algorithm.

This algorithm has symbolically the same state machine diagram and the same proof as the Simple Interest activity tracker algorithm.

3.7 Compound Interest activity tracker algorithm

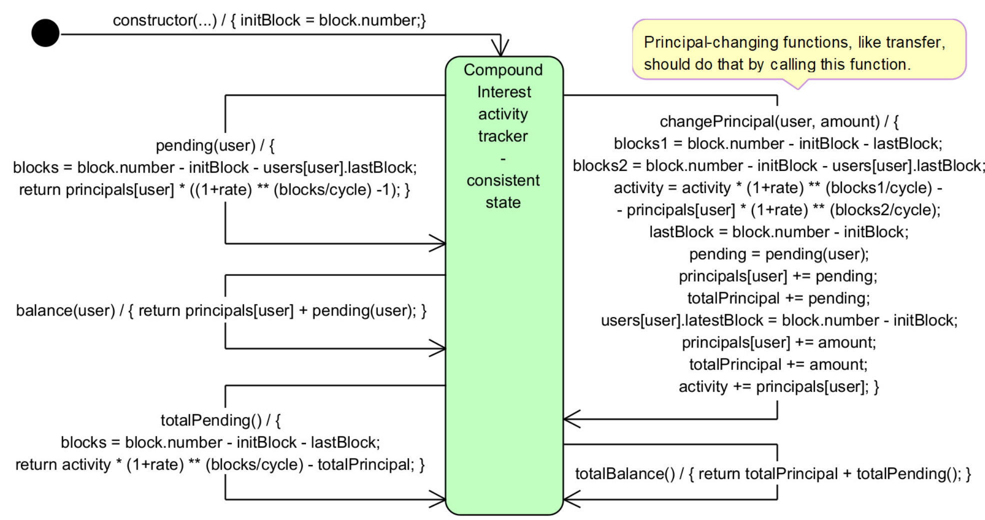

Figure 9. The UML State Machine of Compound Interest activity tracker algorithm.

See Equation 3 for the formula of Compound Interest tasks.

Figure 9 shows the state machine of the Compound Interest *activity* tracker algorithm.

With the same definitions, notations, and assumptions as in Simple Interest activity tracker depicted in Figure 7, we can prove the main part of the algorithm by mathematical induction for principal-chaining moments, as follows:

- **The algorithm is consistent for the initial principal-changing event** $init$.

  (See Section 2.3 for Consistency Criteria.) We have to prove that the algorithm is consistent for any moment m over the moment interval $[init+, n-]$ where n is the next coming principal-chaining event. Firstly, all the four queries in the Consistency Criteria return zero; which is its true value, because the principals of all users remained zero over the interval $[init+, m]$, as there were no principal-changing actions after all principals were initialized to zero at the moment $init+$.

  Secondly, the algorithm allows consistent transfers, because $balance(user)$ is zero and the algorithm can always transfer/debit a zero amount from the asset balance of the user $user$.

- **If we assume that the algorithm is consistent for a principal-changing event** e**, then it is also consistent for the next coming principal-changing event** ne .

  (See Section 2.3 for Consistency Criteria.) If $\dot{u}$ denotes the user whose principal is changed by the event ne, this proposition is proved as follows:

  ☐ $pending(u)^{[(u.lastBlock^{[e+]})+]} = 0$, for any u u.

  Because, $pending(u)^{[r]}$, where $r = (u.lastBlock^{[e+]})+$,

  $= principals[u]^{[r]} * ((1 + rate)^{(B(r) - u.lastBlock^{[r]})} - 1)$, by the algorithm,

$P[u]^{[r]} * ((1+R)^{(B((u.lB^{[e+]})+)-u.lB^{[(u.lB^{[e+]})+]})} - 1)$

$P[u]^{[r]} * ((1+R)^0 - 1)$

$= 0$

☐ $activity^{[e+]} = \sum_{u \in U} principals[u]^{[e+]} * (1 + rate)^{(B(e+)-u.lastBlock^{[e+]})/cycle}$.
(See the algorithm state machine in Figure 9 for $activity$.)

Because, $P^{[e+]}$

$= \sum_{u \in U} pending(u)^{[e+]}$, as the algorithm is consistent at $e+$,

$= \sum_{u \in U} \{pending(u)^{[(u.lB^{[e+]})+]} + P[u]^{[(u.lB^{[e+]})+]} * ((1+R)^{(B(e+)-u.lB^{[e+]})/C} - 1)\}$

$= \sum_{u \in U} \{P[u]^{[e+]} * ((1+R)^{(B(e+)-u.lB^{[e+]})/C} - 1)\}$,

because

- $pending(u)^{[(u.lB^{[e+]})+]} = 0$, for any u u;
- if $(B(e+) - u.lB^{[m+]}) = 0$ and, so, $lB[u]^{[(u.lB^{[e+]})+]}$ does not yet exist at the moment $e+$, which is the case for the user $\dot{u}$, at least, then we can replace the multiplier $lB[u]^{[(u.lB^{[e+]})+]}$ with any value;
- if $(B(e+) - u.lB^{[e+]}) > 0$, then the users principal didn't change and $lB[u]^{[(u.lB^{[e+]})+]} = lB[u]^{[e+]}$.

$= \sum_{u \in U} P[u]^{[e+]} * (1+R)^{(B(e+)-u.lB^{[e+]})/C} - \sum_{u \in U} P[u]^{[e+]}$

$= \sum_{u \in U} P[u]^{[e+]} * (1+R)^{(B(e+)-u.lB^{[e+]})/C} - totalPrincipal^{[e+]}$

On the other hand, the algorithm returns the following value to be $totalPending^{[e+]}$:

$activity^{[e+]} - totalPrincipal^{[e+]}$.

Therefore, $activity^{[e+]} = \sum_{u \in U} P[u]^{[e+]} * (1+R)^{(B(e+)-u.lB^{[e+]})/C}$.

☐ $activity^{[ne+]} = \sum_{u \in U} principals[u]^{[ne+]} * (1 + rate)^{(B(ne+)-u.lB^{[ne+]})/cycle}$.

Because, $activity^{[ne+]}$

$= activity^{[ne-]} * (1+R)^{(B(ne-)-B(e+))/C}$
$- principals[\dot{u}]^{[ne-]} * (1+R)^{(B(ne-)-\dot{u}.lB^{[ne-]})/C} + principals[\dot{u}]^{[ne+]}$

$= (\sum_{u \in U} P[u]^{[e+]} * (1+R)^{(B(e+)-u.lB^{[e+]})/C}) * (1+R)^{(B(ne+)-B(e+))/C}$
$- P[\dot{u}]^{[e+]} * (1+R)^{(B(ne+)-\dot{u}.lB^{[e+]})/C} + P[\dot{u}]^{[ne+]}$

$= (\sum_{u \in U} P[u]^{[e+]} * (1+R)^{(B(e+)-u.lB^{[e+]})/C} * (1+R)^{(B(ne+)-B(e+))/C})$
$- P[\dot{u}]^{[e+]} * (1+R)^{(B(e+)-\dot{u}.lB^{[e+]})/C} * (1+R)^{(B(ne+)-B(e+))/C} + P[\dot{u}]^{[ne+]}$

$= (\sum_{u \in U \setminus \{\dot{u}\}} P[u]^{[e+]} * (1+R)^{(B(e+)-u.lB^{[e+]})/C}) * (1+R)^{(B(ne+)-B(e+))/C} + P[\dot{u}]^{[ne+]}$

$= (\sum_{u \in U \setminus \{\dot{u}\}} P[u]^{[ne+]} * (1+R)^{(B(ne+)-u.lB^{[e+]})/C}) + P[\dot{u}]^{[ne+]} * (1+R)^{(B(ne+)-\dot{u}.lB^{[ne+]})/C}$

$= (\sum_{u \in U} P[u]^{[ne+]} * (1+R)^{(B(ne+)-u.lB^{[e+]})/C})$

Below, we assume any moment r over the moment interval $[en+, o-]$ where o is the next coming principal-changing event after en, and prove that the algorithm is consistent at the moment r.

□ $pending(u)^{[r]}$ returns its true value for any user u.

Because, $pending(u)^{[r]}$

$= principals[u]^{[r]} * ((1+rate)^{B(r)-u.lastBlock^{[r]}} - 1)$

$= P[u]^{[(u.lB^{[r]})+]} * ((1+R)^{B(r)-u.lB^{[e+]}} - 1)$

$= (P[u]^{[(u.lB^{[r]})+]} + 0) * ((1+R)^{B(r)-u.lB^{[e+]}} - 1)$

$= (P[u]^{[(u.lB^{[r]})+]} + pending(u)[(u.lB^{[r]})+]) * ((1+R)^{B(r)-u.lB^{[e+]}} - 1)$

$= balance[u]^{[(u.lB^{[r]})+]} * ((1+R)^{B(r)-u.lB^{[e+]}} - 1)$

which is the total interest created after the latest principal-changing block, after which the user's interest was not distributed. Therefore, this is the true pending interest of the user. (We note that for the user $\dot{u}$, the latest principal-changing event is ne, and the latest principal-changing block is $B(ne+)$)

□ $totalPending^{[r]}$ returns its true value.

Because, $totalPending^{[r]}$

$= activity^{[r]} * (1+rate)^{(B(r)-B(ne+))/cycle} - totalPrincipal^{[r]}$

$= activity^{[ne+]} * (1+R)^{B(r)-B(ne+)} - totalPrincipal^{[ne+]}$

$= (\sum_{u \in U} P[u]^{[ne+]} * (1+R)^{(B(ne+)-u.lB^{[ne+]})/C} * (1+R)^{(B(r)-B(ne+))/C}) - \sum_{u \in U} P[u]^{[ne+]}$

$= \sum_{u \in U}(P[u]^{[ne+]} * ((1+R)^{(B(ne+)-u.lB^{[ne+]})/C} * (1+R)^{B(r)-B(ne+)} - 1))$

$= \sum_{u \in U}(P[u]^{[(u.lB^{[ne+]})+]} * ((1+R)^{(B(r)-u.lB^{[ne+]})/C} - 1))$

$= \sum_{u \in U}(P[u]^{[(u.lB^{[r]})+]} * ((1+R)^{(B(r)-u.lB^{[r]})/C} - 1))$

$= \sum_{u \in U} pending(u)^{[r]}$

where $pending(u)[r]$ returns its true value for all users u. Therefore $totalPending^{[r]}$ also returns its true value.

- ☐ $balance(u)^{[r]}$ returns its true value for any user u.
  Because, according to the algorithm, $balance(u)^{[r]} = principals[u]^{[r]} + pending(u)^{[r]}$, where $principals[u]^{[r]}$ is its true value because it has been accumulated with historic true values $pending(u)$; and $pending(u)^{[r]}$ is proved above to be its true value.
- ☐ $totalBalance^{[r]}$ returns its true value.
  Because, according to the algorithm, $totalBalance^{[r]} = totalPrincipal^{[r]} + totalPending^{[r]}$, where $totalPrincipal^{[r]}$ is its true value because it has been accumulated with historic true values $pending(user)$; and $totalPending^{[r]}$ is proved above to be its true value.
- ☐ The algorithm allows consistent transfers.
  Because the algorithm collects and add $pending(user)$ to the user's actual balance so that the actual balance becomes the same amount as $balance(user)$ returns, before calling the requested transfer actions, which can now transfer/debit up to $balance(user)$ amount of asset from the actual balance.

3.8 Compound Burn activity tracker algorithm

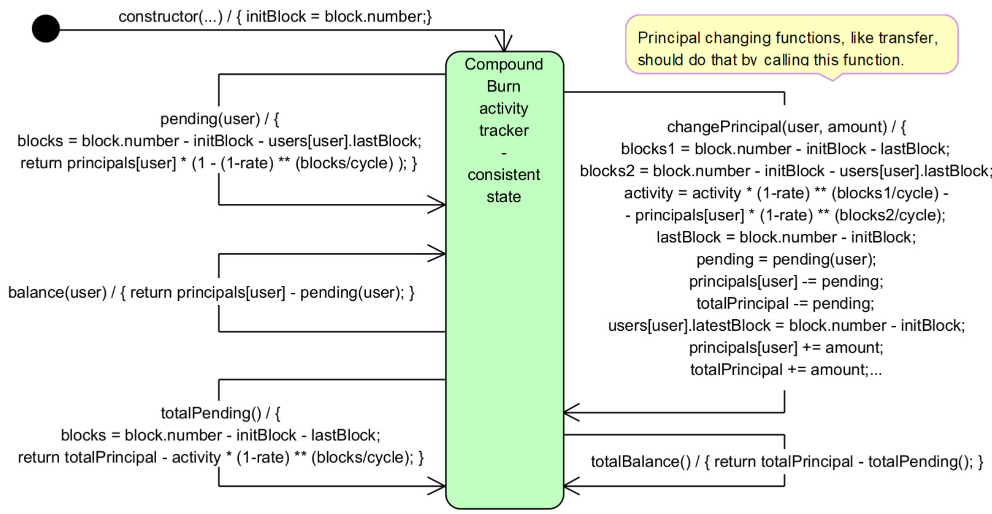

Figure 10. The UML State Machine of Compound Burn activity tracker algorithm.

See Equation 4 for the formula of Compound Burn tasks.

Figure 10 shows the state machine of the Compound Burn *activity* tracker algorithm.

Similarly to the Compound Interest activity tracker, we can prove the following properties:

- ☐ $activity^{[e+]} = \sum_{u \in U} principals[u]^{[e+]} * (1 - rate)^{(B(e+) - u.lastBlock^{[e+]})/cycle}$.
- ☐ $activity^{[ne+]} = \sum_{u \in U} principals[u]^{[ne+]} * (1 - rate)^{(B(ne+) - u.lastBlock^{[ne+]})/cycle}$.
- ☐ $pending(user)^{[r]}$ returns its true value for any user $user$.
- ☐ $totalPending^{[r]}$ returns its true value.

These propositions can be used to prove the consistency of the algorithm by induction for principal-changing moments, as in Section 3.7

3.9 Random algorithms

The algorithms have been proved to be consistent at any moment assuming there are no computer numerical errors, which is not the case. This section discusses mitigating accumulated computer numerical errors.

Losses coming from fitting exponentiation of real numbers into a quotient of unsigned integers are exponentiation errors, while losses coming from fitting division of integers into an unsigned integer are division errors. We propose an idea of improved algorithms that can mitigate accumulated division errors below.

See Listing 2 for how we identify and handle numerical errors in our Solidity implementation of algorithms.

```
function pending(address user) public view returns () {
  uint pending = 0;

  uint blocks = block.number - initBlock - users[user].lastBlock

  if (blocks > 0) {
      # Exponentiation error source: p/q = (1+r)**(blocks/cycle), r = rate / scale.
      (uint p, uint q) = analyticMath.pow(scale + rate, scale, blocks, cycle);

      # Division error source:
      pending = principals[user] - principals[user] * p / q;

      # We handle the division error source by replacing it with this alternating block.
      if (block.number % 2 == 0) {
          pending = principals[user] - IntegralMath.mulDivF(principals[user], p, q);
      } else {
          pending = principals[user], IntegralMath.mulDivC(principals[user], p, q);
      }
      # mulDivF, or multiply_and_divide_returning_floor, returns left-biased quotients,
      # while mulDivC, or multiply_and_divide_returning_ceiling, returns right-biased quotients.
  }

  return pending; # The returned values are accumulated to balance(user)
}
```

Listing 2. Handling division errors in our Solidity implementation of algorithm. We identify two numerical error sources: the exponentiation error and the division error. When $rate = 0.000474$, the whole 377,000 accumulated exponentiation errors are collectively small enough if users change their principal frequently and, so, if their exponents are small. As for the division errors, we adopt a technique that alternatingly chooses between a quotient biased to a smaller value and a quotient biased to a larger value, allowing the hidden division errors to cancel each other.

There are two types of numerical errors for our algorithms when computers are operating in *integers*, as is the case in Solidity programing language:

- Exponentiation.
  Exponentiation errors come in two folds:
  - ◇ Interest exponentiation errors: $|(1+r)^{blocks/cycle} - \{(1+r)^{blocks/cycle}\}|$
  - ◇ Burn exponentiation errors: $|(1-r)^{blocks/cycle} - \{(1-r)^{blocks/cycle}\}|$
- Division error.
  $|i/j - \{i/j\}|$ for any positive integers i and $j \neq 0$.

where $\{operation\}$ means the, theoretically existing, true return value of the computer operation $operation$, whereas $operation$ means the actual return value of the computer operation $operation$. The true value is achieved only if there are no numerical errors, which is often not the case.

In order to mitigate exponentiation errors, we incorporate a 3rd-party mathematics library, called *AnalyticMath*, as shown in Listing 2.

The library is used as follows:

```
(p, q) = analyticMath.pow(a, b, c, d)
```

for unsigned integers a, b, c, d, p and q, where p and q are intended to satisfy:

```
{p / q} is as close to {(a / b) ^ (c / d)} as possible
```

The provider of the library analyticMath assures that their found p and q only satisfy:

```
If a > b, then {p / q} < {(a / b) ^ (c / d)}
If a < b, then {p / q} > {(a / b) ^ (c / d)}
```

Errors that are always less than zero or always larger than zero, like exponentiation errors, have no chance to cancel each other when they are accumulated. Our algorithms demonstrate that 377,000 exponentiation errors, accumulated through the same number of transactions, give an insignificant collective error when exponents are small. Exponents are usually small if most users change their principal frequently.

Exponentiation errors are discussed more in Section 4, where test results suggest an intuition of exponentiation error as follows:

$$|\{p/q\} - \{(a/b)^{(c/d)}\}| = \{alpha * (a/b)^{(c/d)}\} \tag{15}$$

for some small constant $alpha$.

We leave mitigating exponential errors over an extensively long operation and an extremely large number of transactions, to subsequent research, because it requires significantly more work.

In order to mitigate division errors:

- We choose a 3rd party integer division algorithm, rather than Solidity's unsigned integer division. The $IntegralMath$ library provides $mulDivF$ and $mulDivC$ operations, which, respectively, returns the floor integer and ceiling integer of $\{integerA * integerP/integerQ\}$.
- We then alternatingly choose between $mulDivF$ and $mulDivC$, or between a negatively biased value and a positively biased value, so that the errors can cancel each other when they are accumulated to $balance(user)$ throughout the application's operation.

After some mathematical work, we can achieve the following equations:

- If $\sum_{u \in U} balance(u)$ is evaluated with Solidity's floor-returning integer division, and if N denotes the number of accumulation of $pending(user)$, then

$$N/4 < \mathbb{E}(|\sum_{u \in U} balance(u) - \{\sum_{u \in U} balance(u)\}|) < N \tag{16}$$

- If $\sum_{u \in U} balance(u)$ is evaluated with the 3rd-party alternating integer divisions, then

$$\mathbb{E}(|\sum_{u \in U} balance(u) - \{\sum_{u \in U} balance(u)\}|) \leq 1 \tag{17}$$

where $\mathbb{E}$ denotes the expected value and N is the number of principal-changing transactions. Equation 16 indicates that the floor-returning integer division gives errors that diverge slowly linearly, while Equation 17 means the alternating integer divisions give errors that converge to zero, both in an appropriate meaning. As for $totalBalance()$, the proof should be similar as for $\sum_{u \in U} balance(u)$.

Algorithms that mitigate division errors by using the 3rd-party alternating integer divisions are called **random algorithms**. In random algorithms, the long-term behavior of errors should solely be determined by the exponentiation error source. See Table 2 for the classification of algorithms.

Task type	Algorithms that don't handle errors	Random algorithms that handle division errors
Simple Interest	Simple Interest Pendency tracker Simple Interest Activity tracker	Simple Interest Pendency random tracker Simple Interest Activity random tracker
Simple Burn	Simple Burn Pendency tracker Simple Burn Activity tracker	Simple Interest Burn random tracker Simple Burn Activity random tracker
Compound Interest	Compound Interest Pendency tracker Compound Interest Activity tracker	Compound Interest Pendency random tracker Compound Interest Activity random tracker
Compound Burn	Compound Burn Pendency tracker Compound Burn Activity tracker	Compound Interest Burn random tracker Compound Burn Activity random tracker

Table 2. Classification of algorithms

We note, however, N, as both the number of principal-changing transactions and the upper limit of accumulated division errors, is a *negligibly small* number compared to the magnitude of $totalBalance()$. Looking at the existing busiest DeFies, the total number of transactions during their past existence is less than a couple of millions (10^6), while the magnitude of $totalBalance()$, which acts as the denominator in Relative Errors A and B, is usually over 10^{18+5}. This means the accumulated division errors will have trivial effect on *Relative* Consistency Errors, which is the main performance measure. Therefore the random algorithms will *not* give a significant improvement of accuracy, if we don't have extensively many transactions. We just propose random algorithms and do not concentrate on them in our test, as our subsequent research is expected to concentrate on errors.

4 Tests

See Section 2.1 for the definitions of task types. See Section 3.5 for the concepts of pendency and activity. See Section 3.9 for random algorithms.

We perform simple stress tests for them, for the *purpose* of checking if the algorithms can operate in *diversified environments* for *longer periods* and *consistently.* The diversity is achieved by using randomly generated transactions in various test modes, the longevity is tested by running tests a long time, and the consistency is checked by using our Consistency Criteria defined in Section 2.3. The single most important concept in testing is Consistency Errors, defined in Equation 10 through Equation 12.

4.1 Baseline

We introduce $javascriptTruth$ and $solidityTruth$ for testing our algorithms. Our tests aim to assess Consistency Errors, which by and large come from accumulating $pending(user)$, $pendency$, and $activity$. Exponentiation errors and division errors identified above, which are trivial individually, should be concerned because they are accumulated over transactions, with their values biased harmfully in a single direction. For example, $totalBalance()$ is essentially an accumulation formulated as

$$totalBalance()_i = totalBalance()_{i-1} * (1 + rate)^{(B(e_i+)-B(e_{i-1}+))/cycle}, \tag{18}$$

in Compound Interest tasks, where e_i is i^{th} principal-changing events. Every round of the successive accumulation brings a numerical error into $balance(user)$ and $totalBalance()$, and the errors are harmfully biased in a single direction not canceling each other. If there are no numerical errors, then Equation 18 can be simplified to:

$$totalBalance()_i = totalBalance()_0 * (1 + rate)^{(B(e_i+)-B(e_0+))/cycle} \tag{19}$$

In Simple Interest tasks, the corresponding two equations are, respectively:

$$totalBalance()_i = InitTotal * \sum_{i=1}^{N} (1 + rate) * (B(e_i+) - B(e_{i-1}+))/cycle \tag{20}$$

$$totalBalance()_i = InitTotal * (1 + rate) * (B(e_N+) - B(e_0+))/cycle \tag{21}$$

The Equations 18 and 20 are a *return value* of the algorithms, while Equations 19 and 21, which have no accumulation, are almost free of error and very near the *true value* of query $totalBalance()$. We use Equations 19 and 21 as a substitute for $TrueTotal$ defined in Equation 10, in their respective task types. To further reduce errors, we calculate the $TrueTotal$'s substitutes in Javascript programing language, rather than in Solidity language, which operates in unsigned integers creating larger errors. The Javascript version of the $TrueTotal$'s substitutes is called $javascripTruth$ below. For the comparison purpose, we also have the Solidity version of $TrueTotal$'s substitutes and call them $solidityTruth$.

4.2 Testing Procedure

To generate simulated diversified environments, we create an automatic testing program that acts as follows:

- Four simulated users - Owner, Alice, Bob, and Carol; are created on a private block chain, each with enough cryptocurrency for gas fee payment.
- A smart contract that implements the target algorithm, as well as the four queries in Consistency Criteria, is deployed on the block chainwork, with 18 decimal places as usual. See Section 2.3 for Consistency Criteria.
- The smart contract implements transfer, mint, and burn functions on the principal amount of users, by using the $changePrincipal(user, amount)$ function offered by the algorithm.
- The smart contract's constructor mints 10^8 tokens to Owner as the initial amount for the principal of Owner. An interest/burn rate of 0.0474 % a simulated day is assumed, which is equivalent to 1 % interest every 21 days in Compound Interest tasks. One simulated day spans 10 blockchain blocks.
- $transfer$, $mint$, and $burn$ transactions (called simply a function below) are raised randomly from the off-chain part with randomly chosen arguments, like $user$ and $amount$, while the $mintBlocks$ function is called intermittently to advance the block number (or, the internal time) in the block chain, again with a randomly chosen number of blocks to advance. Fixed probability distributions over the functions' occurrences, over $user$, and over $amount$, respectively, are assumed.

  Once a randomly chosen function is called, the function repeatedly tries randomly changing $user$ and $amount$, up to 50 times until it succeeds, thus adhering to the given probability distribution over function occurrences. We note the $transfer$ transaction, for example, may well fail, because the principal amounts for all users may become almost zero after a Compound Burn task runs a long time with a significant burn rate.
- Typically, 200,000 calls are made in a test. $Transfer$ transactions account for 90 % of the total calls, and the remaining part is accounted for by $mint$, $burn$, and $mintBlocks$. In each test, up to 468,000 blocks

representing 46,800 simulated days or 128 simulated years are minted by $mintBlocks$ calls. 1 to 50 blocks are minted by a $mintBlocks$ call, representing 1 tenth day to 5 days minted by a $mintBlocks$ call.

- As random functions are called, the smart contract and testing program cooperate to calculate $solidityTruth$, $javascripTruth$, and Consistency Errors. Sec Section 4.1 for $solidityTruth$ and $javascripTruth$, and Section 2.3 for Consistency Errors.

The testing program has two modes: Free Total Principal test mode and Fixed Total Principal test mode.

- Free Total Principal test mode

  In this mode, the testing program does not call $mint$ and $burn$ transactions, leaving the total principal or reward amount to freely change according to their formulas shown in Section 2.1.

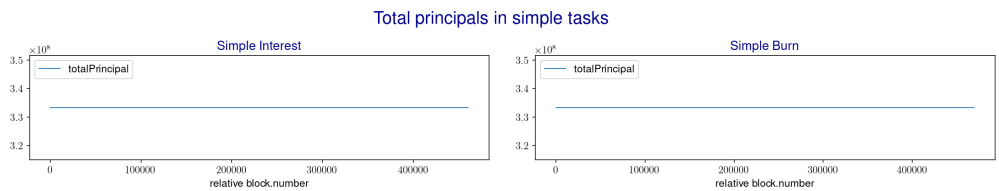

Figure 11. For a simple task in the Free Total Principal test mode, $totalPrincipal$ keeps constantly to its initial value.

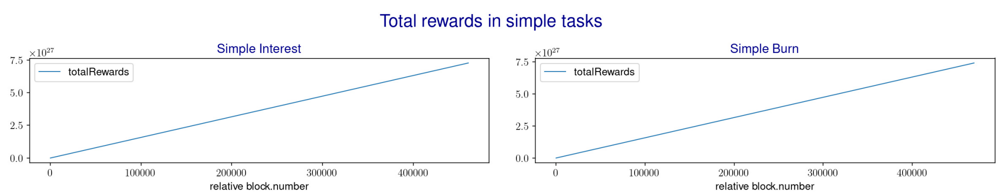

Figure 12. For a simple task in the Free Total Principal test mode, $totalRewards$ grows freely linearly, because the time-linear interest or burn is additively accumulated to $reward[user]$.

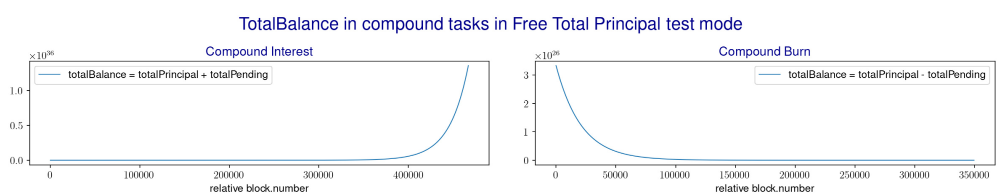

Figure 13. $totalBalance()$ for compound tasks in Free Total Principal test mode grows/shrinks freely exponentially from its initial total principal as much as time goes and $rate$ allows, because the time-exponential interest/burn *is* compounded to/from $principals[user]$.

  We note $TrueTotal$, defined in Equation 10, acting as the denominator in Relative Errors A and B, may get extremely large in a Compound Interest tasks or get extremely small in a Compound Burn task, affecting the Relative Errors to diminish or diverge (unless their numerators change faster in the same direction.) This mode aims to simulate an extreme operation where the total principal is not managed/limited by system administrators and observe how consistent the algorithms are in those harsh conditions.

- Fixed Total Principal test mode

In this mode, the testing program resists change of total principal by choosing a suitable value for the $amount$ argument of randomly called $mint$ or $burn$ transactions. The incremental changes to the total principal amount in Compound Interest tasks or the decremental changes of the total principal in Compound Burn tasks are compensated by the suitable $amount$ arguments passed to the $mint$ or $burn$ transactions, keeping the total principal to its initial value.

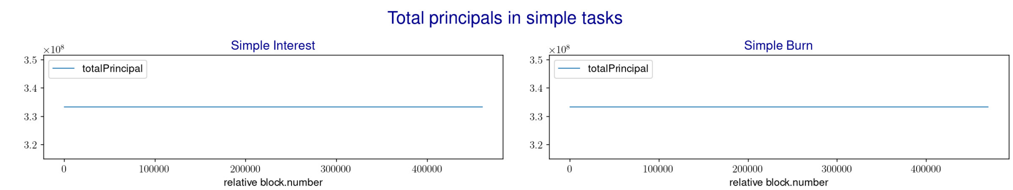

Figure 14. For a simple task in the Fixed Total Principal test mode, $totalPrincipal$ keeps constant to its initial value. While the testing program tries to keep $totalPrincipal$ fixed (with $mint$ or $burn$ transactions from offchain), every $principals[user]$, so $totalPrincipal$ too, is already fixed, because the (linear) interest or burn is *not* compounded to $principals[user]$.

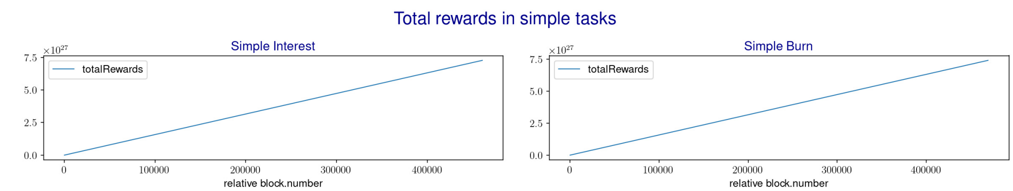

Figure 15. For a simple task in the Fixed Total Principal test mode, $totalRewards$ grows freely linearly, because the linear interest or burn is additively accumulated to $reward[user]$.

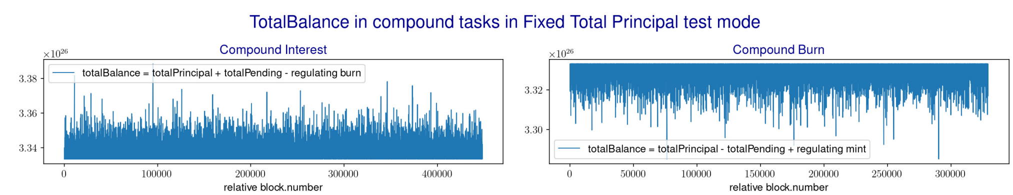

Figure 16. For compound tasks in the Fixed Total Principal test mode, $totalBalance$ is regulated by the testing program with $mint$ or $burn$ transactions, so that $totalBalance$ reverts to its initial value frequently.

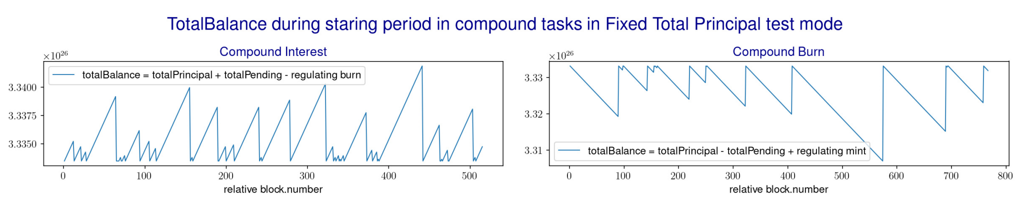

Figure 17. For compound tasks in the Fixed Total Principal test mode, the testing program regulates $totalBalance$, which would otherwise grow/shrink freely as in Figure 13, by pulling it down/up to its initial value intermittently.

This mode aims to simulate a modest operation where the total principal is completely managed to be stable, as will be the case in many applications, by system administrators, and confirm how consistent the algorithms are in those typical conditions.

4.3 Test cases

Test cases are as follows:

- Simple tasks in Free Total Principal mode
  Simple tasks types, which are Simple Interest and Simple Burn, are tested each in its two alternative algorithms: pendency tracker and activity tracker, in the Free Total Principal test mode.

- Compound tasks in Free Total Principal mode
  Compound tasks types, which are Compound Interest and Compound Burn, are tested each in its two alternative algorithms: pendency tracker and activity tracker, in the Free Total Principal test mode.

- Simple tasks in Fixed Total Principal mode
  Simple task types, which are Simple Interest and Simple Burn, are tested each in its two alternative algorithms: pendency tracker and activity tracker, in the Fixed Total Principal test mode.

- Compound tasks in Fixed Total Principal mode
  Compound task types, which are Compound Interest and Compound Burn, are tested each in its two alternative algorithms: pendency tracker and activity tracker, in the Fixed Total Principal test mode.

4.4 Test case: Simple Tasks in Free Total Principal mode

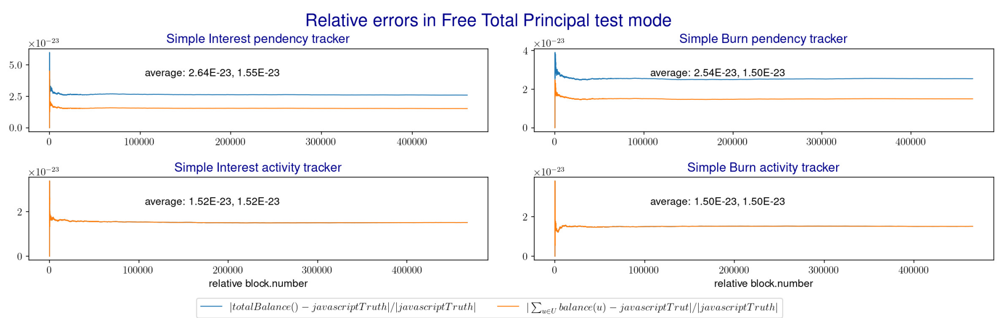

Figure 18. Relative Errors A and B in simple tasks in Free Total Principal test mode. The Relative Errors converge and are less than 10^{-22} during 128 simulated years and 180,000 transfer transactions.

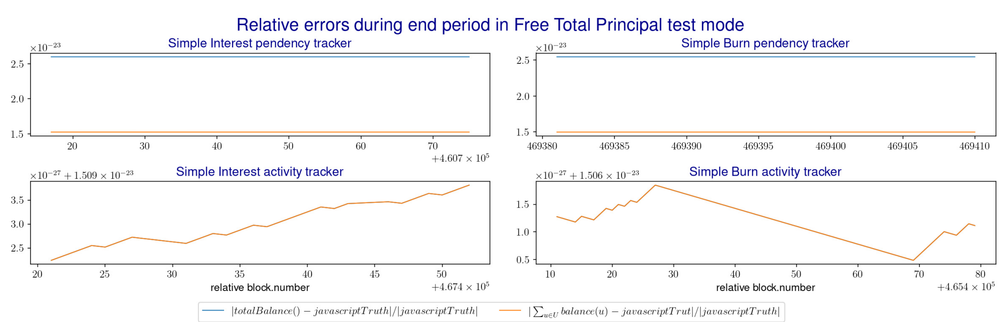

Figure 19. Relative Errors A and B, after a long run, for simple tasks in Free Total Principal test mode. For pendency trackers, $totalBalance$ and $\sum_{u \in U} balance(u)$ reveal significant deviation from their shared substitute true value $javascriptTruth$. For activity trackers, the two values are fluctuating.

4.5 Test case: Compound Tasks in Free Total Principal mode

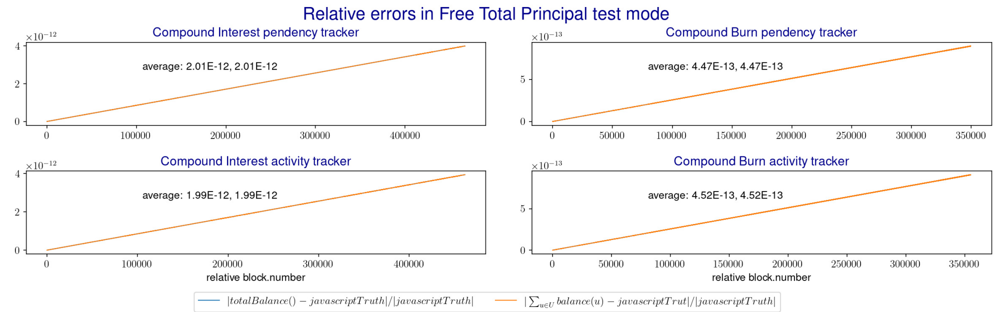

Figure 20. Relative Errors A and B for compound tasks in Free Total Principal test mode. The Relative Errors grow linearly over time and are less than 10^{-11} during 128 simulated years and 180,000 transfer transactions.

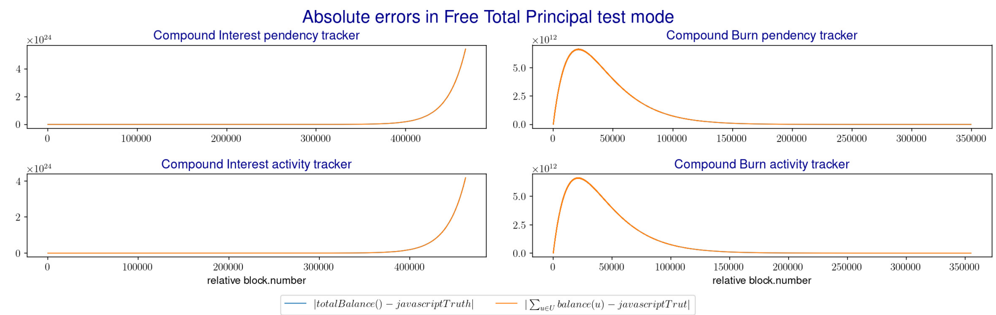

Figure 21. Absolute Consistency Errors for compound tasks in Free Total Principal test mode. They have exponential growth as suggested by the linearity of Relative Errors A and B.

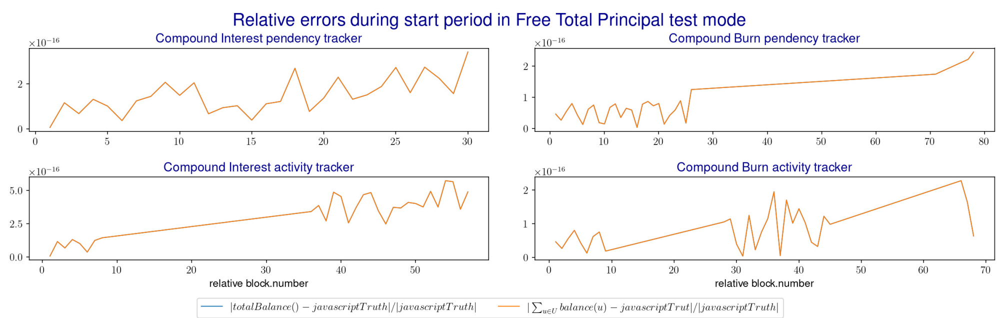

Figure 22. Relative Errors A and B, during the starting period, for compound tasks in Free Total Principal test mode. The two Relative Errors are not distinguishable from each other on this small-resolution plot.

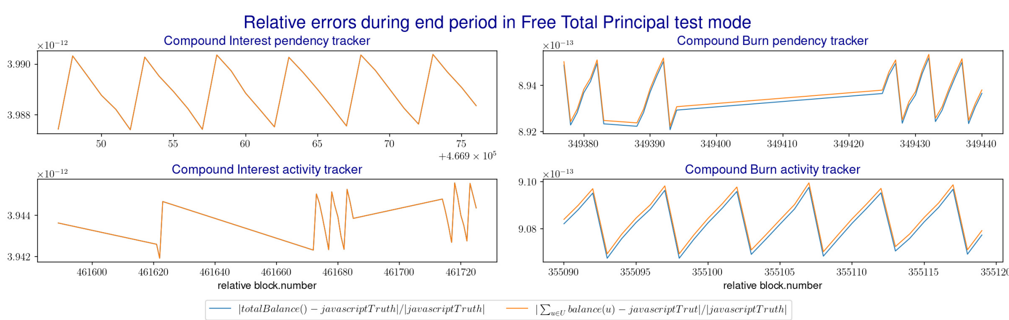

Figure 23. Relative Errors A and B, after a long run, for compound tasks in Free Total Principal test mode. The two Relative Errors are barely distinguishable from each other for burn tasks.

4.6 Test case: Simple Tasks in Fixed Total Principal mode

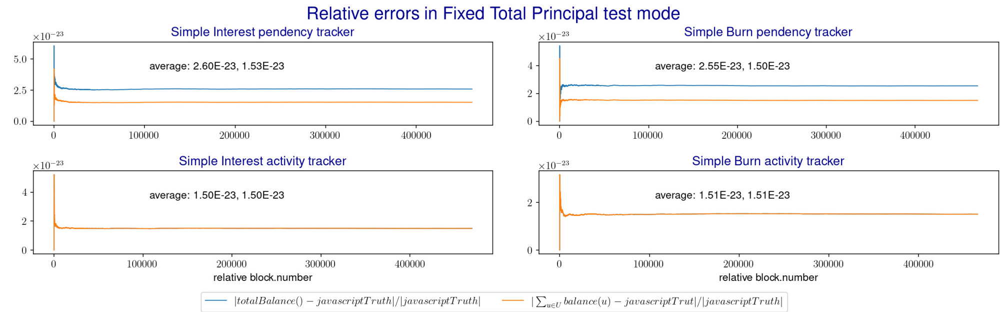

Figure 24. Relative Errors A and B for simple tasks in Fixed Total Principal test mode.

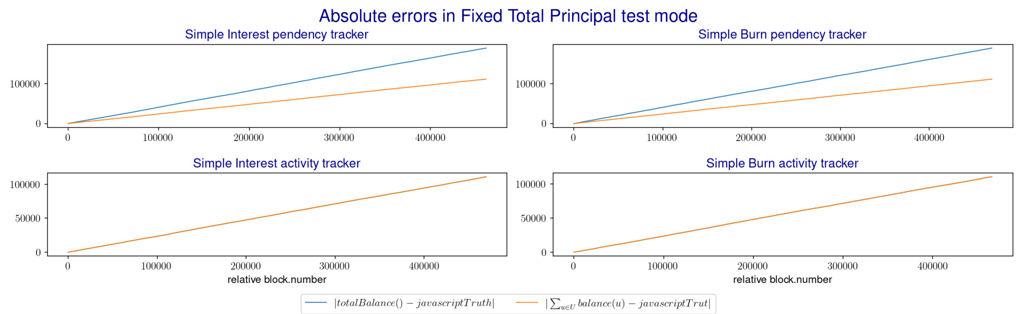

Figure 25. Absolute Consistency Errors for simple tasks in Fixed Total Principal test mode. Activity tracker algorithms show little, if not no, absolute errors.

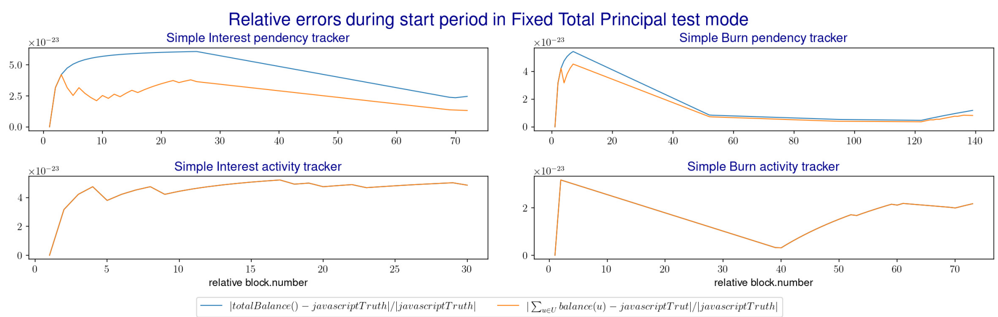

Figure 26. Relative Errors A and B, during the starting period, for simple tasks in Fixed Total Principal test mode.

4.7 Test case: Compound Tasks in Fixed Total Principal mode

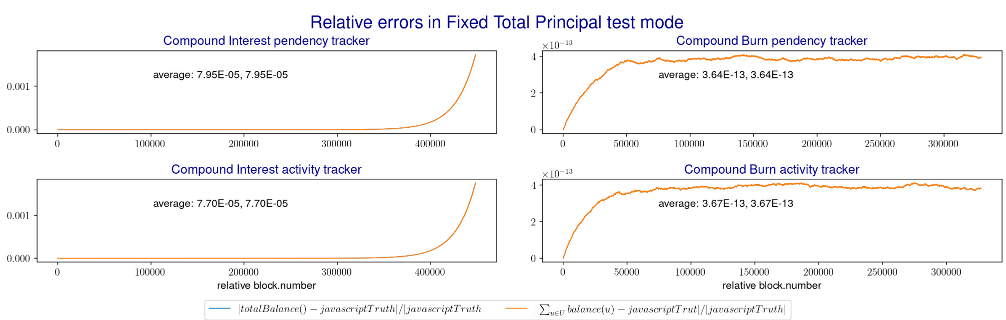

Figure 27. Relative Errors A and B for compound tasks in Fixed Total Principal test mode. While Relative Errors are not growing and lower than 10^{-12} in burn tasks, they are diverging and as high as 0.002, after a 128-year-long simulated run.

We guess the errors diverge, in Figure 27, if the exponent is more than 1, as in interest tasks, where the exponent is $1 + rate$; and converge if the exponent is less than 1, as in burn tasks, where the exponent is $1 - rate$. The interest rate $rate$ in this test scenario is 0.000474, which is equivalent to about 1% every 21 days.

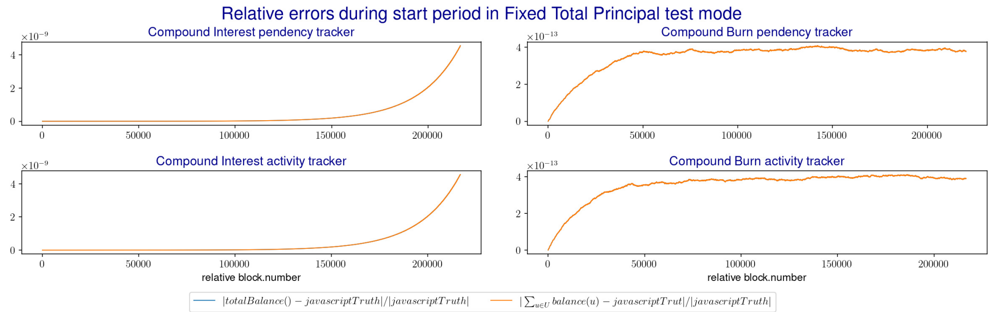

Figure 28. Relative Errors A and B for compound tasks in Fixed Total Principal test mode.

In Figure 28, unlike in Figure 27, the chart spans a half time period, which is a little more than 60 simulated years. The Relative Errors A and B for interest tasks are lower than 10^{-8}, which is within the safe level.

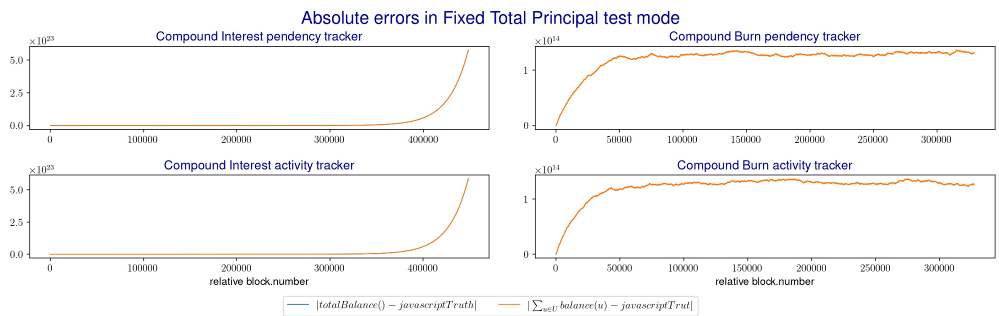

Figure 29. Absolute Consistency Errors for compound tasks in Fixed Total Principal test mode.

We guess, on the left of the chart of Figure 29, that the exponentiation errors are time-exponential, which are successively accumulated to form time-exponential errors.

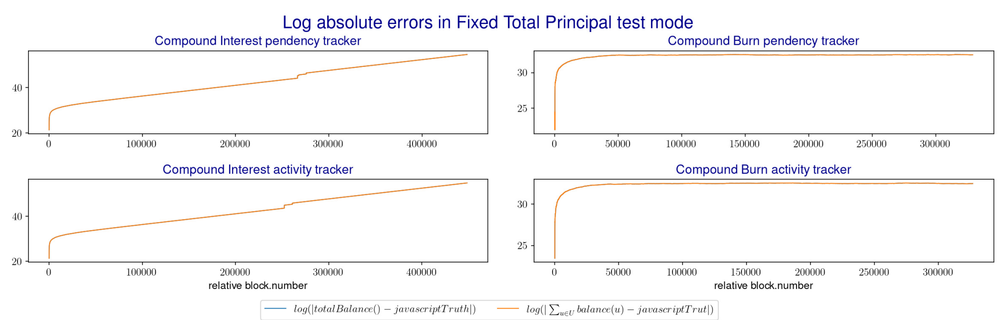

Figure 30. Log Absolute Consistency Errors for compound tasks in Fixed Total Principal test mode. The straight log lines on the left of the chart confirms that the exponentiation errors are time-exponential.

5 Conclusion

We propose, prove, and demonstrate algorithms that solve Simple Interest, Simple Burn, Compound Interest, and Compound Burn reward distribution tasks. The algorithms can distribute rewards to an unknown number of users, adhering to the computational quota if there are no computer numerical errors. Although computer numerical errors are individually trivial, they collectively create a large deviation because the algorithms inherently and constantly accumulate amount figures that have numerical errors. Table 3 shows Relative Errors A and B in various types of task and test modes.

Exponentiation errors turn out to be trivial if users' rewards are collected frequently, but pose harmful if they are accumulated over an extensively long period or a large number of transactions. Division errors are also proved to be small enough to ignore compared to the huge magnitude of asset amount figures, unless we have an extremely large number of transactions.

Task type	**Free Total Principal test mode**	**Fixed Total Principal test mode**
Simple Interest	Keeps around tiny values, below 10^{-22}	Keeps around a value, below 10^{-22}
Simple Burn	Keeps around tiny values, below 10^{-22}	Keeps around tiny values, below 10^{-22}
Compound Interest	*Diverges* slowly linearly, $4{*}10^{-12}$ after 128 years	*Diverges* slowly but *exponentially*, $2 * 10^{-2}$ after 128 years, $5 * 10^{-9}$ after 60 years
Compound Burn	*Diverges* slowly linearly, 10^{-12} after 128 years	Converges to a small value, below 10^{-12}

Table 3. Relative Errors A and B in our test scenario.

We leave mitigating numerical errors to subsequent research, because that will require a significant amount of extra work.

We introduce new concepts and notations that can be reused in rigorous reasoning of decentralized techniques. We compare verbal proof and symbolic proof of decentralized algorithms and demonstrate symbolic proof may be more thorough and effective.

Acknowledgments

We are deeply indebted to Calum Roberts, Dong-Zhe Lian, and Xavier Mitchell-Diggens for their heartfelt encouragement, professional reviews and proofreading, and excellent expertise in DeFi and academic writing. This endeavor would not have been possible without Zheng-Yu Cai, who provided warm care, financial support and facilities, and constant encouragement for the research project. I, the 1^{st} author, would like to express my deepest gratitude to my Mom, Dad, and siblings for all the support and encouragement. I would like to thank my wife for her loving presence, unwavering support, and belief in me.

References

[1] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* URL: `https://bitcoin.org/bitcoin.pdf`.

[2] PancakeSwap. *Distributed trading protocol on BNB Smart Chain.* URL: `https://github.com/pancakeswap/pancake-farm/blob/master/contracts/MasterChef.sol`.

[3] Barak Manos. *Solidity Math Utilities.* URL: `https://github.com/barakman/solidity-math-utils#integralmath`.

[4] Igor Makarov and Antoinette Schoar. "Cryptocurrencies and Decentralized Finance (DeFi)". In: *SSRN Electronic Journal* (Jan. 2022). DOI: `10.2139/ssrn.4104550`.