Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Implementation and Evaluation of a Reputation Framework to Increase Reputation Accuracy

Márton Donát Nagy

Master’s Thesis

Implementation and Evaluation of a Reputation Framework to Increase Reputation Accuracy

Master's Thesis

Supervised by Prof. Dr. phil. nat. Sebastian Steinhorst
Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Advisor Emanuel Regnath

Author Márton Donát Nagy
Schwere-Reiter-Str. 35./420
80797 Munich

Submitted on August 5, 2021

Declaration of Authorship

I, Márton Donát Nagy, declare that this thesis titled "Implementation and Evaluation of a Reputation Framework to Increase Reputation Accuracy" and the work presented in it are my own unaided work, and that I have acknowledged all direct or indirect sources as references.

This thesis was not previously presented to another examination board and has not been published.

Signed:

Date:

Abstract

The advent of technologically scaled transactional spaces and the explosion of user numbers in them raises the problem of who participants should trust. Along human oriented systems like online marketplaces, algorithmic environments are becoming more and more prevalent, where participating agents are autonomous machines. Such domains are the Internet of Things and autonomous driving.

Trust and reputation research aims to provide systems to solve the trust problem. The research field is divided as countless approaches have been proposed in various domains. Many approaches repeat certain patterns to increase reputation accuracy and make the system more resistant against malicious attacks. Our goal is to collect and evaluate the usefulness of these building blocks.

We propose a generic model for reputation systems, and using it we implement a Python reputation evaluation framework called Pyrepsys. To our knowledge, Pyrepsys is the only evaluation framework to support modular reputation improvement methods.

We collected promising improvement approaches from literature. From these, we evaluated the use of aging and weights. Experiments showed that aging makes less sense when agent behavior never changes. For weights to work, the reputation calculation needs to give reputation to pure rater agents as well.

Finally, we identified further ways to improve the Pyrepsys system.

Contents

1

Introduction

Trust and reputation are an inherent part of our biology and social behavior, helping us navigate human-scaled communities. With the advent of technologically scaled transactional spaces, the problem of who to trust quickly becomes an issue, as in an online marketplace for example. Similar problems also arise in networks partly or fully used by machines, such as in decentralized Internet of Things or autonomous driving. Cryptographic methods can guarantee the identity of agents and the integrity of their communication and actions. They cannot, however, tell if what agents say is true or reliable, or whether they act in good faith or not. Trust and reputation systems aim to fulfill this purpose.

1.1 Problem Statement

Countless reputation algorithms and reputation system evaluation frameworks are published in the literature. The problem someone setting up a reputation system faces is the overabundance of methods, and the lack of domain-independent comparison.

Existing reputation systems already contain building blocks that can be reused in a different system to improve its reputation estimate's accuracy. We would like to have a commonly available collection of such improvement techniques, along with their potency of improving reputation methods. We also see the need for a generic evaluation framework that can be used to compare improvement techniques and select which methods make sense to use in various applications. Currently no such system exists.

1.2 Scenario

We aim to stay general in terms of domains and environments. There is however a prioritization of algorithmic environments like Internet of Things, more specifically intersection management in autonomous driving. While it serves as a recurring example throughout this thesis, the models and simulations are not at all limited to this specific use case. Methods and experiments are intended to be general enough that they apply for internet of things (IoT), cyber-physical system (CPS), or any other application as well.

The autonomous driving example assumes a hypothetical algorithm governing intersections, regulating the order vehicles pass crossways. Vehicles are assumed to provide an estimated time of arrival (ETA) to the intersection, based on which the passing order is determined. Vehicle agents may be tempted to report a shorter ETA to gain an earlier right of way priority. The use case for a reputation system is to keep these attempts in check by giving each vehicle agent a reputation.

1.3 Task Definition

We will

- take a survey of reputation methods
- identify independent patterns in literature which can be used as improvement techniques
- take a survey of reputation evaluation frameworks
- see if any published reputation evaluation framework can be used or adapted to compare improvements
- or if not, design and implement a new evaluation system

1.4 Structure of This Document

First, Chapter 2 covers the basics of trust and reputation systems, and explores existing reputation improvement techniques and published evaluation frameworks. Chapter 3 describes the approach used to model and compare reputation improvements. Then, Chapter 4 details the implementation of the introduced model. Results of simulations made with the implemented simulator are shown and discussed in Chapter 5. Finally, Chapter 6 concludes this thesis.

2

Background and Related Work

This chapter gives an overview of related trust and reputation research. First, reputation and trust is introduced, including a quick outlook into usage within the IoT domain. Then, published techniques for reputation system improvement are explored. Most of these revolve around detecting and removing unfair ratings, or mitigating certain specific attack scenarios. Finally, existing reputation system evaluation frameworks are taken into account.

2.1 Trust and Reputation

Researching trustability has been relevant in various domains for decades. With network opening up and growing, more users and things who do not share history are interacting. As new domains move to be integrated into networks and put online, more areas of life face the trustability problem. Online trading or peer-to-peer (p2p) file sharing has been around for decades now, while autonomous driving, sensor systems or widespread connected embedded devices are more recent to face the issue of who to trust in a network.

Domains face different, but still similar challenges. In p2p file sharing, clients are constantly changing. Some with malicious intent may want to poison the shared data in the network.[1] Any method to combat this must be completely decentralized as per p2p protocols. Online trading has also developed its methods to increase safety or at least a sense of it. Shops and marketplaces extensively use user ratings, reviews and other methods of social proof to facilitate trust and aid the purchase decisions of customers. The marketplaces themselves can serve as a central authority, and such systems are highly centralized since the marketplace is trusted by all participants[2].

We all have an innate understanding of trust, usually connected to reliability. Trust can have two forms: reliability trust and decision trust[2, 3]. Reliability trust is defined by

Gambetta in [4] and paraphrased by Jøsang et al. in [3] as

Definition 2.1 (Reliability trust) *Trust is the subjective probability by which an individual, A, expects that another individual, B, performs a given action on which its welfare depends.*

As for decision trust, Jøsang et al. gives a definition in [3] as

Definition 2.2 (Decision trust) *Trust is the extent to which one party is willing to depend on something or somebody in a given situation with a feeling of relative security, even though negative consequences are possible.*

Both definitions show that trust is a directed relational property between two individuals. On the other hand, reputation is generally understood as a common and public judgment of an individual's reliability. More broadly, Jøsang et al. gives the definition from the Concise Oxford dictionary as

Definition 2.3 (Reputation) *Reputation is what is generally said or believed about a person's or thing's character or standing.*

Trust and reputation systems are designed to use these concepts in order to support decision making. A decision can be for example who to transact with, which seller or service provider to choose, whether to accept data from a specific source or not, etc. The participating actors that make decisions are are called agents, or in specific contexts also users, peers or nodes. Agents can be people, machines or algorithms.

Trust management systems calculate trust relationships between individual agents. Trust values are most commonly based on experience from previous interactions or third party opinions from other agents, usually called advisors or witnesses.

Reputation systems maintain a public reputation value for each agent, which can be used to assist agent decisions. A trusted central authority is often in charge of maintaining global reputation values, but there are decentralized methods as well, for example in p2p networks[1, 5, 6].

The focus of this thesis is reputation systems. Because of this, the rest of this work gives less consideration for the components and special cases of trust-based systems.

2.1.1 Trust and Reputation in the Internet of Things

Due to the decentralized nature of IoT, many approaches opt for a more trust-focused approach. MARINE is a trust model for vehicular ad-hoc networks (VANET) with two stages of evaluating trustworthiness[7]. First, previous interactions and recommendations from neighboring vehicles are considered. Then, data from the vehicle under evaluation is checked for

- information quality

- vehicle's message forwarding capability
- opinions from neighbors

The evaluator vehicle drops data from the evaluated vehicle if any of the steps are failed. Another approach for VANETs is found in [8]. It proposes a trust method to mitigate a specific type of attack named Black Hole. This attack consists of malicious nodes taking part in routing and dropping all data packets without forwarding them.

SecTrust is a trust-based method that provides secure communication and routing in the network, and isolates malicious or selfish nodes[9]. Prior experience and other advisor node's opinions are both used to calculate trust values.

The work in [10] introduces a reputation-based method for sensor networks. Nodes privately store a reputation-like value from a subset of other nodes in the network, which is based on prior experiences of the nodes. Together with opinions from other nodes, this forms the basis of trust decisions. The system is distributed, so no central authority is required.

The authors of [11] conducted a survey among trust and reputation method in the field of IoT, also exploring basic concepts, taxonomy and commonly considered attacks.

2.2 Reputation Systems Improvement Techniques

Various papers propose methods to make reputation systems more resistant against attacks and biased or unfair ratings, and improve the accuracy of reputation calculation. The work in [12] suggests four capabilities a complete improvement approach should have:

- *majority*: still work if majority is unfair or malicious
- *flooding*: function under sudden onset of ratings in a short time
- *lack of experience*: still work when agents do not have any prior connections or private experience
- *varying*: agent behavior change should be recognized

2.2.1 Categorization of Methods

Whitby et al. group methods to identify and remove unfair ratings into two categories: *endogenous and exogenous* approaches[13]. Endogenous refers to methods where only the rating values are analyzed for statistical anomalies. Exogenous methods consider factors other than the ratings themselves for detecting unfair ratings. Such external factors are for example the reputation of the rater, in which case the assumption is that low-reputation users are more likely to give unfair ratings.

Reactive-proactive categorization is introduced in [14]. A method is reactive if it identifies existing, already published unfair feedback. Opposed to this, a proactive methods proposes incentives for agents to encourage them to leave fair feedback, so that unfair feedback is prevented before given.

The authors of [12] use two other categories apart from endogenous-exogenous: *public-private* and *global-local*. These are adopted specifically for e-commerce-type scenarios where local trust (past experiences) and local advisor opinions both play a role.

Public and private are differentiated along how advisor opinions are judged for trustworthiness. In private methods, agents evaluate an advisor's opinion based on whether past opinions received by the same advisor turned out to be correct or not. Data for evaluation is limited to the specific advisor's interactions with the specific agent in the past. In public methods, agents judge advisors based on the advisor's ratings across the whole system in the past. Here, all previous opinions of the advisor and their subsequent results are taken into account, not just those which were provided for the agent currently doing the evaluation.

Local refers to methods where looking for received unfair ratings of an agent is based on all the ratings that agent received, but not other agents. For example, if seller typically gets reviews around 4 or 5 out of 5, a 1-rating raises suspicion. In a sense, opinions are isolated between agents. In *global methods*, advisor or rater trustworthiness is judged using ratings on all the agents in the system.

Although defined for an e-commerce advisor-opinion scenario, these categories can be understood in a more general sense. Sellers can be generalized as agents providing content, be it data, service, product or sensor reading, etc. They receive a rating in turn. Vice versa, buyers are receiving the content and providing the rating. Advisor refers to any third party contributing to the final aggregated reputation of a potential transaction partner. Opinion is any rating, review or feedback in a more broader sense.

2.2.2 Statistical Filtering Methods

Many reputation systems are based on statistical methods, and statistical analysis can be used to further improve reputation systems. Whitby et al. proposes a filtering mechanism for unfair opinions for the Beta Reputation System (BRS)[13]. This method extends BRS, which is a Bayesian reputation system based on the beta distribution. The authors argue that unfair and fair ratings have different statistical patterns, mainly seen as differences to majority opinion. Consequently, it is possible to identify and filter unfair ratings with statistical methods. The basis of filtering like this is the assumption that the majority of the users are honest and accurate.

TRAVOS is also a Bayesian trust and reputation method based on the beta probability distribution[15]. It uses both experience-based local trust and global reputation, depending on whether past experience with a particular transaction partner is available or not.

If a potential transaction partner is not already well known, the system relies on global reputation reported by third party advisors. Deception from advisors is countered by evaluating the perceived accuracy of their past opinions. Specifically, first a probability is calculated for how likely it is the third party advisor gives an accurate opinion. This is done on the basis of previous opinions given, and the eventual outcomes of those transactions. Then as a second step, the opinions likely to be inaccurate are altered so as to have a smaller impact on the final calculated reputation. This is done through decreasing the parameters of the distribution which represents the advisor's opinion. An opinion modified in this manner will influence the expected value of the final reputation distribution to a lesser degree.

The Beta Reputation System and TRAVOS are very similar. Both are Bayesian statistical methods using the beta probability distribution. However, BRS does not differentiate between direct observations and advisor recommendations. Additionally, the two methods handle inaccurate (unfair or malicious) ratings differently. TRAVOS is an exogenous approach, scrutinizing each individual advisor based on the perceived accuracy of their past opinions. BRS is an endogenous approach, taking a single rating and judging it based on how far it is from the mainstream opinion.

The Dirichlet Reputation System is worth mentioning as a multinomial generalization of the Beta Reputation System[16]. Using the Dirichlet distribution instead of the beta distribution allows more than two discrete rating levels. It is a method well grounded in Bayesian statistics, and is built around concepts similar to that of the BRS. Specifically, aging of ratings is also present. A similar extension for filtering non-majority outlier opinions like in the BRS was not found for the Dirichlet Reputation System.

The Weighted Majority Algorithm introduced in [17] gives advisor agents a weight to aggregate their opinions with. If an opinion turns out to be incorrect based on the outcome of the involved transaction, the advisor's weight is decreased. Constantly wrong advisors are thus filtered out eventually.

2.2.3 Methods Based on Similarity

The authors of [18] list four capabilities of ideal unfair rating handling methods. These are

- handle unfairly high ratings
- handle unfairly low ratings
- deal with behavior changes of agents
- handle that agents may have genuinely different opinions on service providers

The Personalized Approach is introduced to fulfill the last criteria[18]. Apart from each agent having a public global reputation, every one of them stores private opinions of agents they have previously interacted with. When an agent looks for opinions on a

potential partner, advisor agents provide their opinion. To judge the reliability of these opinions, the asking agent compares its private opinion list with that of the advisor. If they have formed many similar opinions on the same third party agents, their preference is assumed to be similar and the opinion from that advisor is assumed more relevant.
The work in [19] proposes a dissimilarity function for detecting deviations in a recommendation set. The method assumes that dishonest recommendations are inconsistent with other recommendations and they also have a low occurrence rate. A similar dissimilarity function is also introduced for mobile ad-hoc networks in [20].
The authors of [21] introduce two trust or reputation system extensions to combat unfair ratings in e-commerce settings:

- controlled anonymity to avoid unfairly low ratings and negative discrimination
- cluster filtering to reduce the effect of unfairly high ratings and positive discrimination

Cluster filtering uses collaborative filtering techniques to identify the nearest neighbors of agents, based on similarities in commonly rated other agents. The goal is to identify conspirational collectives who give unfairly positive ratings to increase a specific provider's reputation.
The method proposed in [22] looks for similar agents and consider these as trustworthy sources of information for each other. Agents compare their trust values with trust values of similar agents.

2.2.4 Other Methods

The authors of [23] randomly select samples of all reviews to calculate global reputation in an e-commerce context.
A complex five-step process is adopted in [24] to make the reputation system more secure against unfair ratings. The steps are

- Evaluation-based filtering
- Time-domain unfair rating detection
- Suspicious user correlation analysis
- Trust analysis based on Dempster-Shafer theory
- Malicious user identification and reputation recovery

The feedback consistency method is described in [25]. Both of the transaction participants leave ratings on the transaction. The ratings are considered consistent if they evaluate the transaction with the same outcome, whether successful or not. An inconsistency is assumed to mean one of the transaction partners is not honest. When an agent reaches a certain threshold with its ratio of inconsistent vs. consistent feedback, the agent is suspected of leaving false feedback, and feedback from such agents are discounted.

The Context-Aware Approach aims to improve unfair rating detection by analyzing selected factors of the environment the detection algorithm is deployed in, and selecting the most suitable detection approach from a set of methods[26].

Some approaches provide incentives for honest behavior to prevent or mitigate unfair ratings[14, 27].

The method proposed in [28] recovers the temporary damage of sellers who were given a lower rating by a buyer by mistake, and the buyer later willingly corrected this. The seller's reputation is temporarily inflated to revert reputation damage.

First hand trust observations and global reputation are combined in some works[29, 30]. The method from [29] is developed for the mobile ad-hoc network domain. Every node maintains a first-hand reputation (trust) and a second-hand (global) reputation of others. First-hand reputation is exchanged by nodes occasionally, and data is combined to reach a better accuracy. The system employs aging and occasional reputation re-evaluation to enable reputation redemption.

Interdisciplinary approaches have also been introduced to improve reputation. Machine learning can monitor statistical properties or agent behavior to filter the reputation system[31, 32, 33]. Combining subjective recommendation systems with reputation systems to achieve better final advice is proposed in [34]. Rating scale normalization comes from collaborative filtering systems[35, 36]. Each agent's typical rating scale is determined and the different scales are normalized into a common scale.

2.3 Evaluation and Comparison

Literature has seen a number of publications that evaluate reputation methods or create freestanding evaluation frameworks for this purpose. Table 2.1 lists these attempts, along with their domains. This section describes the most notable evaluation and comparison approaches.

The ART testbed is one of the earliest comprehensive evaluation frameworks published for comparing trust and reputation systems[39]. It simulates a multi-agent system (MAS) environment of independent agents within an artwork marketplace. Multi-agent systems refer to any kind of collaboration environments, where intelligent independent agents operate in order to reach their goals. Agents in MAS environments typically have to collaborate with each other in order to reach their goals. An agent's goal in the ART testbed is to collect the most utility by the end of the simulation, which serves as the basis of comparison between different agent approaches. Thus, results from ART reflect each agent's capability in navigating the marketplace scenario, rather than the overall effect of the reputation system. ART also does not directly support testing to what degree reputation methods can discourage or remove malicious intent.

TREET is a follow-up to the ART testbed[54]. Its goal is to focus more on evaluation of trust and reputation systems and their resistance to security attacks within an e-

Name	**Domain**
Agate&etal18[37, 38]	distributed systems
ART[39]	multi-agent systems (MAS) (art appraisal)
ATB[40]	MAS
C&E15[41]	generic
DART[42]	MAS
EstarMom[43]	MAS
Euphemus[44]	online market
Irissappane&etal12[45]	marketplace
Janiszewski17[46]	generic, theoretical
Liang&Shi08[47]	p2p
Macau[48]	MAS
NEVER[49]	open networks
QTM-P2P-Sim[50]	p2p
Schlosser&etal04[51]	MAS
SIFT[52]	decentralized open networks
TO-Sim[53]	p2p
TREET[54]	MAS (marketplace)
TRM-EAT[55]	generic
TRMSim-WSN[56]	Wireless Sensor Networks
Youssef&etal17[57]	MAS

Table 2.1: Reputation evaluation frameworks in the literature, and the domains they focus on. Multi-agent systems can encompass a broad range of environments, they can be used at least partially generally.

commerce setting. TREET can simulate both centralized and decentralized reputation systems, supports collusion attacks and extension with custom attack methods. The testbed's drawback lies in its commitment to the e-commerce domain. Also, TREET's metrics evaluate based on the success agents see as a result of their decision making, and not the quality of reputation or trust values.

The Alpha Testbed (ATB) focuses on the agent's decision making mechanisms[40]. The authors argue that the way agents decide who they transact with influences evaluations of reputation systems. ATB supports changing the decision mechanisms of agents independently of the reputation or trust model, thus separating the two. Domain independence is another advantage of this framework. Custom metrics are possible to implement, although researchers have pointed out that system-level metrics that evaluate system-wide performance are not supported[44]. ATB's authors mention its inability to accommodate trust models that represent information substantially different from formats used in ATB, or when opinions are computed or shared using special protocols. Additionally, ATB only simulates one single agent in full, called the alpha agent. All other agents are a represented by mocks and do not actually use the simulated trust system. This is a clear disadvantage when a trust or reputation system revolves around social interactions or virtual organization among agents.

The QTM or P2P-Sim simulator focuses on trust and reputation systems deployed in p2p file sharing networks. QTM is a flexible system that supports custom reputation methods and attack scenarios. A trace file serves as input, which lists in advance which agents request which files, determining the simulation's transactions. Evaluation is based on a clearly defined metric called hit rate, calculated as

$$\text{hit rate} = \frac{\#\text{ valid files received by good users}}{\#\text{ transactions attempted by good users}} \tag{2.1}$$

Implementing custom metrics is not supported by default. As the QTM authors say, shared files can either be valid or invalid, and binary feedback is expected after each file sharing transaction. Supporting only binary transaction outcomes and feedback values is QTM's major limitation. Tested trust or reputation models also need to contain a selection procedure for peer agents to decide who they download their requested files from. This is not always part of models.

TRMSim-WSN is a simulator for trust and reputation systems specifically tailored for wireless sensor networks[56]. Highly domain-specific features like routing, energy consumption, radio range, battery and source/sink nodes limit this simulator's cross-domain applicability.

The works in [37, 38] (Agate&etal18) introduce an evaluation system for distributed reputation algorithms. This relatively recent (2018-2020) framework aims to support reputation management system design by testing robustness against security attacks. The framework stays domain-independent by disregarding communication protocols between agents and the reputation system as well, while enabling custom reputation calculation methods and agent behavior. Agents are also allowed to implement their own decision making for selecting transaction partners.

The framework Youssef&etal17 presents a MAS scenario in which students with different skills must complete their homework[57]. The work Janiszewski17 introduces a set of theoretical metrics that can be used to evaluate different reputation systems[46].

3

Approach

This chapter describes how reputation environments are modeled to allow comparison of improvement techniques. First, the model elements are discussed. Then, configurable parts like reputation and improvement methods, and behaviors of participating agents are covered. Finally, concepts of the evaluation framework used for simulations are introduced.

Table 3.1 provides a reference for notations introduced throughout this chapter.

3.1 Transactions and Content

In the following of the thesis we assume that a reputation system consists of a set of *agents* $A = \{a_1, a_2, ...a_n\}$, which engage in transactions with each other over time. Transactions can be data transfer, a purchase, service usage, etc. One of the transaction partners fills the role of a *provider*, while the other is a *consumer*.

Transactions involve the delivery or exchange of something, like products, service provision or any kind of data. Concrete examples are a book, a hotel room, files shared on a

Notation	Meaning
A	all agents of a scenario
a_m	the m^{th} agent, $a_m \in A$
$C^r_{i,m}$	the claim with index i, made by agent a_m in round r
$Q_{meas,m,i}$	quality of claim C_i as measured (seen) by agent a_m
$Q_{true,i}$	ground truth of claim C_i
$R_{m,i}$	a review of claim C_i, the review is by agent a_m
$R_{author,m,i}$	the author review of $C_{i,m}$, made by agent a_m

Table 3.1: Reference of notations used in the introduced model. When the information carried by a subscript is not relevant in the mentioned context, that is omitted from the notation for clarity.

peer-to-peer network, an online comment, a contractual agreement. It can also be sensor data or an algorithmic result like a self-driving car's estimated time of arrival to the next intersection.
We refer to these objects of exchange as *content* from here on. Because the model aims to stay domain-independent, we elected to

- not explicitly define the specifics of transactions
- not model the content
- not model content-exchange processes or protocols

3.2 Claims

Before agents commit to a transaction, they have a preliminary understanding of what content exchange would take place. Claims [1] model the promoses of content providers. Claims are denoted as $C_{i,m}$, where i is the claim's identifier index, and agent a_m is the claim's author. Conceptually, a claim is made up by the following components.

1. representation of the provided content's quality, named ground truth or claim quality
2. the author agent that made the claim
3. an author review
4. the time from which the claim is available (published)

Claims do not explicitly contain anything about the content they represent, because as discussed previously, content is not modeled in order to stay domain-independent. Content is only implied with the claim, and only its quality is represented. More broadly, claims do not contain anything about what the context of the reputation scheme is. The following is all assumed:

1. some kind of content behind the claim
2. a way of offering or distributing said content
3. an environment in which the content is produced and consumed

Claiming (Figure 3, above) is the process of making (publishing) new claims. It start with a new claim $C_{i,m}$ without any reviews, containing a ground truth. The claimer agent a_m measures this hidden quality and distorts it, resulting in a value representing the claimer's opinion of his own claim. This is attached as an author review $R_{author,m,i}$ to the claim to be published. Subsequent sections in this chapter discuss the contepts of ground truth, distortion and reviews.

[1] The claim name comes from its old use. Formerly, the author review was called claim value. It was said the agent "claims that the quality of its content is *xyz*"

Context	**Example Content**	**Example Meaning of Valuation**
File sharing	Legally distributed music	File is what was promised
Marketplace	Books	Enjoyment
Sensor data	Estimated time of arrival	Accuracy of estimate
Online discussion	Social media post	Popularity, agreed or not

Table 3.2: Example contents and their possible quality valuations within various contexts. Some valuations are more subjective, others can be accurately judged.

3.2.1 Quality Valuation

Quality valuations or estimations model a quantitative valuation of the quality or goodness of the content. What this means can differ between domains. Table 3.2 lists examples for contents and their possible valuations. For example, in an online marketplace of entertainment products, the valuation can be some approximation of the entertainment value the content provides. In the case of a file-sharing context, it could indicate whether the shared files that form the content are intact, and what is provided is also what was promised.

Three kinds of valuation are modeled, distinguished by the source of the valuation. These are:

- ground truth (or claim quality)
- author's review
- regular review

3.2.2 Ground Truth

The ground truth, or alternatively called claim quality, denoted by $Q_{true,i}$ is meant to model the objectively true but hidden and not knowable quality of a content behind the claim $C_{i,m}$. This assumes that there is a way to calculate or judge a content's universal quality with the chosen valuation. In the case of the autonomous vehicle's estimated time of arrival, the ground truth represents the actual physically measurable length of time the vehicle needed for arrival. Of course, accurate and objective judgment is often not possible. For example, the enjoyment of books, or the validity of a social media post is hardly universally estimable.

Still, ground truth is a useful tool for modeling reputation schemes. A baseline truth is often needed for evaluating the performance of the reputation scheme, or in the case of this work, improvement techniques. Additionally, such a claim quality can be used as a starting point when modeling how other agents see the content. This method is called measuring claims and is discussed in Section 3.2.4.

Ground truth is an inherent quality in every claim. When an agent claims, the new claim will contain a ground truth value that is not controlled or influenced by the claimer at all. In this model, a ground truth is seen as a kind of circumstance, one that would only show as time progresses. For example, when the autonomous vehicle makes an estimate for a time of arrival, the real and accurate time taken to arrive is not yet known. For modeling purposes it is assumed that it will be known, and it was in fact generated as ground truth at the time the claim was made.

Finally, ground truths are hidden from agents, and they have no way of directly accessing this valuation. The closest estimation of the ground truth value that agents can obtain is through the above-mentioned measuring process, which is a model of trying out the content and forming an opinion of it.

3.2.3 Author's Review

An author review $R_{author,m,i}$ is a valuation modeling what the agent a_m who is author of claim $C_{i,m}$ says the quality of the content is. It is a self-appraisal, a declaration of what the claimer thinks its claim is worth. Modeling the author's opinion is a somewhat unusual approach, since this is usually not explicitly part of reputation systems. The benefit of having author reviews is that reputation calculation methods or agent decision making can use it. When this is not needed, other parts of the model would simply disregard author's reviews.

Author reviews can be interpreted in multiple ways. When the reputation scheme uses an explicit value which agents publish along their claims, it is exactly that, an explicit self-assessment of provided content. Otherwise, it can imply an unspoken, but still advertised positioning of the claim. This can be a marketing message for example, or an expectation that a specific director's movies are usually good. These give no explicit valuations tied to contents, but some kind of positioning can still be implied from the image, history, expectations, etc.

3.2.4 Measuring Claims

Measuring claims is the term used for when an agent assesses the quality of a claim. This models when a consumer upon receiving a content, tries it out and leaves feedback based on first-hand experience. Additionally, an agent can also measure its own claim.

The measured quality $Q_{meas,m,i}$ denotes agent a_m's assessment of claim $C_{i,n}$. With the model's assumption on the existence of a ground truth, measuring a claim can be seen as an agent trying to guess this hidden quality valuation. Agents are modeled here with various levels of expertise, which affects how close or how reliably they can measure the ground truth. Know-how, experience, expertise or inside knowledge are among the possible factors playing a role. The model's implementation uses Claim Truth Assessment Inaccuracy for this, which is a parameter of agents representing the maximum inaccuracy they can make when measuring.

As discussed, making a new claim has two main stages: measuring the ground truth and distorting. The measurement serves as input for distortion. Distortion models conscious actions, honesty or willing manipulation. Measuring claims is also a sort of distortion, albeit not under the control of the agent. While distortion is *intent*, measurement represents *competence*.

3.3 Regular Reviews

Regular reviews (or simply reviews) model the feedback on products from agents that are on the consumer end of transactions. A review $R_{m,i}$ is the opinion of agent a_m on claim $C_{i,n}$, where $m \neq n$. Every review implies that a transaction took place, and content was exchanged. The agent a_m who leaves the review is in the consumer role, and is called *rater*.

A fundamental assumption of the model is that raters leave feedback on the content, not the content's provider. This means that the subject of reviews are claims, not the claimers. A rater leaving a review on a claim represents what that rater thinks the content quality is. Another interpretation is that a review is what the rater thinks the rated claim's author review should have been. In essence, what the rater would publish as author's review, were that rater the author of that claim. Both interpretations have the same outcome.

3.4 Reputation Calculation

Two reputation calculation methods were designed for evaluation, both of them based on relatively simple averaging. One drawback of using custom methods is they are not directly comparable with any other related reputation system. Another issue is both methods assign reputation to claiming agents only. Consumers who only rate and never claim, do not get a reputation.

The two methods compensate with the benefits of simplicity. Unlike with more complex methods, it is possible to easily follow what happens. Averaging is also a very common-sense approach. Many people would expect something like this when seeing a score-based reputation system, although this is rarely the case. Finally, both methods can support weights, one of the improvement techniques used.

3.4.1 Difference of Author Reviews and Regular Reviews

This method makes use of the unique author's review system. It exploits that both the author and consumers make reviews of claims. Assume an agent a_m has made a series of claims $\delta = \{C_{i,m}, C_{i+1,m}, ...C_{i+N-1,m}\}$, that is N number of claims in total. To get the reputation of agent a_m,, each of its published claims δ has its reviews averaged, and the difference between this average and the author's review is calculated (3.2). Once ready for all of an agent's claims, the differences themselves are averaged as in (3.3).

$$\begin{aligned} \text{difference}(C_{i,m}) &= \left| \frac{\sum_{n \in A} R_{n,i}}{\text{number of reviews } R_{n,i}} - R_{author,m,i} \right| \\ \text{where } n &\neq m \end{aligned} \tag{3.2}$$

$$\text{average difference}(a_m) = \frac{\sum_{i \in \delta} \text{difference}(C_{i,m})}{\text{number of claims in } \delta} \tag{3.3}$$

The maximum possible difference is a property of the reputation scheme, known as the difference between the lowest and highest possible review valuation.

Finally, the agent's reputation is calculated based on the percentual relationship of the average of differences and the theoretical maximum difference (3.4).

$$\begin{aligned} \tau &= \frac{(\text{max difference}) - \text{average difference}(a_m)}{\text{max difference}} \\ \text{reputation}(a_m) &= \tau * (\text{reputation range}) + \text{min reputation} \\ \text{where max difference} &= (\text{max reputation}) - (\text{min reputation}) \\ \text{reputation range} &= (\text{max reputation}) - (\text{min reputation}) \end{aligned} \tag{3.4}$$

If an agent's author reviews of its own claims would perfectly align with the reviews they received, the agent's reputation becomes the maximum possible. Similarly, if the difference is at the highest possible, e.g. when an agent's author reviews are all at the maximum rating and all reviews are at minimum, the reputation becomes the lowest possible. In between it is straight proportional.

Basically honesty of claimers is rewarded, while dishonesty is punished through reputation. Reputation are given based on how close agents rate their own claims compared to the average opinion. Another way to explain it, is that regular raters approximate the claim's ground truth. Here, an agent's reputation comes from how close that agent valued its own claims compared to the claims ground truths as approximated by reviewers. In the end, an agent's reputation informs others how reliable that agent's author reviews are.

3.4.2 Simple Review Average

Here, reputation is the average of received reviews. The method simply collects every review on all claims an agent has. The average of these received reviews becomes the agent's reputation. Collecting and averaging is repeated separately for all agents. As such, the raputation of agent a_m is given as

$$\begin{aligned} \text{reputation}(a_m) &= \frac{\sum_{i \in \omega} R_{n,i}}{\text{number of reviews } R_{n,i}} \\ \text{where } n &\neq m \\ \omega &= \text{all claims by agent } a_m \end{aligned} \tag{3.5}$$

The specialty in simple averaging of reviews is in the different interpretation of reviews. Section 3.3 discussed previously how reviews are meant to give feedback on claims, not on the claimer agents. Simple review averaging turns this around and makes reviews behave as rating the claimers themselves.

This is best seen through an example. Let the rating range be [1,9], i.e. for all reviews $\forall R : 1 \leq R \leq 9$ and all quality variations $\forall Q : 1 \leq R \leq 9$. An honest claimer a_m publishes a claim $C_{i,m}$, which has a low ground truth of $Q_{true,i} = 2.5$. Accordingly, let the claimer's author review of the claim is also low, because the claimer is an honest agent. Let the author review be $R_{author,m,i} = 2$. If some other agents a_n, a_o, a_p then rate this claim, and they rate based on the quality of the content represented by it, they would leave mostly low reviews. Their reviews could be

$$\begin{aligned} R_{n,i} &= 2 \\ R_{o,i} &= 1 \\ R_{p,i} &= 3 \end{aligned} \tag{3.6}$$

Assuming a_m has no other claims and no other agents have given ratings apart from the three above, the reputation of the claimer a_m becomes

$$\text{reputation}(a_m) = \frac{R_{n,i} + R_{o,i} + R_{p,i}}{\text{number of reviews}} = \frac{2+1+3}{3} = 2 \tag{3.7}$$

Even though the claimer was honest in declaring its own content as of low quality, the resulting reputation is still low. Here, the reputation signals that the claimer provides content that is low in quality.

The other case is when raters review the claimer itself, how good their transaction experience was, whether it went according to expectation, etc. So raters consuming a content behind the above claim fully expect it to be of low quality, because the author's review also warned them. Assuming the transaction goes well, raters leave a good (high) review. This results in a higher reputation for the claimer. In this case, reputation signals if claimers are honest.

As seen, Simple Review Averaging changes what reviews and reputation mean away from their default intended meaning. For this reason, the use of Simple Review Averaging was mostly neglected in favor of the other reputation calculation method described in Section 3.4.1.

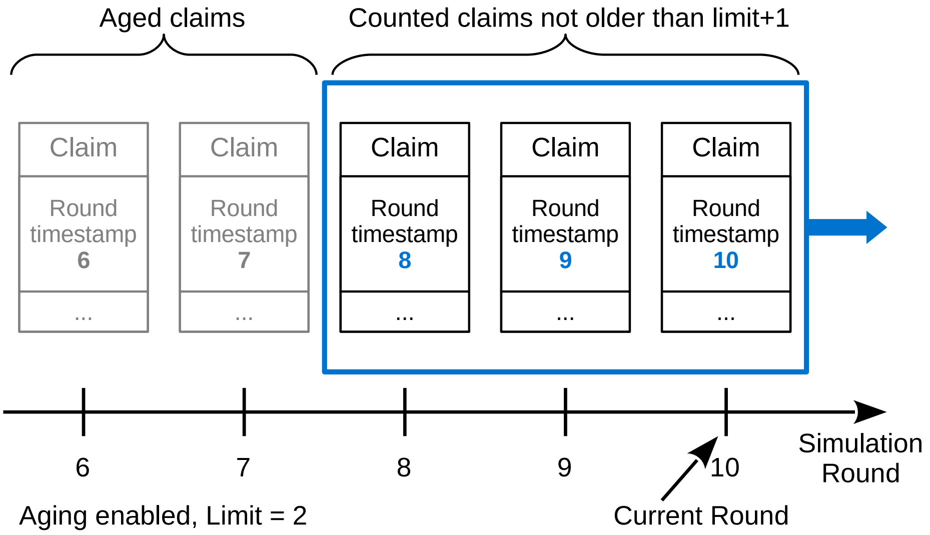


Figure 1: Schematic showing the adaptation of aging as improvement technique. Aging maintains a window of time in which recent claims fall. These remain active and counted. Every claim older than the limit is removed.

3.5 Improving Reputation

3.5.1 Aging

Aging's purpose is to limit the memory of the reputation system, so as to help follow changes in agent behavior or sentiment. Older data is forgotten, so no agent is locked into a reputation they no longer deserve. The *Limit* parameter specifies how long data is retained. Generally, *Limit* can be any unit of time measurement. The proposed implementation from Chapter 4 divides time into rounds, and the aging limit gives a number of rounds, so it is assumed to be a natural number $\mathbb{N}$.

The schematic function of aging is shown in Figure 1. If the current round r_{now} indicates the present time, then for any claim $C^r_{m,i}$ made in round r, aging is:

$$\begin{cases} \text{remove } C_{m,i} & r_{\text{now}} - r > \text{Limit} \\ \text{still count } C_{m,i} & \text{otherwise} \end{cases} \tag{3.8}$$

Removing a claim also removes all attached reviews on that claim.

3.5.2 Weights

Weights aim to give preference to some agents over others when aggregating reviews. The purpose of this is to give more weights to opinions of agents who have managed to build a good reputation, and less to those with bad reputation. Intuitively, this acts in

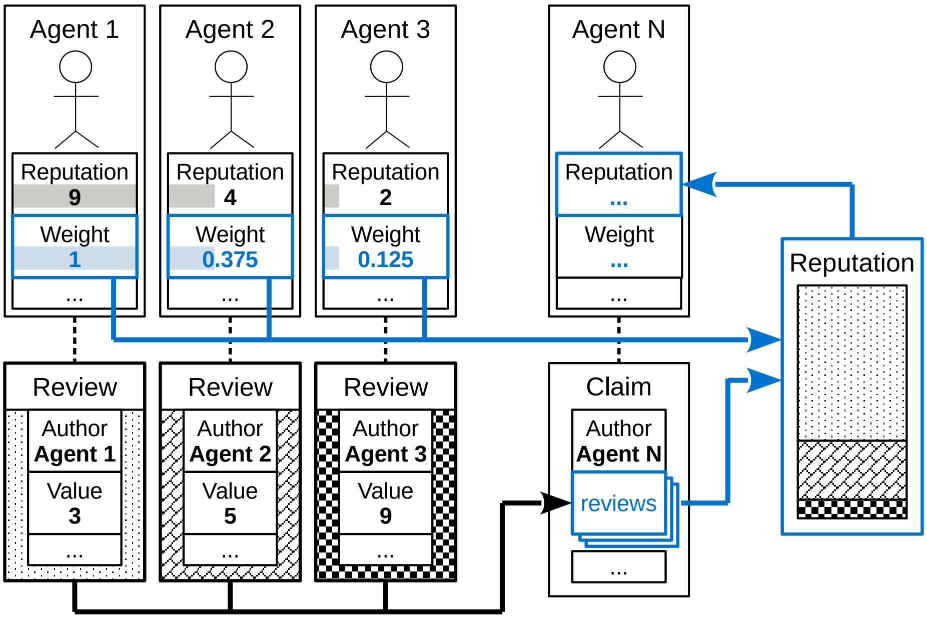


Figure 2: Schematic showing the adaptation of weights as improvement technique. Whichever method is used to calculate *Agent N*'s reputation, it also takes the weights of the raters into account. Their reviews are represented according to their weights. How agents get their weights can differ. Here, weights simply reflect the agent's reputation.

the opposite direction than aging, because weights preserve the status quo, while aging helps to change it.

Figure 2 shows how weights are modeled. An agent's reputation serves as its weights, calculated proportionally. Reputation calculation methods use weighted average for aggregating reviews.

3.5.3 Stakes

We propose stakes as a new improvement method, consisting of claimers publishing a stake along their new claims. A stake gives how certain claimers are in their valuation of their new claim, i.e. the author's review. A high stake signals confidence of the agent in the author review's accuracy, while a low stake notifies others that even the claimer is not sure whether its author review is correct.

Stake is a similar to a self reported weight, with a scope of only the involved claim. Other agents and even reputation calculation can use the stake when making decisions. For example, a low stake could mean the claimer does not receive much of a reputation penalty if the author review turns out to be inaccurate, while a higher stake incurs an amplified reputation correction.

Since stake requires providing an extra valuation with every claim, it is best suited for algorithmic systems. A sensor can provide stakes based on its known uncertainty for example.

3.5.4 Other Selected Methods

Outlier filtering can be modeled by removing reviews which have significantly different valuations than the majority opinion. This can be a way to prevent a small number of saboteurs to influence a claim's overall review valuation. If the majority opinion of the claim shifts, the removed reviews need to be reconsidered and re-added if they are no longer outliers.

Anonymous ratings serve to isolate the content from the content provider to prevent biased rating. This is effectively implemented within the model if none of the agents use a strategy that discriminated against specific agents.

Rating range normalization aims to even out different rating habits of agents. For example, some agents avoid giving minimum and maximum ratings and stay at the middle ranges, others rate strong opinions more willingly. Normalization extends the midrange rater's ratings to span the whole rating range. This method is only applicable if agents already have a lot of ratings, because enough data is needed to confidently determine the typical rating range.

3.6 Distortion

Distortion models the conscious and willing efforts of claimers to influence other's perception of their content quality. This is one vector for agents to attack and exploit the reputation system, namely from the provider's side. With this concept, the model allows the setup of more elaborate attack scenarios.

Distortion fits in the model as part of making new claims, seen in Figure 3. When an agent a_m makes a new claim $C_{i,m}$, that agent has some valuation of what the claim's quality is. This is called the measured quality $Q_{meas,m,i}$ as per Section 3.2.4. The claimer agent A needs to make an author's review $R_{author,m,i}$ about its new claim $C_{i,m}$. Distortion basically defines how the agent comes up with the author review from the known the measured quality of the claim.

$$\text{distort}_{a_m}(Q_{meas,m,i}) = R_{author,m,i} \tag{3.9}$$

The most honest approach is to do no distortion at all. Here, the author's review contains exactly the claimer's valuation of the claim quality. Less honest agents can either state a higher or lower value than their own estimate. To what extent and how often they do it depends on the distortion method they use.

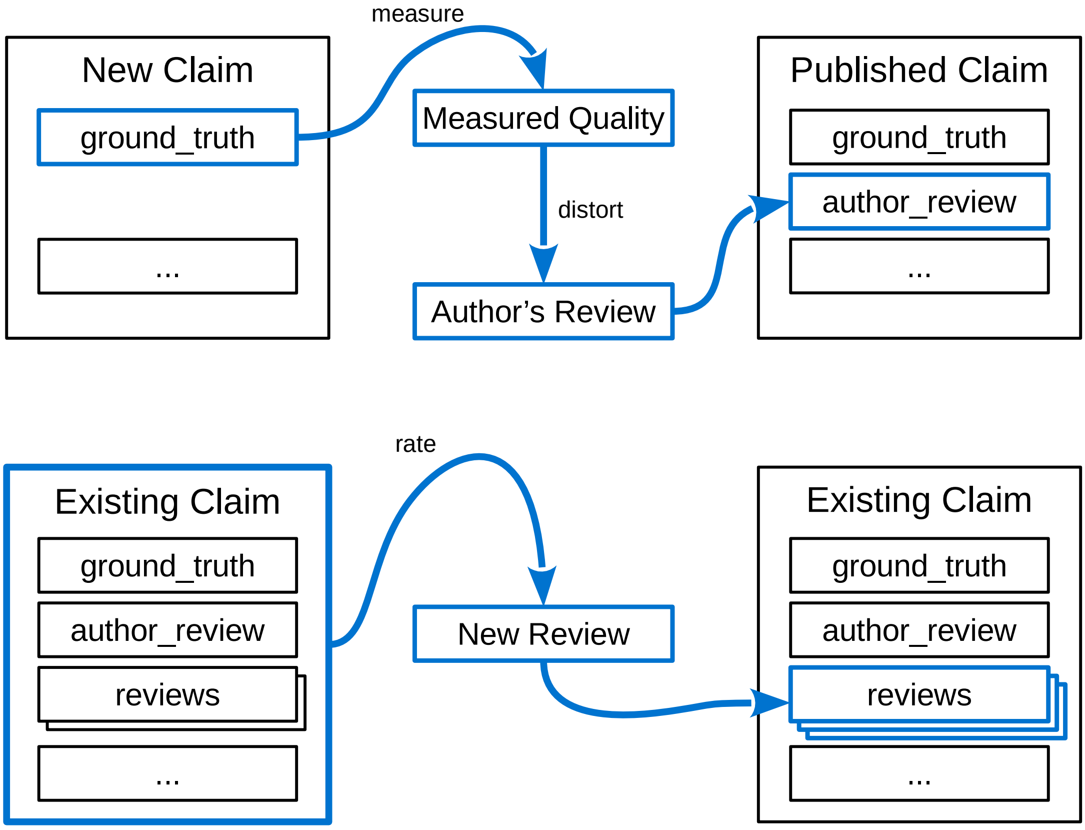


Figure 3: Schematic representation of claiming (above) and rating (below) processes, with the involved claims and reviews.

3.7 Rating

Rating models the feedback agents leave on content they have used. When an Agent a_m rates a claim $C_{i,n}$ made by another agent a_n, a regular review $R_{m,i}$ is produced.

$$\mathrm{rate}_{\mathrm{a_m}}(C_{i,n}) = R_{m,i} \tag{3.10}$$

Rating is analogous to distortion in that both end up producing a kind of review. The difference is that rating produces regular reviews on published claims, while distortion is used for author reviews.

Apart from this, rating may use a much larger amount of input data, everything that is part of or connected to the rated claim $C_{i,n}$. This includes not only the measured quality, but also the other reviews of the $C_{i,n}$, data of the claim author agent a_n like a_n's reputation, other claims of a_n and their review, and so on. This allows modeling elaborate methods of feedback behavior.

Such amount of data is often not reasonable to utilize at once. Therefore, models of rating usually select one valuation related to the claim, and then apply some manipulation

to this value. This input represents some numeric quality of the claim, like the author review, the quality as measured by the rater agent a_m, the average of other reviews, etc. Comparing how well these perform can be a task of its own. No specific value is assumed while discussing methods below. The general notation X is used instead.
Rating is the attack vector of those on the consumer side of transactions. Many attacks and exploits found in related works can be modeled as rating methods, because reviews are usually the most important factor in calculating reputation.

3.7.1 Simple Rate Methods

Simple rating methods are characterized by using only one or a few input values and they can be described as a static transfer function. The modeled ones are shown in Figure 4. Every transfer function can have parameters and an input X.
The two commonly considered attack methods are *promotion* and *slandering*[58]. They are modeled with linear or breakpointed linear methods, as in equations 3.11 and 3.12. Linear breakpointed can work both ways, setting either a constant low or a high limit for ratings before or after the breakpoint, respectively. Parameters: a, b, *direction*, *breakpoint* and *breakpoint value*.

$$\text{rate}_{\text{linear}}(C_{i,n}) = aX + b \tag{3.11}$$

$$\text{rate}_{\text{breakpointed}}(C_{i,n}) = \begin{cases} aX + b & \text{if } X \text{ within } \textit{breakpoint} \\ \textit{breakpoint value} & \text{otherwise} \end{cases} \tag{3.12}$$

Second order polynomial ratings described by (3.13) are included from [17], where they are called exaggerated positive and negative. Parameters: a, b, c and *direction*, the latter influencing if the polynomial is added or subtracted.

$$\text{rate}_{\text{secondorder}}(C_{i,n}) = X \pm (aX^2 + bX + c) \tag{3.13}$$

Invert rating is also adapted from [17].

$$\text{rate}_{\text{invert}}(C_{i,n}) = -X + \max(X) + \min(X) \tag{3.14}$$

Flattening was introduced to model the often-seen behavior of people avoiding strong opinions, tending toward the safe middle values. This is based on a well-established observation in psychology and sociology, and is called the central tendency bias[59, 60]. Flattening reduces the more extreme values towards the middle range. The strength of flattening is given by parameter $a < 1$.

$$\begin{gathered} \text{rate}_{\text{flatten}}(C_{i,n}) = a(X - \textit{mid}) + \textit{mid} \\ \text{where } \textit{mid} = 0.5\big(\max(X) - \min(X)\big) + \min(X) \end{gathered} \tag{3.15}$$

Flattening can be seen as conservative or reserved rating. The opposite of flattening is

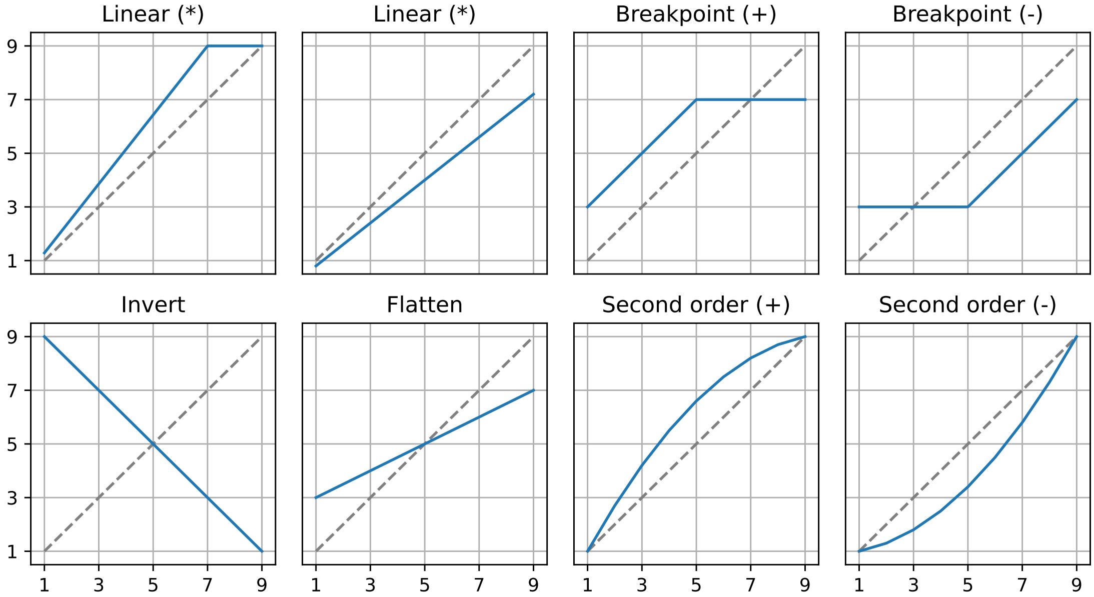


Figure 4: Transfer functions of simple rating methods. The dashed line is the input without transformation. It can be any single value from the rated claim, for example its author review, or measured quality. Rating methods show example parametrizations, these can of course change. Both the input and ratings are selected to be within the integer interval $[1, 9]$.

extreme rating, when an agent tends to give more extreme ratings, around the edges of the rating range. This method uses the same transfer function as in (3.15), but with a multiplier $a > 1$ that amplifies toward extremes. Extreme rating can model agents that mostly have strong opinions, either very high or very low, a behavior that has also been observed in online rating systems[61].

Finally, the simplest of all rating methods is when nothing is done. This means some quality X of the claim is returned without altering. An example is the method that simply repeated the author's review of the rated claim.

3.7.2 Random Rate Methods

Random error rating is a method where sometimes a significantly inaccurate rating is made. It can model a sensor's rare erroneous data. Also, agents whose attack approach is rate normal most of the time to build reputation and occasionally insert a malicious rating. The malicious rating may carry more weight due to the rater's otherwise good reputation. The method is modeled as per (3.16). Parameters: *chance*.

$$\text{rate}_{\text{random error}}(C_{i,n}) = \begin{cases} random & \text{if } chance \text{ is taken by coincidence} \\ X & \text{otherwise} \end{cases} \tag{3.16}$$

Various other random ratings can be used to model actors who want to weak havoc or are completely ignorant. Random, lower half, and higher half random all give a random value within the whole or part of the rating range.

$$\text{rate}_{\text{random}}(C_{i,n}) = \text{random value in } \big[\min(X), \max(X)\big] \tag{3.17}$$

$$\text{rate}_{\text{random higher half}}(C_{i,n}) = \text{random value in } \left[\frac{\min(X)}{2}, \max(X)\right] \tag{3.18}$$

$$\text{rate}_{\text{random lower half}}(C_{i,n}) = \text{random value in } \left[\min(X), \frac{\max(X)}{2}\right] \tag{3.19}$$

3.7.3 Rate Methods of Collectives

Collaborating agents working towards a common goal can form collectives. Such groups are common in real online reputation systems. Producers often have a loyal following among consumers, who will promote the producer's content with higher ratings. Some marketing or other agencies take clients to improve their online reputation by a coordinated promotion attack. Producers might also initiate slandering-type attacks against competitors. This is sometimes done by organized or spontaneous mobs, leaving baseless bad reviews on businesses or sellers for various reasons.

A main distinction is whether the collective promotes or slanders. Slandering is typically aimed towards an agent outside the collective. Promotion on the other hand can target members of the collective. As also mentioned before, a collective can build up the reputation of its members by promoting each other, so that a later planned slander attack carrier more weight.

3.7.4 Experience-Based Rate Methods

Experience-based rating is when a consumer judges a content based on personal experience with it. An agent's own experience is modeled by measuring claims, discussed in Section 3.2.4. When an agent a_m measures the claim $C_{i,n}$ made by agent a_n, it results in the measured quality $Q_{meas,m,i}$.

Technically any of the discussed rate methods can be experience-based, if X is chosen as $Q_{meas,m,i}$. Still, explicitly defining such a rate method is worth it because rating from own opinion is common.

$$\text{rate}_{\text{own experience}}(C_{i,n}) = Q_{meas,m,i} \tag{3.20}$$

3.8 Resolution

Resolution determines what values are allowed in various parts of the model. For example, if reviews are constrained by a resolution of 1, and minimum and maximum values are

1 and 9, then the result of ratings can be integers from 1 to 9, i.e. $\forall R$: $R \in \mathbb{Z}$ and $1 \leq R \leq 9$.

There are two main uses for resolution. One of them is the already mentioned review resolution. It applies to any review made, and its purpose is to force review values to follow the wished reputation scheme. With separate resolution handling, rate and distort schemes remain independent of the exact selection of allowed values they are expected to produce. This means they do not need to be customized along this, their results will simply be changed to the nearest resolution step.

The other use is with measured claim quality values. Measured claim resolution effectively limits how finely grained agents can assess claim ground truths. For example, let review resolution be 1 and measurement resolution be 0.5. Here, reviews are strictly integers. But when agents assess a claim's quality by measurement, they may get a value between two integers too. For example, a measurement of 2.5 signifies the agent thinks the claim is better than 2 and worse than 3. The agent will still not be able to publish a review of 2.5 because the review resolution is 1. However, the information is there, what the agent does with it is left for the agent to decide.

The idea behind limiting measured claims in this way is the following. When thinking of what quality a content (claim) has, it is often not possible to exactly determine it. For example, in the 5-star scheme, one could think something is around 3 or 4 out of 5, but not give a more exact estimate. Measuring claims models this thought process. If measurement would give 3.4562 in this case, there is no uncertainty where exactly the quality falls. If resolution is at 0.5, measurement can only return either 3, 3.5 or 4, or even other values if the agent has a huge inaccuracy. A similar but slightly better claim would also measure at the same values. This is more realistic when agents represent people judging content freely. On the other hand, a more fine-grained resolution could be used when the scenario works with an algorithmic reputation environment. Where agents are machines and there is an algorithmic way to assess a claim, it is reasonable to allow more freedom for claim measurements.

A third resolution domain could also be created for reputations, limiting the granularity of global reputations. The reason this was not included is that reputation calculation methods serve as a single source of reputation values. They can adhere to specific resolution limits as part of their calculation if needed. For example, if a reputation strategy is build to rate in the 5-star system, it is implemented to give integers [1,5] anyway. On the other side, a scenario can have many different kind of behaviors for agents. Designing multiple versions of each with different resolutions unreasonable, hence the review resolution domain.

3.9 Scenarios

A scenario is the description of a complete reputation environment including the reputation schemes and agents with various behaviors. All parameters that are needed during simulation are part of the scenario. Some of the most important are:

- how reputation values are calculated
- which reputation improvement methods are applied
- the size of the agent population
- how agents behave during simulation
- how long the simulation goes (i.e. how many rounds)
- the possible rating and reputation values (e.g. binary, integers 1 to 5)
- what seed is used to generate random numbers
- what graph and data exports metrics should be created

In this way, a scenario encapsulates all that is in a single simulation. This helps organize combinations of simulation parameters, and compare them. For example, when the goal is to see what effect malicious agents have, one could draft separate scenarios with varying percentage of malicious agents ranging from 0% to 100% in each. For a complete list of parameters with details, see Appendix A.

3.10 Custom Evaluation Framework

There have been many reputation evaluation frameworks made in the past, as surveyed in Chapter 2. The first approach was to find one of these suitable for use, for which a few general requirements were applied. These are:

- available source code
- domain-independence
- preferably widely accepted
- extensibility and ease of modification
- flexibility with simulation parameters and components
- preferably some common reputation schemes already implemented

Most of these are needed to accommodate unique approaches and fulfill specific requirements, such as:

- the use of one or more reputation improvement methods
- ability to implement custom reputation schemes

- flexible review values (custom integer ranges)
- second-order rating schemes (like stakes)
- ability to represent claims in some form
- can define custom evaluation metrics
- full simulation of all agents

None of the examined evaluation frameworks satisfied all needs. Overall requirements and unique approaches warranted the inception of a new reputation simulation framework. Because of this, a custom system was designed to simulate, compare and evaluate selected reputation schemes under various environmental conditions. The evaluation framework uses the model described throughout this chapter so far. The next chapter details the concrete Python implementation of the system.

4

Implementation

Simulations are done with a custom-made python simulation framework named Pyrepsys. This chapter gives the implementation-specific details of Pyrepsys. For a conceptual description, see Chapter 3.

Pyrepsys is a one-stop solution for simulating, comparing and evaluating reputation schemes. Simulation of scenarios is performed on a round-by-round basis. Shared configuration among scenarios, simulating batches of scenarios in succession and controlled random number generation help compare different scenarios. Evaluation is aided by metrics. These are data processing objects that automatically generate graphs and export data from simulations.

Flexibility and extensibility are among the main design objectives of Pyrepsys. Variables can be altered in scenario configuration files. The size and composition of the agent population is customizable. New agent behaviors, metrics, reputation calculation and improvement methods can be easily added.

Miscellaneous supportive functionalities are also discussed in this chapter. A scenario configuration creator can be used to make a batch of scenarios with variations along selected configuration parameters. Extensive automated self-testing can verify the integrity of the simulation framework if any changes are made. Finally, profiling and benchmarking tests are provided to identify simulation bottlenecks and measure performance.

4.1 Simulation Flow

Simulation in Pyrepsys is built around scenarios. Each invocation consists of simulating one or more scenarios after each other. At the beginning, the default scenario configuration is fetched. Then Pyrepsys reads and applies the individual configuration for each scenario. The scenario is simulated after these preparations. Conducting the simulation is the responsibility of the *ScenarioSimulator* class. Once all scenarios are finished, the results

Module	**Responsibility**	**Notable Classes**
agent	Define agents with the simulation data structure	*Agent*, *Claim*, *Review*
behavior	Submodule. Define agent rate and distort strategies	
config	Scenario configuration related. Read, parse, provide configs. Prepare other classes.	*Configurator*
errors	Custom exceptions	
helper_types	Accessory type definitions	*SimulationEvent*, *ResolutionDomain*, *LocalConfig*
helpers	Accessory function definitions. Resolution handling, Internal-Agent-Exposed conversion	
instantiator	Creation of metric, reputation strategy, behavior strategy and improvement handler instances	*Instantiator*
main	Program entry point, preparation of logging and directories, coordination of simulation	
metrics	Submodule. Defines data export and graph drawer metrics	
paths	Directory paths	
reputation	Submodule. Define improvement handler and reputation strategy classes	
results_processor	Containment and coordination of metrics	*ResultsProcessor*
scenario_creator	Scenario creator utility	
scenario_simulator	Simulate one single scenario. Aka System	*ScenarioSimulator*

Table 4.1: Modules making up Pyrepsys. Modules `behavior`, `reputation` and `metrics` form submodules.

processor module exports collected data and draws graphs based on the selected metrics. The simulation part's schematic flow is shown in Figure 5.

Scenario simulation is performed on a round basis. The number of rounds is part of the scenario configuration. Each round consists of four distinct phases:

1. Claiming
2. Rating new claims
3. Applying reputation improvements
4. Reputation calculation

4.1.1 Claiming

Claiming is the part where agents can make new claims. During this phase, all agents get an opportunity to make one new claim. Whether an agent ends up publishing a claim or not depends on multiple factors.

First of these is the agent's claiming probability. This represents the likelihood of them attempting a claim if given an opportunity. A more consumer-type agent would have a lower probability and thus claim rarely if ever. On the other hand, producer-type agents would tend toward higher probabilities, and claim more often. The random check whether agents attempt a claim or not is performed on the main random chain.

If the agent passes this check, a claiming process begins. First, a claim is generated containing a random ground truth. The ground truth is not directly accessible for the agent. It can however be measured, which is the second step of the claiming process. Measurement results in the measured claim quality, which represents an approximation of the claim's true quality. How good this approximation is, depends on the agent's ability to assess claims. Further details of measuring claims is described in Section 3.2.4.

With the measured truth, the agent executes its distortion strategy. It essentially applies one of the distortion methods found in Section 3.6. The resulting value is the distorted claim quality.

At this point the agent checks whether the distorted quality falls in the agent's claim range. Claim range serves as a minimum and maximum limit as to what claims an agent is willing to publish. If the distorted claim quality falls outside these limits, the claim is discarded and claiming is aborted. In this case, the agent will not claim in this round.

If the limits are not violated, the agent prepares the claim for publishing. The author review is created and appended to the claim. Its value is always the distorted claim quality from the previous step. Finally, the agent appends the claim to the list of his claims, and so the claim is published. The claiming process is illustrated on Figure 3.

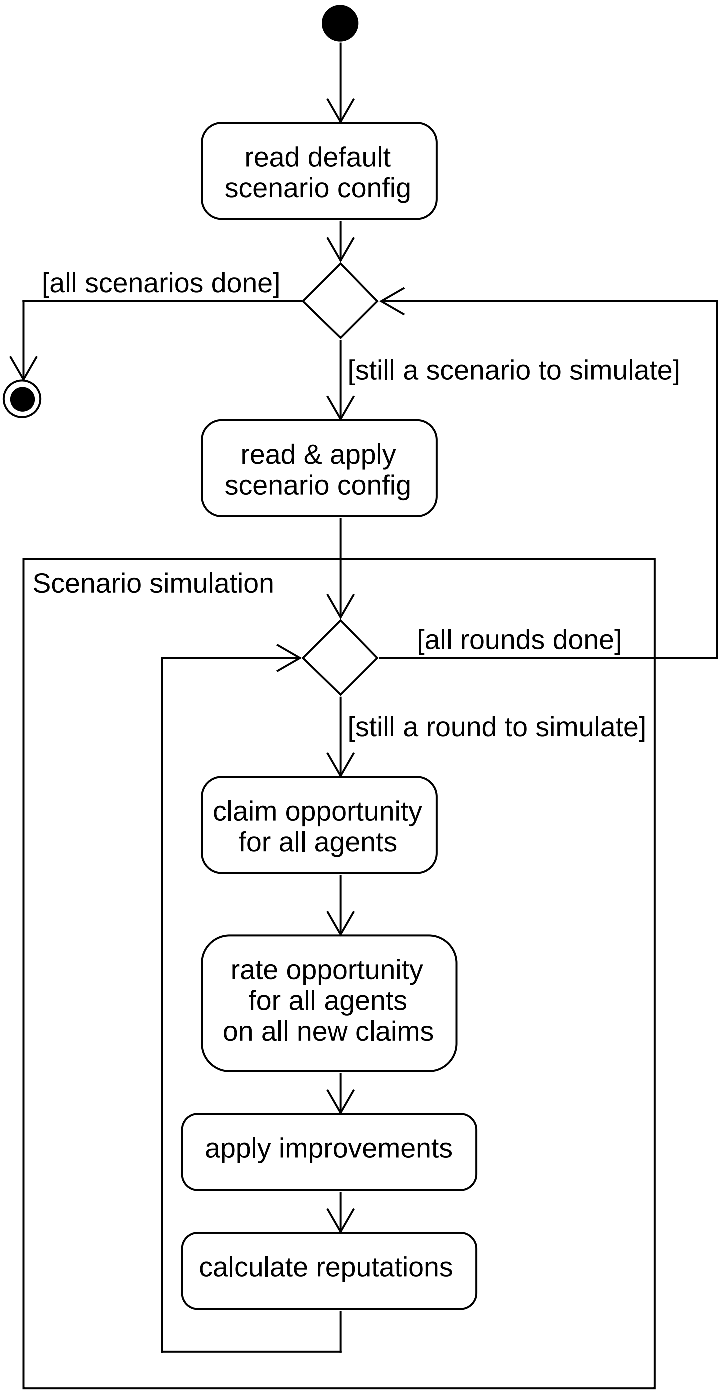


Figure 5: Simulation flow in Pyrepsys. A single simulation unit is the scenario. Each scenario is simulated independently after one another. Rounds make up the simulation within a scenario. Each round consists of four distinct phases: claiming, rating, improvement and reputation calculation. The number of rounds is specified in the configuration.

4.1.2 Rating

Rating stage is when agents can rate claims published in the current round. Each new claim made in the previous section is put up for rating. For each of these claims, every agent other than the claimer is given an opportunity to rate it. The maximum possible number of reviews made during one round is

$$(N^r_{reviews}) \leq N^r_{claims}(N_{agents} - 1) \tag{4.21}$$

Where N^r_{claims} and $N^r_{reviews}$ are the number of claims and reviews made in the current round r. N_{agents} is the number of agents in the current scenario.

Similarly to claiming, agents are given opportunities to rate a claim. Whether or not they take it and leave a review depends on the agent's rating probability. This percent chance represents each agent's willingness to rate encountered claims. A low rating probability means the agent is passive and rates seldom. A higher chance to rate represents an enthusiastic agent that often makes reviews of claims. Each time an agent is given a chance to rate, a check is performed on the main random chain.

If this check is passed, the agent makes a review. The schematic process of rating is shown on Figure 3. It consists of executing the agent's rating strategy with the claim under review as input. This is done to enable a wide range of rating methods. The claim's other reviews, the author's review, the author's identity or the rater's own measurement of the claim ground truth may all play a role in determining the review value.

Once the review value is ready, a review object is created. It is appended to the review list on the claim and also to the rater agent's list of own reviews. With this, the review is published and the agent has finished the claiming process. The same agent can possibly make a review in the same round again, but only when rating another claim.

4.1.3 Applying Improvements

Improvements are the third main step of rounds. At this point, all claims and reviews of the current round have been published. As discussed in detail in Section 4.3, claim and review data is saved into a data structure formed by the list of agents. This list is passed to the selected improvement methods by the *ScenarioSimulator* for processing. All modifications are made on this mutable list of agents.

Improvement methods are implemented as separate handler classes. These handlers form a handler chain as per the *chain of responsibility* software design pattern. See Figure 6 for an UML model. Each handler is formed by inheriting from *reputation::AbstractHandler*. This abstract class defines the interface improvement handlers need to adhere to. It also gives the default behavior for the handlers. The method `handle(request)` is the default behavior of calling the next handler in the chain if any, or simply returning if the current handler is the last in the chain. This behavior is meant to be executed

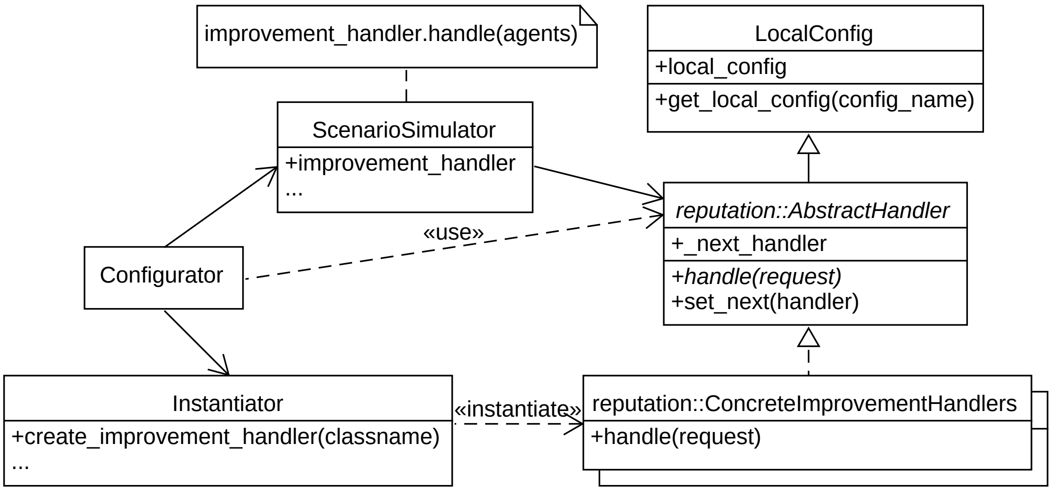


Figure 6: UML class diagram of improvement handlers. *Configurator* creates instances of the required improvement handlers via the *Instantiator* and sets the handler chain's entry point in *ScenarioSimulator*. When improvements are due, *ScenarioSimulator* calls the first handler with the agents data, which includes all claims and reviews. *LocalConfig* provides custom configuration for each individual improvement handler. This is discussed in Section 4.5.3 in detail.

at the end of all concrete improvement subclass `handle()` calls like this: `return ↪ super().handle(agents)`. *AbstractHandler*'s second method `set_next(handler)` is used to build the handler chain by giving which handler comes after this one is finished. Normally this is done by the configuration module during scenario preparation.

Organizing the improvements as a handler chain has multiple benefits. Firstly, improvements remain modular and flexible. Separate handlers can be added to or removed from simulations. The order of handling is free to change. Implementing new improvements is easily done by inheriting from the abstract handler class. Furthermore, the client code calling the handlers does not depend on which handlers are active. This means *ScenarioSimulator* is independent of improvements. Apart from a simple structure, this allows changing the order or composition of the chain during runtime. Although this is not currently supported, implementing this feature is simple. Finally, by putting improvement methods in a class, they are encapsulated. As such, there is a clear distinction between data and methods related to improvements and other parts of the system.

4.1.4 Reputation Calculation

After improvements have finished, the fourth and final step of a simulation round is calculating agent reputations. Reputation calculation methods are implemented with the strategy software design pattern. Each reputation scheme is implemented as a class that inherits from a common abstract parent class, *reputation::ReputationStrategy*. This class provides a simple interface containing the method `calculate_reputations(agents)`

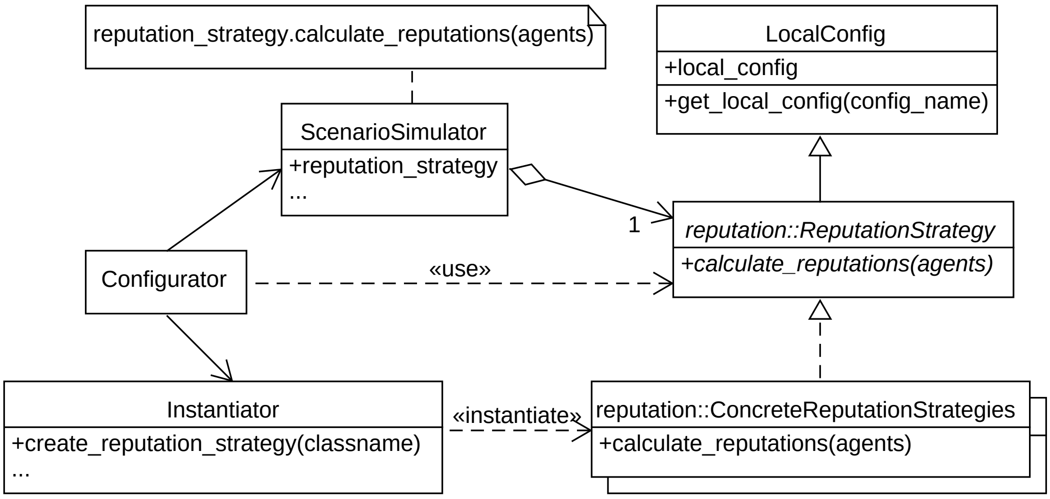

reputation_strategy.calculate_reputations(agents)
LocalConfig
+local_config
+get_local_config(config_name)
ScenarioSimulator
+reputation_strategy
...
1
reputation::ReputationStrategy
+calculate_reputations(agents)
Configurator
«use»
Instantiator
+create_reputation_strategy(classname)
...
«instantiate»
reputation::ConcreteReputationStrategies
+calculate_reputations(agents)

Figure 7: UML class diagram of reputation strategies. *Configurator* creates an instance of the selected reputation strategy via the *Instantiator* and gives it to *ScenarioSimulator*. At reputation calculation, *ScenarioSimulator* calls its reputation strategy object with the agents data, which includes all claims and reviews. *LocalConfig* provides custom configuration for strategy instances. This is discussed in Section 4.5.3 in detail.

which concrete strategies must implement.

After improvements have finished, *ScenarioSimulator* starts reputation calculation by calling this method on the reputation strategy it stores. The method takes the mutable list of all agents of the scenario as input. Then the reputation strategy's single responsibility is calculating agent reputations based on whatever method it employs and updating each agent's `global_reputation` attribute with it. This holds the agent's latest public reputation score. Before the first reputation calculation takes place, it holds an initial reputation set according to configuration before the first round.

Using the strategy pattern allows modularity and flexibility, much of the same advantages with improvement handlers. New reputation strategies are easy to add to Pyrepsys by creating a subclass of *reputation::ReputationStrategy* in the `reputation` subpackage and putting the classname on the import list. Strategies are independent of the client class, *ScenarioSimulator*. This means they are simple to exchange.

Additionally, encapsulating reputation schemes into classes makes them self-contained and forms a clear structure. Figure 7 shows the UML model of reputation strategies. implemented with the strategy pattern

4.2 Agents

4.2.1 Behavior Strategies

Behavior strategies serve as the implementation of agent's rating or distortion methods. The *behavior* subpackage contains related classes. Both rating and distortion strategies are implemented very similarly to reputation strategies as described in Section 4.1.4.
The involved classes are shown in the UML diagram in Figure 8. Both behavior strategy types share a common abstract ancestor called *BehaviorStrategy*. This class serves two main purposes. First, it describes a common interface for both rate and distort strategies. This consists of the `execute()` function, which takes as parameter the calling agent, some data that is needed for the rating or distortion method, and an optional random seed. Second, the common ancestor provides shared functionality for random number generation with the `rng()` method. This can be used by behaviors to start a branch-off random chain and generate random numbers.
Distortion and rating diverge with their own abstract adapter classes *DistortStrategy* and *RateStrategy*. Both are child classes of the common ancestor class. Their main purpose is to enable the same random chain functionality while also providing a different interface for both behavior types. *DistortStrategy* provides `distort()`, *RateStrategy* provides `rate_claim()`. Both classes wrap `execute()` calls to their respective behavior calls. Additionally, the adapter makes a preemptive check on the input data. *RateStrategy* allows *Claim* objects to be passed, while *DistortStrategy* allows measured claim quality values: integer and float.
Finally, concrete strategies of both types derive from the abstract adapter parents. *Configurator* creates the concrete behavior strategy instances via *Instantiator* and assigns them to each agent according to scenario settings. Agents then call upon their behavior strategies during simulation through the `execute()` interface.

4.2.2 Measuring Claims

Measuring is taking an approximation on the hidden true quality, or the ground truth of a claim. It results in a value that is comparable to the ground truth or any review on the claim. This is the measured claim quality.
Since assessing the quality of a claim depends on the assessing agent's know-how, experience or expertise, this is reflected in claim measurement as well. Agents have the attribute called Claim Truth Assessment Inaccuracy (CTAI) for this. This represents the accuracy the agents can guess a claim's ground truth. CTAI is given for each agent in the scenario configuration. The value of CTAI is the maximum error an agent can make when assessing a ground truth in each direction. This is represented in Equation 4.22.

$$C_{gr\ truth} - \text{CTAI} <= \text{measured quality} <= C_{gr\ truth} + \text{CTAI} \tag{4.22}$$

Whenever an agent needs to measure a claim, it calls its method `measure_claim()` This

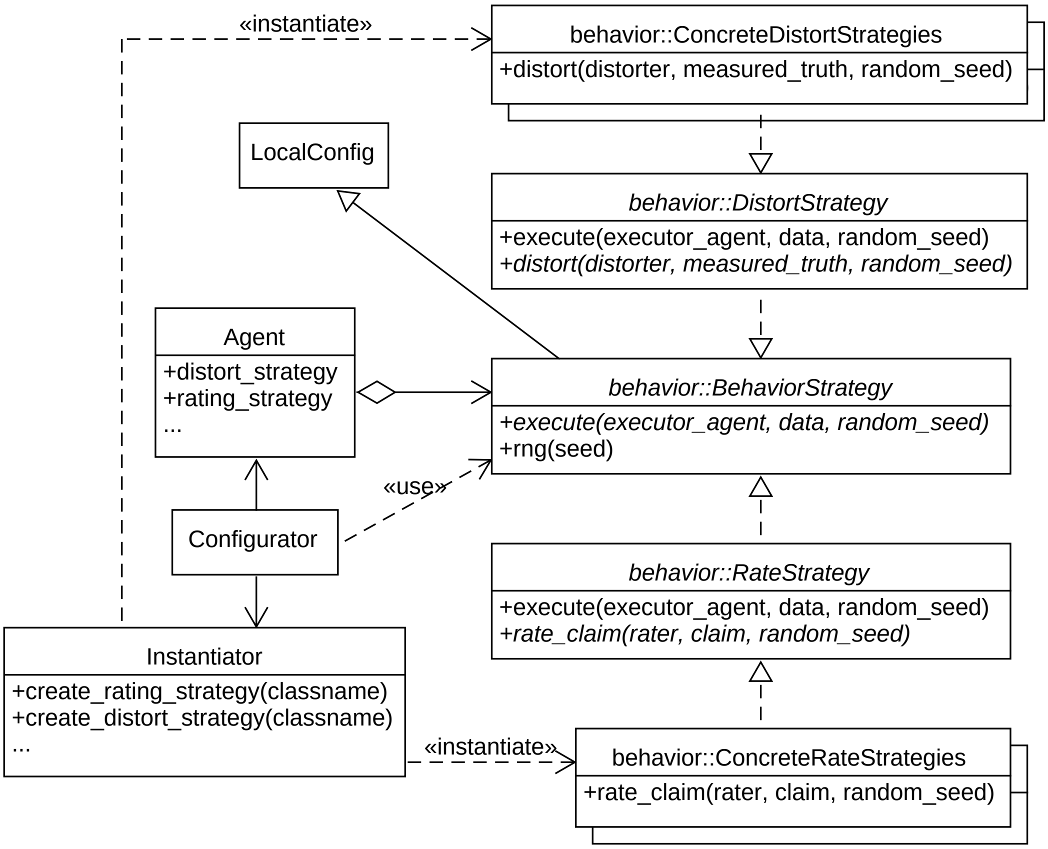

«instantiate»
behavior::ConcreteDistortStrategies
+distort(distorter, measured_truth, random_seed)
LocalConfig
behavior::DistortStrategy
+execute(executor_agent, data, random_seed)
+distort(distorter, measured_truth, random_seed)
Agent
+distort_strategy
+rating_strategy
...
behavior::BehaviorStrategy
+execute(executor_agent, data, random_seed)
+rng(seed)
«use»
Configurator
behavior::RateStrategy
+execute(executor_agent, data, random_seed)
+rate_claim(rater, claim, random_seed)
Instantiator
+create_rating_strategy(classname)
+create_distort_strategy(classname)
...
«instantiate»
behavior::ConcreteRateStrategies
+rate_claim(rater, claim, random_seed)

Figure 8: UML class diagram of agent behavior strategies. The abstract class *behavior::BehaviorStrategy* serves as a common base for both distort and rate strategies. Two interface classes *behavior::RateStrategy* and *behavior::DistortStrategy* both inherit from it. Concrete strategies are descendants of these two classes. *Configurator* creates behavior concrete strategy instances via the *Instantiator* and sets them on the agent using it. When rating or distorting, the agent calls `execute()` on the appropriate behavior strategy. *LocalConfig* provides custom configuration for strategy instances, discussed in Section 4.5.3 in detail.

takes as argument the claim to be measured and a random number generator. Since claims are measured on-demand, the generator is always a branch-off random chain. The measurement error is created as a random value with uniform distribution between [-CTAI, CTAI]. This is then added to the claim's true quality.

$$\text{measurement error} = \text{rng.uniform}(-\text{CTAI}, \text{CTAI}) \tag{4.23}$$

$$\text{measured quality} = C_{gr\ truth} + \text{measurement error} \tag{4.24}$$

The above equations summarize how claim measurements are made. Theoretically, an agent could find out the true quality of a claim by making multiple measurements. This is prevented by doing measurements inside the *Agent* class before any other parts of the code are called where the measured quality is needed. Currently only one such case exists. Measured quality is used as input to distortion. The distort strategy does not get the whole *Claim* object containing the ground truth. Instead, the measurement is made before calling distortion, and passed on as argument instead of the claim's object.

A disadvantage of using a simple uniform distribution to randomize measurements is exactly its simplicity. A uniform error can be seen as a kind of simple noise. Other kind of random distributions or even more elaborate models of assessing the quality of a claim might be needed. On these occasions, a viable way could be implementing a strategy pattern similar to agent behaviors and reputation strategies.

Depending on what approach a simulation takes, it might not want claims measured. Claim measurement can be seen as involuntary distortion on the agent's part. On the other hand, distortion represent willful manipulation of the claim's quality in the author review. This is the default approach. However, the claim measurement functionality is not needed when distortion strategies model all manipulation happening between spawning and publishing a claim. For these cases, it can be turned off. This is achieved by setting the CTAI of all agents to 0. Every measurement will simply return the claim's original ground truth in this case. Simulation results reflect this as if agents would operate with the ground truth itself when distorting.

4.3 Data Handling

4.3.1 Simulation Data Structure

Pyrepsys keeps simulation data in memory during simulation. The principle is that data is stored in hierarchical classes that represent the agents, claims and reviews. The top level owner of all simulation data of a scenario is *ScenarioSimulator*. The structure is shown in Figure 9.

The list of created agents is stored by *ScenarioSimulator* in a list called `agents`. Every agent is represented by an *Agent* class instance. Agents store their own claims they made in a list called `claims`. Claims are instances of the *Claim* class.

When claims are made, the claimer agent always appends an author review to their claim. This is a *Review* instance and is stored in the claim object as `author_review`. Reviews made by agents other than the claimer after the claim is published are also *Review* instances. They are stored at two locations. For once, they are appended to the claims they are a review of, in the *Claim* object's `reviews` variable. Second, every *Agent* object stores the reviews that agent ever made in a list called `reviews`. Although the the *Review* object of a review is referenced at two locations, Python's handling of variables ensures the review object exists only once. The drawback of this is that when removing a claim, it has to be removed from two places. The advantage is, it is much faster and easier to find all the review a specific agent has ever made. This is useful for some improvement methods, or possibly for rate or reputation strategies.

Both claims and reviews also store a weak reference back to their author. Weak references are provided by Python's `weakref` module. Objects linked with weak references are not kept alive when the all references to them are weak. This adds minimal overhead but big benefits. When traversing backwards from claim or review to author, the author weakref is called to retrieve the referred *Agent* object. However, were weakref not used, agents, claims and reviews would be referencing each other in a circle. This would prevent them to be destroyed (collected) properly once the simulation is done and ScenarioSimulator empties the agents list. Weak references prevent this, because they do not count as references when looking for whether an object is still linked from anywhere. When only weak references remain referring to an object, garbage collection will destroy it.

Passing the Data

All data reflects the latest state during simulation. Improvements act on this data and modify it accordingly. For example, the aging improvement removes old claims directly from the live agents list stored in *ScenarioSimulator*.

This approach has disadvantages with certain improvements that make only temporary changes to claims or reviews. For example, an outlier filtering removes extreme opinions. Once majority opinion shifts toward the opinions previously deemed an outlier, they need to be re-added since they are no longer outlier. With the approach used, it has to remove reviews from the two lists in agent and claim, and then store them until they are to be re-added. This is not an issue in itself, because improvement handlers are implemented as classes and can store data.

Another drawback shows itself at the end of the simulation. Here, the original unimproved state of the data that includes removed or disabled parts is not available for processing and evaluation. Metrics solve this issue however. This way partly the reason for how metrics were implemented as they are, called on various points of the simulation. At each simulation event where metrics are called, they export the needed data.

The advantage of keeping the data structure always in the live state is that it remains

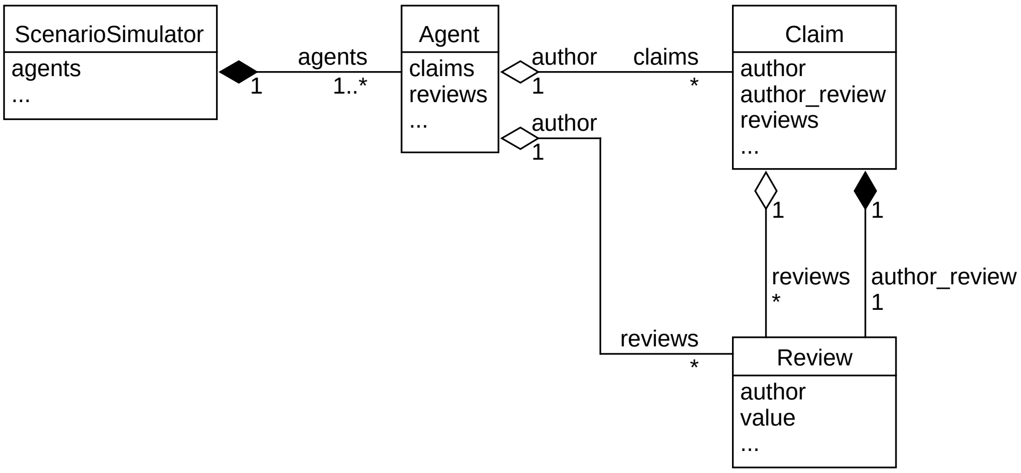


Figure 9: UML class diagram detailing how data is stored during simulation. *Agent* instances representing agents in the scenario are held in a list in *ScenarioSimulator*. Each agent holds a number of claims made by that agent. Claims have exactly one author review attached to them. Other reviews are held in a list in *Claim*. Agents also hold the reviews made by them in a list of their own. Claims and reviews link back to their authors with weak references.

free of clutter. Apart from this, simulation time is not increased by having to iterate through and check for inactive or removed objects.

Alternative Storage Approaches

Some alternative approaches were also considered. One alternative would have been storing the effects of improvements within the data structure with flags. For example, aging would add a flag `counted=False` to claims deemed too old. This would be good for improvements like outlier filtering, that take objects out of consideration and possibly re-add it later. Outlier filtering would only need to change a flag back to `True` to re-enable a review or claim.

One drawback of this would be that all the temporarily removed data objects would still remain in the live data lists. They would be iterated over many times during rounds, increasing simulation time. To prevent this, removed objects could also be stored in separate lists. In this case however, the approach is mostly identical to what was implemented anyway.

Another drawback is that the simulator needs to check for the various flags introduced. There can be many flags, like `counted`, `removed`, `boosted`, `weakened` etc. Once checks for flags are built into Pyrepsys, improvements would no longer be modular, because code specifically tailored to them would remain in the simulator constantly.

Another approach to organize the data would have been storing different states of the data. For example, also store a version without any modification by improvements, just the raw data that was generated by agents. In parallel, the actively improved version is also stored.

Generally, such data duplication is disadvantageous. It takes more memory and is harder to manage. Processing time also increases. Apart from this, the unimproved raw data would tell a false picture. For example, it would store all claims for all rounds, without telling some claims were removed by aging. Or store all reviews for all rounds, even though a couple of them would have been filtered inactive for some rounds. Improvements also have a feedback effect on the data generated in subsequent rounds. For example, a review of a claim removed by filtering can possibly change how another agent rates the claim. Only seeing the raw data that includes the filtered review, the decision (rating) process of the other agent could not be reconstructed properly.

So the two datasets could be analyzed strictly together only. This would be problematic, since the live view would hold the data from the last round only. There is no information about what happened when in rounds before the last.

Data in Metrics

Metrics are a special case of data storage. As mentioned above, one of the reasons they came to be was that data often needs to be saved along the time domain. An example is following the change of reputations in every round. For this purpose, metrics are hooked into points of simulation like the end of rounds and end of scenarios. These are called simulation events. With this, any snapshot of the data is possible.

Metrics get access to simulation data structure discussed above. Specifically, they receive the list of agents stored by *ScenarioSimulator*. From it, they are free to take and store data as they wish. Because metrics are implemented as classes, they can store data of their own.

The drawback of this approach is that data is stored multiple times after all, in the metrics. This takes up some additional memory. However, metrics rarely store a copy of the raw data only. Most often they do some kind of (pre)processing already at storage. For example, calculate an average of something per round, or make aggregates of agent data, etc. This justifies their separate data storing. When time-based raw data does need to be stored by metrics, it is only done on a need basis. Meaning when that metric is not needed, it is turned off and the system is no longer strained by holding unneeded data.

4.3.2 Internal and Agent-Exposed Values

Reviews, claim qualities and reputations can take up any value depending on the reputation scheme. The simplest case is when reviews are either positive or negative. This might be represented as 0 and 1. In other cases, discrete values are needed, like *good*, *ok* and *bad*. Finally another common case is when values are integers in a range. For example, the 5-star system using 1, 2, 3, 4 and 5 as possible values is common. By

Scheme	**Rate Values**	**Rating Settings**		**Representations**	
		Range	**Resolution**	**Internal**	**External**
Binary	+, -	[0,1]	1	0, 1	0, 1
Labels	good, ok, bad	[1,3]	1	0, 0.5, 1	1, 2, 3
5-star	1, 2, 3, 4, 5	[1,5]	1	0, 0.25, 0.5, 0.75, 1	1, 2, 3, 4, 5
Precise	1, 1.1 ... 4.9, 5	[1,5]	0.1	0, 0.025 ... 0.975, 1	1, 1.1 ... 4.9, 5

Table 4.2: Example rating schemes and Pyrepsys scenario configuration implementing them. The last two columns show how possible values are represented in the system. Using rating range and review resolution all common schemes can be configured. External refers to agent-exposed representation.

default, Pyrepsys also uses this approach with values between 1 and 9. This is however, decided by the scenario configuration.

The need for flexibility in reputation schemes necessitates a generic storage solution of values in Pyrepsys. This is the reason values are stored differently internally than they are displayed and used by agents. Agent-exposed representation is when a value is in the state facing the agents, the reputation system and display or export functions. This is what is mentioned in examples above. Internal representation is the way Pyrepsys stores values in the internal data structures.

Internally stored values are in the range [0, 1]. Agent-exposed values are controlled by scenario settings. The scenario configuration parameters *MIN_RATING* and *MAX_RATING* determine the possible range. In the code, whenever a variable stores internal values, it end with '`_i`', while variables of agent-exposed values end with '`_ae`'.

Every time a value is fetched from or is saved to another state of representation, it is converted. Conversion is done by two functions in the `helpers` module. These are `agent_to_internal()` and `internal_to_agent()`. Both have a shorter alias in the form of `a2i()` and `i2a()`. These functions take a single value v as input and return the converted equivalent in the other representation. Conversion follows equations 4.25 and 4.26.

$$v_{internal} = \frac{v_{ae} - \text{minimum rating}}{\text{rating span}} \tag{4.25}$$

$$v_{agent\ exposed} = (\text{minimum rating}) + (\text{rating span}) * v_i \tag{4.26}$$

Used together with resolution, it is possible to reproduce schemes like the example ones at the beginning of this section. See sections 3.8 and 4.3.3 for details on resolution handling. Table 4.2 shows how the scenario should be configured for common schemes. Agent-exposed representation gives numeric values even when they are meant as labels or other non-numeric choices. In these cases, behavior and reputation strategies and metrics have to take care of interpreting them as such.

4.3.3 Resolution

Resolution settings control what exact values are allowed for select variables during simulation. This section discusses implementation details. See Section 3.8 for conceptual discussion of resolutions.

Domains and Conversion Points

Pyrepsys defines two resolution domains. Each of these represent a resolution constraint for a part of the claiming, rating or reputation process. When a value that is part of a process enters a domain, it is converted to the configured resolution. Figure 10 shows which domains apply to what parts of the claiming and rating processes.

Domains determine the smallest allowed step for selected variable values. The two scenario parameters that control the two domains are `REVIEW_RESOLUTION` for reviews and `MEASURED_CLAIM_RESOLUTION` for measurements. Both of these expect the value of the smallest allowed step given in agent-exposed representation.

There are three transformation points as also seen on Figure 10.

1. on the result of a claim quality measurement, before the agent sees the measurement in its rating or distort strategy
2. after the execution of a rating strategy on the returned review, before publishing the review
3. after the execution of a distort strategy on the returned distorted quality, before publishing as author's review

Reviews and distorted qualities are results of the customizable behavior strategies of agents. Conversion of these results is done without any punishment for agents that do not prepare their values for the configured review resolution. All reviews are saved with this resolution, whether an agent's review or a claimer's author review.

Code implementing resolution handling is found in the `helpers` module.

Resolution Conversion

When a value that is part of a process enters a domain, it is converted to the configured resolution. The main function for converting between resolution domains is `convert_resolution()`. It expects two arguments. First, `number` holds the value to be converted. Second, `target_resolution_domain` gives which resolution it should convert to. This is given in the form of an enum defined in `helper_types`. The purpose of `convert_resolution()` is to call the `find_nearest_step()` with the appropriate arguments to find which step the value to convert falls nearest to.

The function `find_nearest_step()` accepts `number` holding the value to be converted and `steps_sorted_list`, a list sorted by ascending order and containing all the allowed steps of the chosen resolution domain. It is this function's job to effectively convert to the new resolution by selecting a possible step from the list based on the value of `number`. For this, it uses a bisection algorithm `bisect_left` from Python's `bisect` module.

Left bisection efficiently searches for an element in a sorted list. Originally it is meant to be used for efficiently inserting new items into a sorted list. Given a sorted list and a new item, bisect returns the index where the item should be inserted into the list to keep it sorted. Since it uses binary search for finding this, its efficiency is $O(logN)$. The difference between left or right bisection is if the element to insert is already in the list, should the returned insertion index be left or right of the existing equivalent element(s).

With the point of a would-be insertion found, `find_nearest_step()` decides on the returned resolution step. The index returned by left bisection is denoted by `i`. If the input `number` coincides with resolution step `steps_sorted_list[i]`, that is returned. If not, then `number` is between `steps_sorted_list[i]` and `steps_sorted_list[i-1]`. The distances from both are calculated as seen below.

```
diff_left = round(abs( steps_sorted_list[i-1] - number ), 8)
diff_right = round(abs( steps_sorted_list[i] - number ), 8)
```

Rounding is needed to correct small floating-point errors that can result from working with small numbers. The step with the smaller distance is returned. If the distances are equal, the larger, right-neighboring step is returned.

Creating Resolution Steps

Ordered lists of possible resolution steps are created for every resolution domain. These are called step lists or resolution ladders. The step lists are sorted by ascending order and are created during scenario configuration by the function `_configure_system_resolutions()` in the `helpers` module. The method is simply starts with the smallest value, increments it with the resolution's step and adds it to the step ladder. This is repeated until the maximum allowed value is reached. Each step list is stored in a module-level variable in `helpers`.

The function `_configure_system_resolutions()` is an exception from all others related to configuration, which are located in the `config` module in the *Configurator* class. The reason for putting this function in `helpers` instead of `config` is to avoid a circular import dependency between the two modules. An alternative solution was to keep the functionality in the *Configurator* class. In this case, the resolution steps would have to be requested via the `get()` method of *Configurator*, or with a similar call. This adds an overhead, and since resolution conversion is a very common operation, so it can build up to a noticeable factor. Because of this, the resolution ladders are kept locally in `helpers`.

To allow (re-)calculation of the resolution steps every time the scenario configuration changes, *Configurator* must let the `helpers` module know when this happens. This is solved with callbacks from *Configurator*. The function for calculating resolution ladders

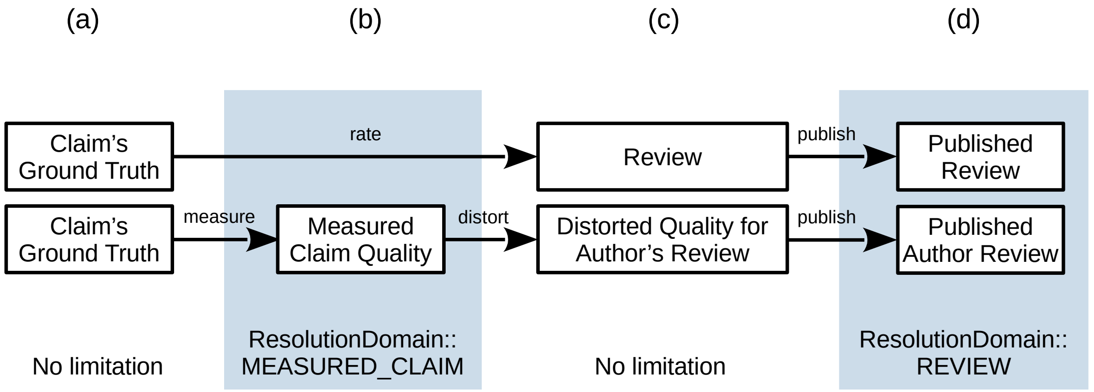


Figure 10: Resolution domains in various stages of rating (top) and claiming (bottom). (a) Ground truths have no limitation and can take up any value within the rating span, stored as internal representation. (b) Claim quality measurements are produced in the *measured claim resolution domain*. (c) In their distorting or rating strategies, agents can compute an output without regard to resolution constraints. (d) Once the produced values are published, they are converted into the *review resolution domain*.

is registered with *Configurator*, which calls it every time the scenario configuration is updated. For more discussion on configuration updated callbacks, see Section 4.5.4.

4.4 Random Number Generation

Python's `random` module is used for generating random numbers. Many parts of Pyrepsys uses random numbers for simulation. These include:

- Ground truth of new claims
- Random error on claim quality measurements
- Checks against an agent's claim or rate probability
- Some agent behavior strategies

In order to facilitate comparability and repeatability of simulations, Pyrepsys implements repeatable random number generation. Python's `random` rolls its internal state every time a random number is taken from it. If the state of the generator is known and can be restored, random number sequences can be recreated. Such sequence of random numbers are called random chains here.

Repeating the chain of a random number generator is possible by seeding it. Scenarios provide a configuration parameter called `seed` for this purpose. The class *ScenarioSimulator* owns an instance of the random number generator created with the `random` Python module. Before simulation of a scenario, the generator is seeded with the above parameter. This generator provides the so-called main random chain. It serves random values for checks and uses where the same number and type of random value is needed regardless of what execution path the simulation takes. These are for example checks whether agents will claim or not, provide a review or not, and providing random seeds for throwaway random chains.

Branch-off random chains or throwaway chains are supplementary random number sequences. These are used when it is not known in advance how many or what type of random values are needed, or whether any will be needed at all. Such cases are claiming and rating randoms, or agent behavior strategies.

Throwaway chains are made by branching-off the main chain. That means a random value is rolled from the main generator, and used as seed for the throwaway chain. To provide a constant usage of the main chain, every time there is a possibility that a branch-off chain will be needed, a random number will be generated for its seed. This way the chain is used equally regardless of the scenario's settings.

An example illustrating the purpose of the two random chains is claiming. Claiming requires three random values:

- one for the claim probability check to see if the agent will attempt a claim
- one for generating a ground truth (quality) in the new claim
- one for the measurement error when the agent assesses the claim's quality

The first one of these must be generated whether the agent claims or not. The second two only if the check to claim succeeds and claiming is attempted. Changing the number of agents or agent's claim probabilities would modify which agents enter the the claiming process, and so the agents claiming afterward the shift would get different random values than before. To prevent this, each claim opportunity uses exactly two random values off the main chain: one for the probability check, and another as a seed for the branch-off random chain. If the agent passes the probability check, the generated seed is used to create the branch-off random sequence. The random values for the new claim's quality and the measurement error are taken from this throwaway chain, which is destroyed after the claiming. In this manner, all agents receive the same inputs from the simulator even between scenarios, provided that the order of the agents stays the same.

4.5 Configuration

This section discusses topics related to configuration in Pyrepsys. It also covers some scenario parameters, but not all. For a usage-oriented description of every possible scenario parameter, see appendix A instead.

4.5.1 Scenario Definition

Scenarios are expected as `.yaml` files located in the `scenarios` folder. Each file contains a list of parameters in yaml format, telling what the scenario is about. Parameters include the reputation strategy, used improvement handlers, agents, enabled metrics, resolution, rating range, the initial seed and other settings.

Scenario files can either be fully or partially defined. Full definition requires the scenario give an explicit value for all possible parameters. Partially defined scenarios contain only a subset of all settings.

Scenario files can fill two different roles when running Pyrepsys. First, regular scenarios are the separate simulation units that will be simulated after each other. These are given as a list to simulate, typically more than one per invocation. Second, the default scenario is used as a fallback for parameters while simulating the list of regular scenarios. Only one scenario file can be given as default scenario every Pyrepsys invocation. The two kinds of scenarios form a two-level system. Whenever a regular scenario is simulated and a parameter is requested, the system checks among the parameters from the scenario file. This is called the active configuration. If a parameter is not defined there because the scenario is not fully defined, then the default scenario serves as fallback. The requested parameter is returned from the default configuration. For this reason, the scenarios used as default scenario must always be fully defined.

Defining scenarios with this two-level configuration system is simpler. Usually only one or at most a few parameters are varied between scenarios to see the isolated effect of the changes. Here, the default scenario should contain the baseline values for scenario parameters, fully defined. Then, each specific simulated scenario can be partially defined only containing parameters that differ from the default configuration. Pyrepsys also provides a scenario creator utility, which uses the same approach to create scenarios with combinations of given parameter values.

4.5.2 Reading Scenario Files

Reading configuration from scenario files to memory is done by the *Configurator* class. It provides two separate methods for reading active and default scenarios. Both are wrappers for the same class method responsible for reading scenario files, named `_config_from_file_to_memory()`. After opening and reading the file contents, it calls on the third party `pyyaml` module to read the contents into a dictionary. A name for the scenario is added to the dictionary by taking the filename without the `.yaml` extension. The function also generated the hash of the scenario file it just read using the sha256 algorithm. This will be logged to provide a quick check whether two scenarios of the same name contain the same settings.

4.5.3 Configuration Storage Locations

Multiple locations are responsible for storing configurations in Pyrepsys. *Configurator* is the main source of settings. Apart from that, locally stored config collections also exists, and there is one local configuration cache.

Configurator

Configurator is located in the `config` module and is responsible for most configuration-related functionality. Its main tasks are reading scenario files, preparing other modules and classes for simulation, and providing configuration parameters when requested.
The class method `get()` can be used to request config values from *Configurator*. It takes an argument with the name of the config parameter. This function implements the two-level configuration retrieval. First the active config is checked, and then the default if the active configuration does not contain the requested parameter.
Configurator's storage of active scenario settings is changed between scenarios, while the default config remains the same for the whole duration of a Pyrepsys run.

Local Configuration of Strategies and Handlers

Some behavior or reputation strategies, improvement handlers or even metrics are parametrized and so can have configuration values attached to them. An example for how these are defined in scenario files is found below.

```
improvement_handlers:
  - name: "Aging"
    limit: 6
  - "Weights"
```

Here, aging is defined with one parameter called `limit`. When configuration is attached in this way, the original param value is replaced with a dictionary, and the `name` param is used to hold what was just a string before. For comparison, the `Weights` improvement handler uses no parameters in the above example, so only the implementig class' name is needed as a string.
Strategies, handlers or metrics that are extended with parameters like this are interpreted by the *Configurator* class in a method called `_unpack_extended_config_list()`. After an instance of the appropriate class is created, it receives a dictionary containing its parameters if there are any. The storage solution is the class *LocalConfig*, from which all strategies, handlers and metrics inherit. This class provides a method similar to *Configurator*'s `get()` for accessing the local config. These are then accessible from within the class methods of strategies, handlers or metrics.
Storing these parameters with the class instances instead of a central repository like *Configurator* allows instantiation of different versions of the same class. Simply the local config needs to be specified with different values.

Configuration Caches

There is one occasion where a Pyrepsys module uses a local cache for scenario configuration that normally would be requested from *Configurator*. This caching appears in the `helpers` module. Before its introduction, the conversion functions between internal representation and agent-exposed representation made multiple config param requests each. These are very frequently used functions, because they are called every time an internally stored value is fetched, or when an agent-generated value is stored. While *Configurator*'s `get()` is fast, the sheer amount of calls made incurred a noticeable overhead. When this local cache was introduced, it resulted around 70% improvement in the cumulative time spent in the function that converts from internal to agent-exposed representation. This can mean around minute that was shaved off. Other functions in helpers also benefit from speed improvement, albeit to a lesser degree.

Caching works by storing needed configuration parameters and precalculated values in the `helpers` module. Module functions can then access the cache dictionary instead of making a request from the `config` module. *Configurator* updates the cache via a callback each time the scenario configuration is modified.

4.5.4 Update Callbacks

Configurator allows registering callbacks for configuration changes. These are called every time the default or active scenario configuration changes. Other parts of Pyrepsys can use this to be notified of changes in configuration.

The method to register the callback functions is `register_config_updated_callback()`. Every time a new scenario file was read and parsed, *Configurator* executes all registered callbacks. Update callbacks are used by the local config cache in `helpers` and the resolution step ladder creator method. Refer to Section 4.3.3 for why resolution handling is solved this way.

4.5.5 Setting Up Simulations

The `main` module serves as an entry point and coordinates the Pyrepsys classes to perform simulations. After preparatory tasks like logging setup and creating the output folder, instances of *ScenarioSimulator*, *ResultsProcessor* and *Instantiator* are created. The system-wide *Configurator* instance is also requested. The `config` package provides `getConfigurator()`, which returns the same instance of *Configurator* every time it is called. *ScenarioSimulator* receives *ResultsProcessor*, the configurator gets the *Instantiator* instance and the output directory path.

Now, the scenario files can be read and the objects can be configured. First, the default scenario settings are read. Then, for each scenario, its file is read, *ScenarioSimulator* is configured, *ResultsProcessor* is configured, then begins the simulation. This repeats until all scenarios were simulated.

The scenario simulator's setup is divided into three parts: improvement handlers, reputation strategy and agents.

Reputation Strategy and Improvement Handlers

Reputation strategy and improvements have a similar setup. First, the params storing them are requested. Then, a method called `_unpack_extended_config_entry()` parses parameters that are extended with local configuration described in Section 4.5.3. *Instantiator* is used to create reputation and improvement handler object instances. The instances receive their local config is any was set for them, and they are assigned to the *ScenarioSimulator*. For the improvement handlers, the improvement chain is tied together in order of appearance in the scenario file, and the scenario simulator only receives the first handler.

Agents

For setting up agents, the agent base behavior definitions also need to be parsed first. Base behaviors are template settings for agent definitions, defined in scenario files under the parameter `agent_base_behaviors`. Each of them must contain all possible agent parameters, along with the base behavior's name. For example:

```
agent_base_behaviors:
  - name:                 "HonestAccurateRater"
    distort_strategy:     "DistortDoNothingStrategy"
    rate_strategy:        "RateFromOwnExperience"
    claim_range:          [0, 1]
    claim_probability:    0
    rate_probability:     1
    claim_truth_assessment_inaccuracy: 0.0625
```

Base behaviors can then be used when defining agents. Below, the above definition is used to define two agent types. Both of them will have 10 agents created of the type. The second type overwrites the base behavior's `claim_probability` of 0 with 0.2.

```
agents:
  - amount:               10
    base_behavior:        "HonestAccurateRater"
  - amount:               10
    base_behavior:        "HonestAccurateRater"
    claim_probability:    0.2    # overwrite claim% from the base behavior
```

The typical usage method for base behaviors is defining a set of base behaviors in the default scenario's file, and then using these definitions in other scenario files that will be simulated. Otherwise, `agent_base_behaviors` also adheres to the two-level configuration system. If it is re-defined in an active scenario's file, that definition overwrites the one in the default scenario settings.

Each agent type's rate and distort strategies are created in a similar manner to reputation strategies and improvement handlers. The local config parameter extensions are parsed, then instances are created. With the agent's configuration parameters and behavior strategies ready, the *Agent* instances are created.

This is done by *ScenarioSimulator*, with its method for this task called `create_agents()`. Agent objects are saved by scenario simulator into the agents list.

Metrics and ResultsProcessor

Metric instances are special in the regard that they are persistent between all simulated scenarios of a Pyrepsys invocation. This is because metrics usually compare multiple scenarios, and they need to be able to collect and store data from each of them. *ResultsProcessor* stores metric instances for this reason.

During configuration, *Configurator* checks if *ResultsProcessor* already stores instances of the listed metrics of scenarios. If not yet, the same process from earlier sections involving local configs and the *Instantiatior* is used to create the missing metric instances. *ResultsProcessor* receives them and will keep them until Pyrepsys terminates.

Including or excluding a metric from data collection during a scenario is determined by the `metrics` parameter in scenario files. The two-level system of default and active configurations applies to it like to any other parameter. The typical usage is listing metrics used in the default scenario's file, and not defining any metrics in the simulated scenarios. This results in the same default metrics being used for all scenarios. If any scenario defines the `metrics` parameter, *only the metrics found in this definition* will be used for the scenario, since it completely overwrites the one in the default. To add an extra metric along the defaults to a scenario, it needs to be listed explicitly with the default metrics as well. To exclude one compared to the default definition, all other not-excluded metrics need to be listed.

4.6 Results Processing

Pyrepsys offers optional built-in results processing to form a full pipeline from simulation to data analysis and visualization. The `results_processor` and `metrics` modules can collect, process and export simulation data as graphs or tables. *Metric* classes define what data is collected, and how it is processed and exported. *ResultsProcessor* contains the metrics, notifies them of events and funnels simulation data to them. Together, metrics and *ResultsProcessor* form a publish-subscribe architecture. The UML diagram of these classes is found in Figure 11.

4.6.1 Simulation Events

Pyrepsys defines four simulation events as listed in Table 4.3. Each time execution reaches one, the event is triggered by notifying the results processor of it. Then, *ResultsProcessor* notifies the metrics of events that concerns them.

Two events have fixed consequences. The `BEGIN_SCENARIO` event calls every active metric's `prepare_new_scenario()` method, which is used to make preparations before the scenario starts. `END_OF_SCENARIO` calls the `export()` method of active metrics, which is used to make post-processing on the data, assemble graphs or tables, and export them into files.

Event	**Called Metric Method**
BEGIN_SCENARIO	`prepare_new_scenario()`
END_OF_ROUND	`calculate()` if registered
END_OF_SCENARIO	`calculate()` if registered
END_OF_SIMULATION	`export()`

Table 4.3: Simulation events in Pyrepsys. *ResultsProcessor* is notified of events from other parts of the code. It notifies metrics based on what event was triggered and which metrics are active. The second column shows which methods of active metrics are called.

The two other events are `END_OR_ROUND` and `END_OR_SCENARIO`. Each activated metric subscribes to none, just one or both of these two. Whenever these events trigger, the `calculate()` method of subscribed metrics is called by *ResultsProcessor*. This is the time metrics can save data from the simulation system and do pre-processing on it.
Which of the two events a metric subscribes to is determined by what kind of reporting the metric does. In any case, simulation data is passed for `calculate()` for both events in the form of the agents list, the same one saved by *ScenarioSimulator*. Depending on which event it is, the just ended round's number or the ended scenario's name is also passed.

4.6.2 Metrics

Generally there are two types of metrics: data exporters and graph plotters. The distinction is not strict however, and metrics can even have both functions at the same time. It makes sense to divide metrics along what kind of reporting they do. When a metric calculates a specific kind of value, statistic or aggregate, it can make sense to produce graphs and data exports from the same metric, since the data is available anyway.
Every metric is defined as a child of the abstract *metrics::Metric* class, as seen in Figure 11. Then, selecting metrics for usage is done through the scenario configuration by giving its classname. The initialization method contains a metric's name and its events of interests. Additionally, every metric implements at least the three previously discussed functions. Graph plotting metrics in Pyrepsys use `matplotlib`, while data exporters use Python's `csv` module. Exported files are saved to an invocation-specific output directory called the artifacts directory by default.
Using metrics with Pyrepsys is optional. The costs of using them is the extra memory needed for storing collected data, and the additional processing time for dealing with the data and creating the export formats. The memory footprint obviously depends on how big of a simulation data is amassed and how many separate metrics store parts from it. Processing time spent on metrics was observed to be tens of seconds when simulation takes a few minutes.

Mark	**Meaning**
perf	Test does profiling or is a benchmark
manual	Test needs manual evaluation of results
long	Test takes a longer time than most others

Table 4.4: Pytest test markers in Pyrepsys. Relevant tests are flagged with them. Marks can be used to select which tests to run or exclude from running.

The benefit of metrics is the convenience of automatic reports and graphs exported based on simulation data. Flexibility and extensibility is also fulfilled. Adding new metrics is as easy as with any other behavior or reputation strategy, or improvement handler.

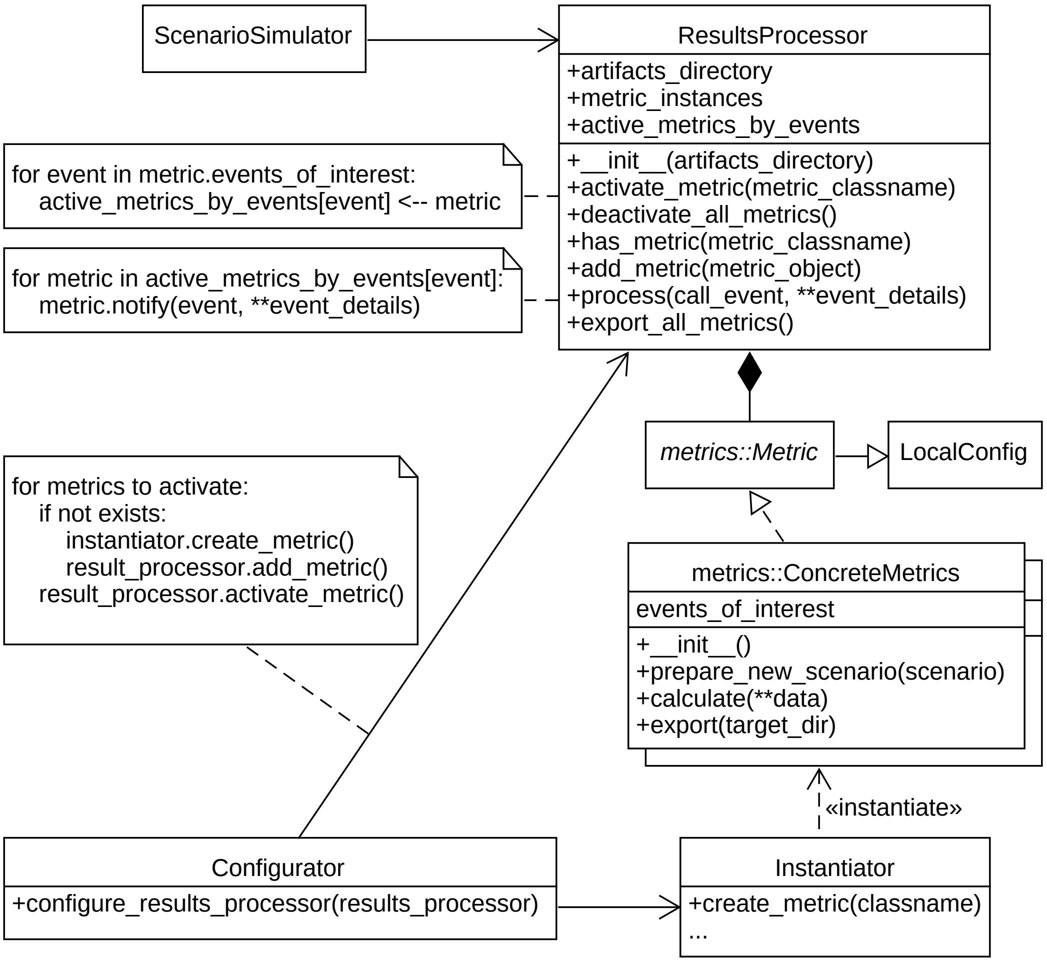
4.7 Other Facilities

4.7.1 Automated Self-Testing

Pyrepsys contains an automated testing suite. It can be used to verify the integrity of functionality under the test coverage. Tests can also be added to perform sanity checks of specific components, like metrics. Such tests contain scenario configuration that is known to produce an expected and explainable results. Finally, the test suite contains tests that profile and benchmark the simulator. Using these the most time-consuming parts of the simulation can be identified.
Testautomation is provided by Pytest. Tests reside in the `tests` directory in Python files. Each of them is a function and are found automatically by Pytest. To allow some control over what tests to run, Pyrepsys uses Pytest's mark feature. Some tests are marked with the identifiers found in Table 4.4.

4.7.2 Logging

Pyrepsys logs notable events and information in a logfile saved to the artifacts directory. This is implemented using Python's `logging` module.
Logging is set up by `pyrepsys.main`. Messages are logged into the logfile called `simulation.log`. They are also displayed on the console in a shortened format. Each Pyrepsys module gets its own logger with the `logging.getLogger()` method. It returns the logger after the `name` parameter provided to it, which is chosen as the name of each Pyrepsys module.
Messages of different severity are logged, including debug, info, warning and error. By default, only info level messages or more severe are displayed. If debug messages are needed, the default level can be overwritten in the `pyrepsys.main` module. It is also possible to change the severity of selected Pyrepsys modules only. This is beneficial when debugging a specific module.


ScenarioSimulator
ResultsProcessor
+artifacts_directory
+metric_instances
+active_metrics_by_events
+__init__(artifacts_directory)
+activate_metric(metric_classname)
+deactivate_all_metrics()
+has_metric(metric_classname)
+add_metric(metric_object)
+process(call_event, **event_details)
+export_all_metrics()
metrics::Metric
LocalConfig
metrics::ConcreteMetrics
events_of_interest
+__init__()
+prepare_new_scenario(scenario)
+calculate(**data)
+export(target_dir)
«instantiate»
Configurator
+configure_results_processor(results_processor)
Instantiator
+create_metric(classname)
...

```
for event in metric.events_of_interest:
    active_metrics_by_events[event] <-- metric
```

```
for metric in active_metrics_by_events[event]:
    metric.notify(event, **event_details)
```

```
for metrics to activate:
    if not exists:
        instantiator.create_metric()
        result_processor.add_metric()
    result_processor.activate_metric()
```

Figure 11: UML class diagram of the results processor and metrics. Metric instances are created by *Configurator* via *Instantiator*. *ResultsProcessor* registers metrics according to what event they are listening to. *ScenarioSimulator* and the `main` package trigger events on the results processor. Metrics are notified of events that interest them. *LocalConfig* provides custom configuration for each individual metric. This is discussed in Section 4.5.3.

5

Evaluation

This chapter presents experiments performed with Pyrepsys. First, the used metrics are introduced. Then, the various experiment are described, first the parameters and expectation, then the results of simulations. Finally, a discussion formulates takeaways from the conducted simulations.

5.1 Metrics

Effectiveness is measured as a global, system-wide accuracy, as opposed to an agent-utility approach. This is chosen because measuring individual agent's success makes less sense, since agents can not decide who they transact with.

The main metric for system accuracy is reputation scores compared to claimer's inaccuracy. Claimers who publish accurate and reliable author's reviews should have a good reputation, while, inaccurate author's reviews should result in a lower reputation. This way, an ideal linear relationship exists between inaccuracy and reputation, where the lowest possible claiming inaccuracy should map to the highest reputation, while the theoretical largest inaccuracy in the worst possible reputation. This ideal line is shown as a dashed diagonal in plots of this metric. If an agent is above the line, it managed to cheat the system by getting a better reputation than its inaccuracy and honesty would proportionally warrant. If under, the agent is punished harder than it should be under straight proportions.

Total claiming inaccuracy is determined by two factors: measurement inaccuracy and distortion. Since measurement is not under the agent's control, it is also called the involuntary inaccuracy. Distortion is fully each agent's doing, so it is voluntary and also called dishonesty.

$$\text{Total Claiming Inaccuracy} = \text{Measurement Inaccuracy} + \text{Dishonesty} \tag{5.27}$$

This is computable for a claim C_A made by agent A using the claim's author review and its ground truth.

$$\text{Total Claiming Inaccuracy}(C) = |Q_{true,C_A} - R_{authorA,C_A}| \tag{5.28}$$

Averaging every claim of an agent gives the *Average Total Claiming Inaccuracy* of that agent.

Average Inaccuracy of Raters measures how close regular reviews fall to the ground truths of rated claims. It is calculated as

$$\frac{\sum_{C \in \text{all claims}}((\text{average of review on } C) - Q_{true,C})}{\text{number of all claims}} \tag{5.29}$$

This metric is most influenced by the setup of agents and their behavior strategies. For this reason it can be used less to judge the effectiveness of reputation or improvement schemes, and more to see how the agent population rates. When more dishonest or malicious raters are present, this metric will indicate that with a larger average inaccuracy.

5.2 Setup

Experiments are set up and simulated with Pyrepsys. Each one consists of multiple scenarios with one or two selected parameters that are varied. Most other parameters are kept the same to allow clear comparison. Fixed parameters are

- rating range: $[1, 9]$
- rating resolution: 1
- measurement resolution: 0.5
- reputation strategy: `BasedOnAvgDifferenceOfClaimsAndReviewsWithWeight`
- initial reputation: 5
- number of rounds: 80
- seed: `pyrepsys_random_seed`

Varied parameters are the improvement techniques and agent population composition. Agent base behaviors shown in Table 5.1 are defined as templates. Apart from the behavior strategies, the default agent parameters are

- CTAI: 0.125
- claim range: [0,1], or [0.5,1] for `HonestClaimer` and `HonestClaimerRater`
- claim probability: 0.8 for claimers, 0 for only raters
- rate probability: 0.8 for raters, 0.1 for only claimers

Experiments are each identified with a letter.

Base Behavior	**Behavior Strategy**	
	Distort	**Rate**
`HonestClaimer`	`DoNothing`	`DoNothing`
`DishonestClaimer`	`MaxSometimes` (0.5)	`RateFromOwnExperience`
`HonestRater`	`DoNothing`	`LinearFromClaimerReputation` (7)
`InfluencedHonestRater`	`DoNothing`	`BetweenAuthorReviewAndExperience`
`MidrangeRater`	`DoNothing`	`Flatten` (0.5)
`MaliciousRater`	`DoNothing`	`RateInvertedSlope`
`HonestClaimerRater`	`DoNothing`	`RateFromOwnExperience`
`DishonestClaimerRater`	`MaxSometimes` (0.5)	`LowrateHonestClaimers`

Table 5.1: Agent base behaviors used as template in simulations. Values in parenthesis show parameters for behavior strategies that need them. `MaxSometimes` has *chance* of claiming the highest quality.

Experiment A: effect of aging

This experiment varies two parameters, both over three possible settings, making nine scenarios total. First, improvements are varied as 1) no improvements, 2) aging with limit 25 and 3) aging with limit 10. Then, the number of `MaliciousRater` agents are varied as 1) none 2) 30 and 3) 60. `HonestRater` and `MidrangeRater` extend the agent population so that there are always 60 raters with these two in equal numbers. Lastly, all scenarios include 10 `HonestClaimer` and 10 `DishonestClaimer` agents.

Experiment B: malicious raters

In this experiment, the ratio of dishonest raters is increased gradually. Since reputation calculation is based on the honesty of claim valuations in the form of author reviews, dishonest raters who otherwise never claim will not get a reputation adjustment.

The experiment has 10 `HonestClaimer` and 10 `DishonestClaimer` agents, and a total of 60 rater agents. These are made up of 1) none 2) 20 3) 40 and 4) 60 `MaliciousRater` agents. `HonestRater` and `MidrangeRater` agents fill the remaining rater spots equally, until all scenarios have 60 raters.

The four versions are simulated with and without the weights improvement enabled, totaling at 8 scenarios.

Experiment C: dishonest claimers

This experiment has three versions, each a sub-experiment. The first one (C/I) increases the ratio of dishonest claimers. All scenarios have 60 raters, 20 of each type: `HonestRater`, `MidrangeRater` and `InfluencedHonestRater`. The `HonestClaimer` to `DishonestClaimer` ratio is changed as 1) 10:0 2) 8:2 3) 2:8 and 4) 0:10. The weighted and weightless versions are simulated additionally, totaling at 8 scenarios.

Since a third of the agents are influenced by author reviews in their rating decisions, claimers are expected to be able to manipulate these agents into a better rating than the claim's deserved quality. However, the other agents are also honest and quite numerous, so combined they are expected to keep the dishonest claimers at bay.

The second (C/II) experiment has the rater types `HonestRater` and `MidrangeRater` removed, so only 20 `InfluencedHonestRater` remain as raters. Compared to (C/I), the rater, or consumer agent population becomes more susceptible to influence. This means they take author reviews of claims more seriously overall. These agents are expected to be more easily manipulated, so dishonest claimers can possibly get a better reputation than before.

The third (C/III) version fixes the ratio of honest and dishonest claimers, 10 agents each type. However, the 10 `DishonestClaimers` make dishonest claims increasingly often: 1) never 2) 50% 3) 85% and 4) 100%. This aims to find out whether there is an upper limit to frequency of dishonest claimers still tolerated. Similar to (C/II) only 20 `InfluencedHonestRater` agents take part as raters. However, their rate strategy is varied between 1) the usual for this agent type `RateBetweenAuthorReviewAndExperience` and 2) `RateFromOwnExperience`. This changed the normally influenceable raters to make rate decisions purely from their own opinion.

Experiment D: effect of weights

This experiment also has three sub-experiments and it aims to find out the effect of weights as improvement technique. Because weights are calculated from reputations, both claimers and raters should get reputation adjustments. For this, all agents need to be claimers, including the raters. This is often not the case in reputation systems for people, but is common for algorithmic reputation environments, where agents are machines.

(D/I) uses `HonestClaimerRater` and `DishonestClaimerRater` agents in the numbers 1) 18:2 2) 14:6 3) 10:10 and 4) 2:18. Improvement techniques are varied as 1) Aging with limit 30 and 2) Aging with limit 30 and Weights.

(D/II) makes honest raters dogmatic, in that they will tendentially give higher reviews for claims that come from good-reputation agents, and lower for claims with lower-reputation authors. This is the `LinearFromClaimerReputation` rate strategy with limit of 7. Limit gives the point above which reviews are amplified higher, while below it they are lowered. Similarly, dishonest agents become selfish saboteurs. They will always give the highest possible author reviews (`MaxSometimes` with chance 100%). When rating, they will rate everything with the lowest score, using `LowrateAll`. Improvements and honest-dishonest numbers stay the same.

(D/III) changes honest raters from dogmatic to rating based on their own experience. Reputation plays no role in their reviews. Everything else stays the same.

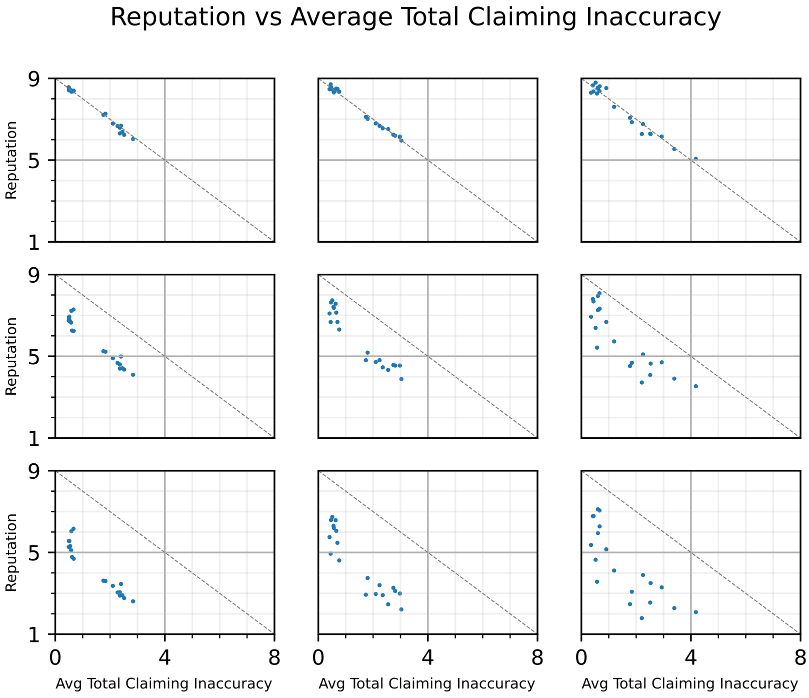


Figure 12: Average Total Claiming Inaccuracy and reputation scatter diagram of experiment A: effect of aging. Aging increases toward the right column, while malicious raters increase toward the bottom row.

Experiment E: measurement accuracy

This experiment is for finding out the influence of measurement accuracy. The CTAI of raters is configured as 1) zero 2) 0.125 3) 0.25 and 4) 0.375. A CTAI of 0.125 increment means 1 full score in agent-exposed representation in the integer rating range [1,9]. Every scenario has 5 `HonestClaimer` and 5 `DishonestClaimer` agents, both who are set not to rate at all. Raters use the rate strategy `RateFromOwnExperience` and are created in the amounts 1) 10 and 2) 1. The purpose of this is to find out whether more agents are able to combat the bigger inaccuracy better than only a single agent can.

5.3 Results

Experiment A: Effect of Aging

Figures 12, 13 and 14 show the results of the aging experiment. The claiming inaccuracy diagram on Figure 12 shows that more malicious agents pull every agent's reputation down from the ideal line. The Figure shows the two distinct groups of honest and dishonest claimers. Each dot is a claimer agent. Honest claimers form the groups on the left in each subfigure. They have better reputation than the other group, which are the

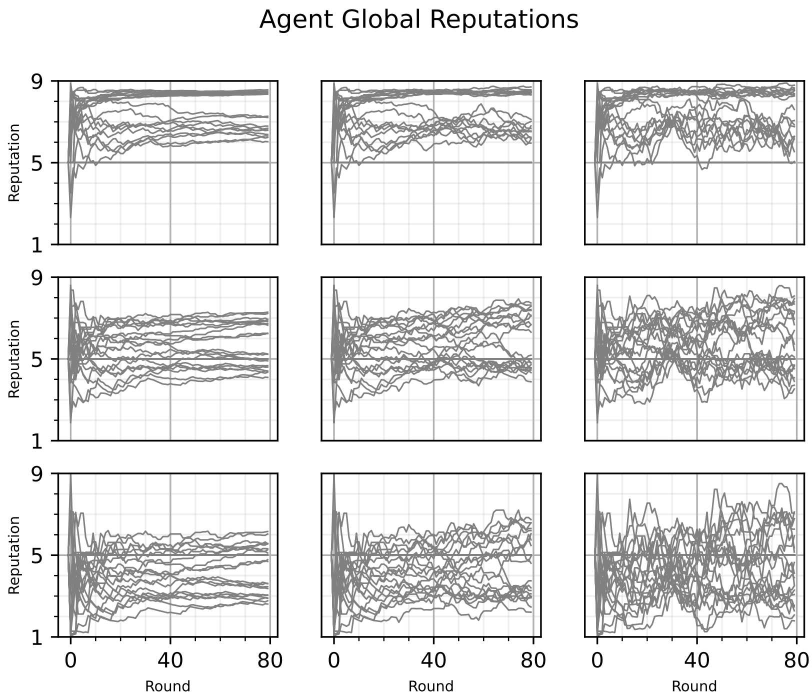


Figure 13: Progression of agent reputations per rounds in experiment A: effect of aging. Aging increases toward the right column, while malicious raters increase toward the bottom row.

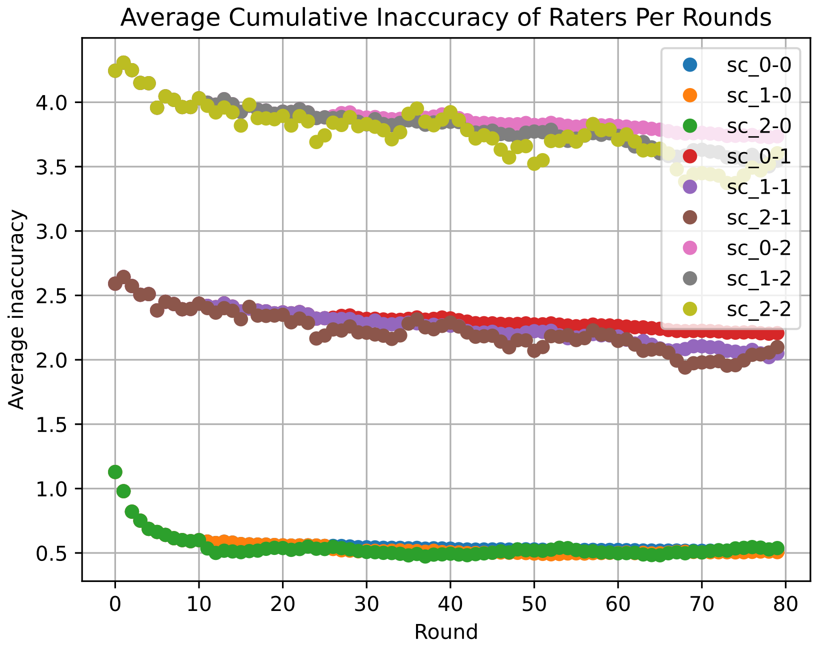


Figure 14: Averaged cumulative rater inaccuracy per rounds in experiment A: effect of aging. In the legend, first index increases with more aging, the second with more malicious agents.

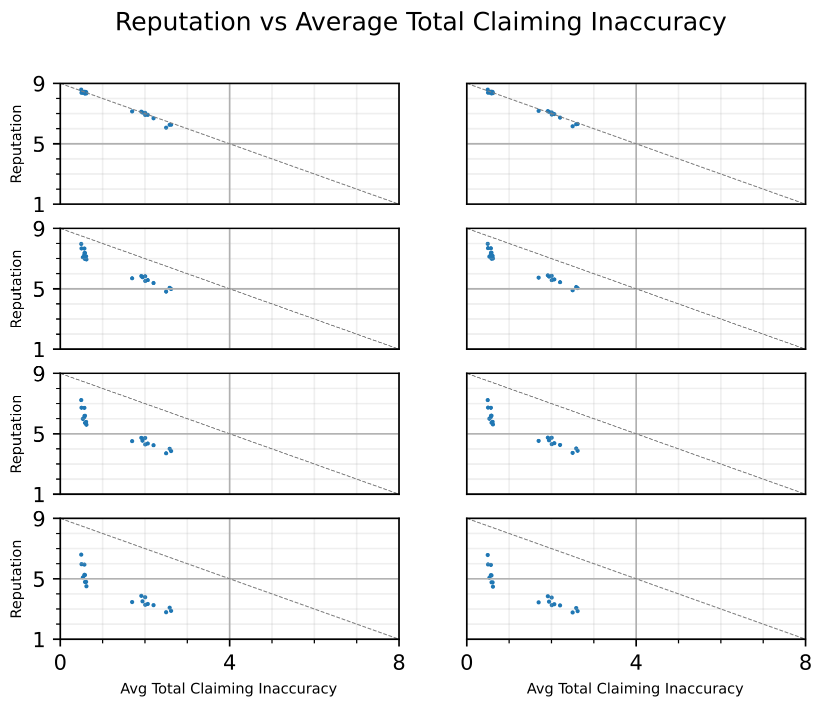


Figure 15: Average Total Claiming Inaccuracy and reputation scatter diagram of experiment B: malicious raters. Malicious raters increase toward the bottom row. The left column is without any improvements, the right side is with weights enabled.

dishonest claimers.

In this experiment, the effect of aging brings no improvement for reputation, in fact it makes reputation scores more spread out for groups. This is observable on Figures 12 and 13. The former shows that agent groups are less and less concentrated with more aging, while Figure 13 shows that reputations become more erratic with more aging. This can be explained by the smaller amount of data available for averaging and other aggregations with aging. Since aging removes old claims and all involved reviews that are older than the configured limit, this data is lost as far as reputation calculation and review averaging goes.

Figure 14 illustrated the rater's inaccuracy. Since raters do not claim, and their rating methods do not need old claims, they are less affected. However, since `HonestRater` agents use the reputation of rated claim authors, the fluctuation of reputation scores also influence their ratings. This is seen as swings in inaccuracy on the aged scenarios.

Experiment B: Malicious Raters

Increasing the number of malicious raters results in a similar effect as observed in the previous experiment, only without the interference of aging. Figure 15 shows that malicious raters push the reputation of all claimers equally down.

Honest claimers are smeared towards their ideal reputation levels, as they form a sort

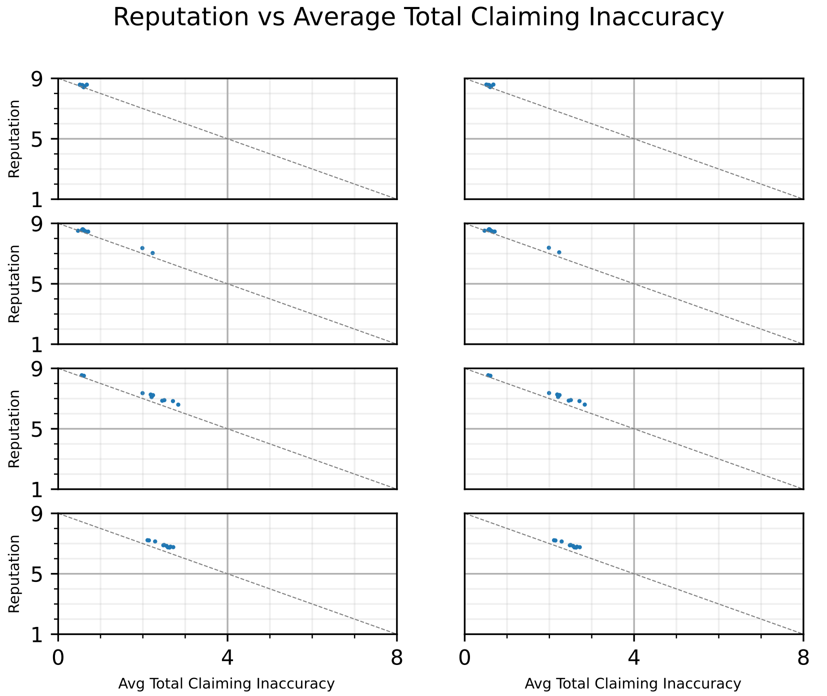


Figure 16: Average Total Claiming Inaccuracy and reputation scatter diagram of experiment C/I: dishonest claimers. The number of dishonest claimers increases downwards. Left column: no improvements, right: weights enabled.

of vertical line. This means their inaccuracy was the same, still they received different reputations. On the bottom row with all the raters malicious, the spread of honest claimer reputations is a little over 2 points.

Because malicious raters never claim, they do not get reputation adjustments and stay on the default of 5 for the whole simulation. This means there is no way to filter their ratings, even with weights and a number of honest raters still in the system.

Experiment C: Dishonest Claimers

C/I results are shown on figures 16 and 17. Dishonest claimers are positioned above their exactly ideal reputation, but the difference is insignificant. Good-intentioned raters are able to keep order even when all claimers are dishonest. Figure 17 shows however, that it takes around 20-30 rounds for reputations to settle. Both figures show no improvements on the left columns and weights on the right side. There is no difference between the two versions.

Experiment C/II removes all raters that are not influenceable, i.e. that are not `InfluencedHonestRater`. It also tests the effect of higher CTAI for honest claimers and raters both. As seen on figures 18, dishonest claimers are able to get an even better reputation by influencing raters with high author reviews. In the right column, honest claimers have more inaccuracy, so they are more spread out horizontally on the total

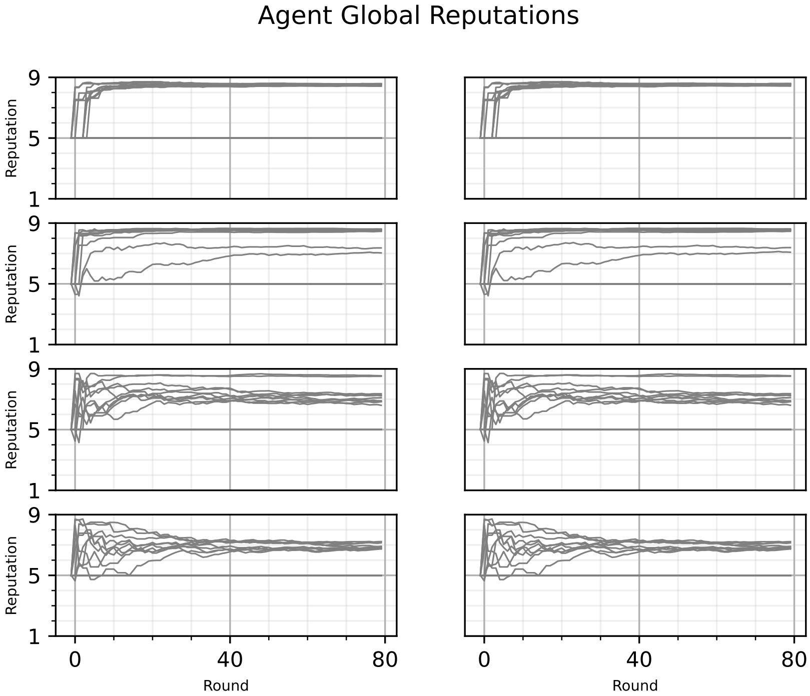


Figure 17: Progression of agent reputations per rounds in experiment C/I: dishonest claimers. The number of dishonest claimers increases downwards. Left column: no improvements, right: weights enabled.

claiming inaccuracy axis.

Rater inaccuracy is also affected by the higher CTAI, shown on Figure 19. As expected, a higher CTAI makes reviews more inaccurate.

C/III simulates dishonest claimers that are dishonest increasingly often. As observed on Figure 20, dishonest claimers slide rightwards on the subfigures, meaning they make more and more dishonest claims. With influenceable raters, the dishonest claimers are able to acquire a better reputation. In fact, the undeserved reputation gains get relatively larger as more and more dishonest claims are made. However, if raters disregard the author's review (right, Figure 20), reputation scores stay on the ideal line regardless of dishonest claims.

Experiment D: Effect of Weights

All agents are both raters and claimers in this experiment, because this is needed for them to have reputation adjustments. D/I increases the number of dishonest agents. Bad-intentioned raters are no longer malicious, instead they rate honest agents dogmatically with a lower score. As expected, honest agent reputations are brought down, seen in Figure 21. Using weights makes no difference here.

D/II makes agents dogmatic and divisive. Honest agents give better ratings to good

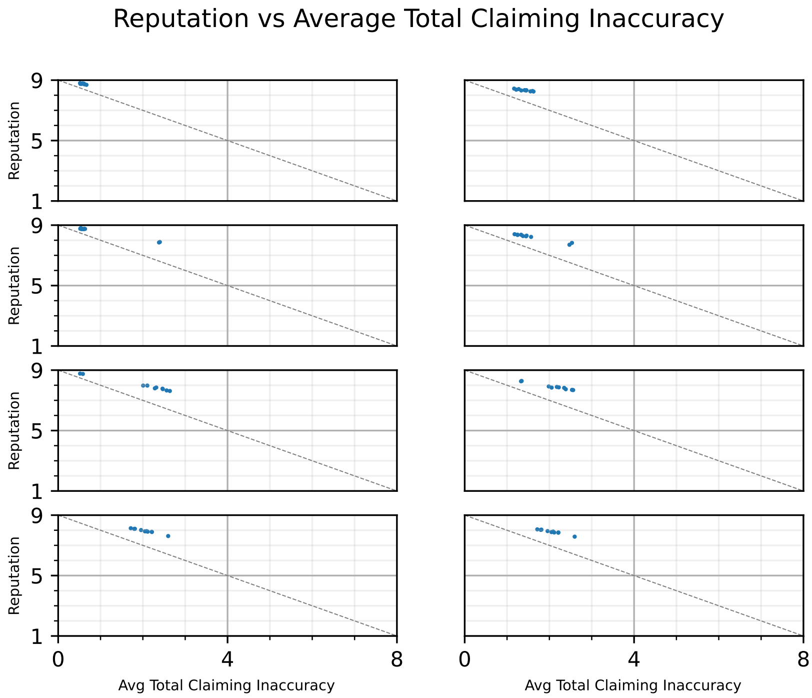


Figure 18: Average Total Claiming Inaccuracy and reputation scatter diagram of experiment C/II: dishonest claimers. The number of dishonest claimers increases downwards. Left column: honest raters and claimers have the default CTAI, right: CTAI is tripled for them.

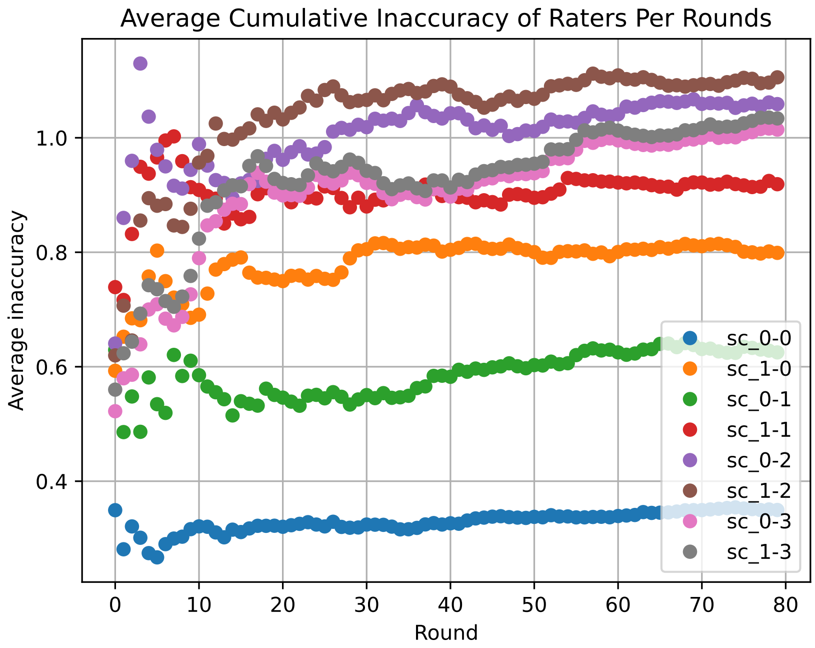


Figure 19: Averaged cumulative rater inaccuracy per rounds in experiment C/II: dishonest claimers. In the legend, first index means lower CTAI if 0, higher if 1, applied only to honest raters and claimers. The second index increases along with the number of dishonest claimers.

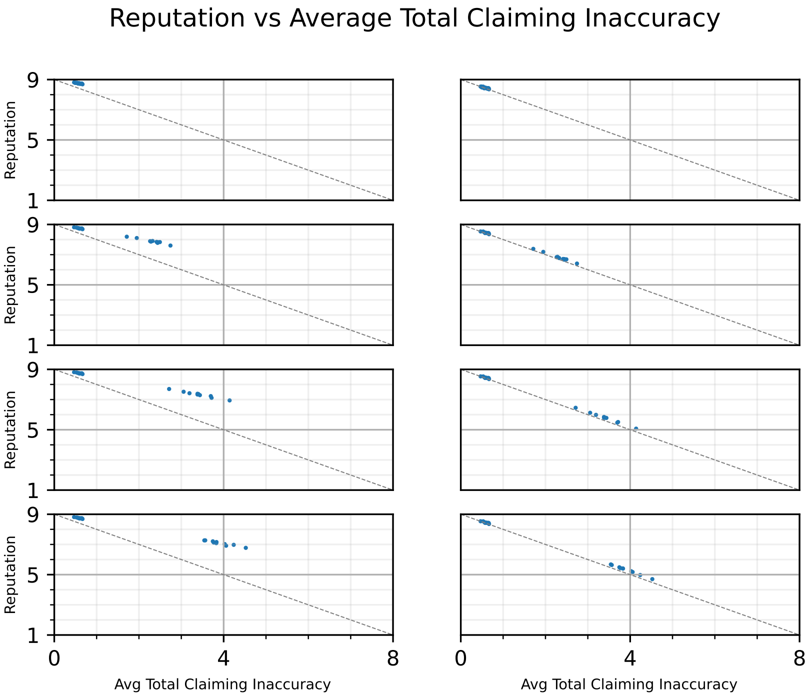


Figure 20: Average Total Claiming Inaccuracy and reputation scatter diagram of experiment C/III: dishonest claimers. Dishonest claimers make exaggerated maximum quality claims more often towards the bottom row. Left column: raters are influenced by author reviews, right: raters rely on their own experience only.

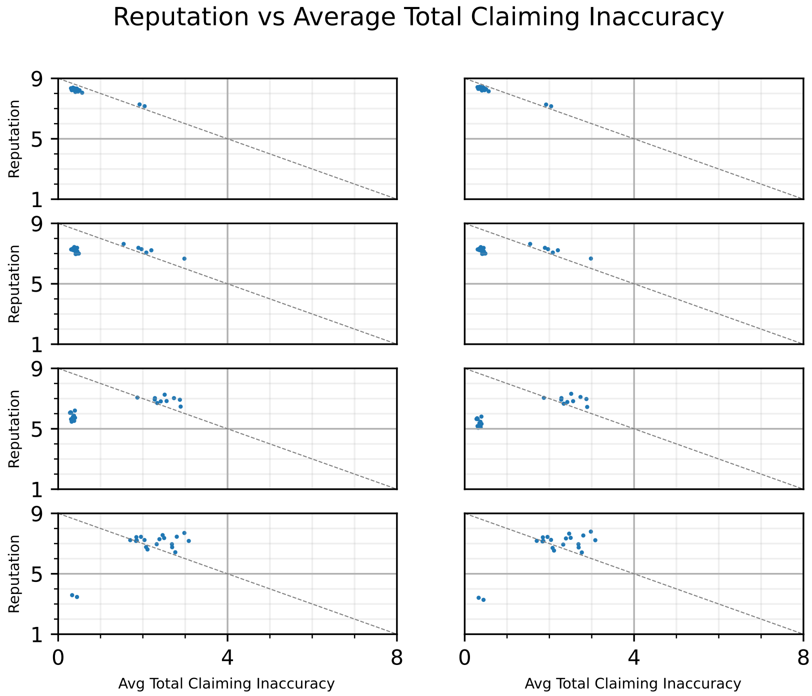


Figure 21: Average Total Claiming Inaccuracy and reputation scatter diagram of experiment D/I: weights. The ratio of dishonest claimer-raters increases downwards. Both columns have aging with limit 30, right side also has weights.

reputation claimers and worse for bad reputation ones. Dishonest agents claim dishonest every time. Figure 22 shows that honest agents are able to recover their good reputation with the help of weights. This is a relatively long process, taking 30-60 rounds (Figure 23). In comparison, D/III results show that the same recovery of honest agent recovery no longer happens when the honest agents are not dogmatic. On Figure 24 all agents fall below the deserved ideal reputation line. Weights pushes reputations closer to ideal, and all agents receive this effect.

Experiment E: Measurement Accuracy

Increasing the measurement inaccuracy (CTAI) of agents is expected to have the effect of more inaccurate ratings, both author and regular. This was partially shown in experiment C/II as well. Here, inaccuracy is varied even more.

Figure 26 shows that rater inaccuracy generally increases with CTAI, as expected. When 10 agents rate, the average inaccuracy is limited, while a single agent rates much more inaccurately. Time makes sense, as the inaccuracies average out for more agents. Similar is observed in Figure 25, where 10 completely inaccurate agents (bottom left) can set the reputations within 30 rounds, while a single agent with the same inaccuracy (bottom right) fails to arrive at the same reputations in 80 rounds. Inaccurate raters can find out the reputation of agents accurately when they are in big numbers.

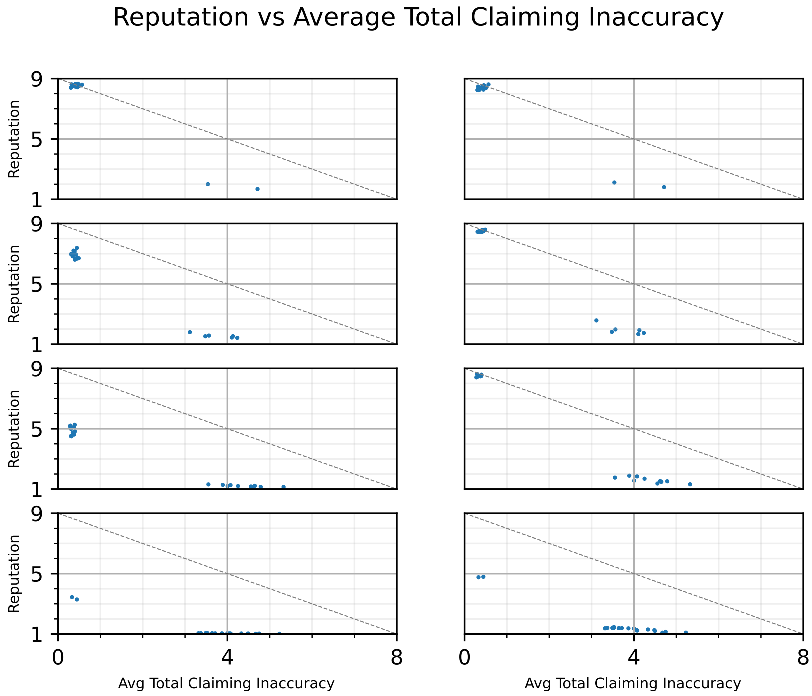


Figure 22: Average Total Claiming Inaccuracy and reputation scatter diagram of experiment D/II: weights. The ratio of dishonest claimer-raters increases downwards. Both columns have aging with limit 30, right side also has weights. Honest raters are dogmatic.

5.4 Discussion

Simulation Framework

The created Pyrepsys framework fulfilled expectations, but not without remarks. Pyrepsys does not currently model agent decision making, for example there is no way for agents to decide who they interact with. This is the basis of measuring agent success as metric, as stressed by the Alpha Testbed covered in Chapter 2. However, such functionality would be possible to implement.

Another weakness, which is also an advantage, is generality. When working within a concrete domain, a more niche evaluation framework can model the reputation environment better, especially when the domain's main differentiator is on the communication protocol or network level. For example, wireless sensor networks use reputation system to decide routing, and routing outcomes feed back to reputation (or trust) scores. The TRMSim-WSN simulator was build specifically for this purpose, to simulate reputation or trust-based routing[56].

Pyrepsys is able to work without specifics of domains, and this is good when needed. So far no universal and cross-domain evaluation frameworks gained widespread acceptance and use, even though researchers introduced multiple previously. Simulating different

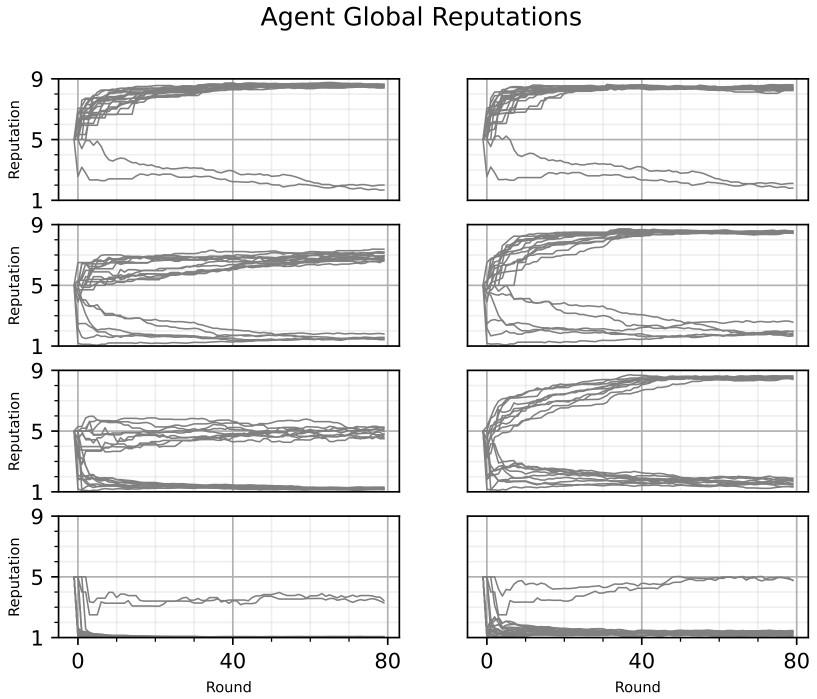


Figure 23: Progression of agent reputations per rounds in experiment D/II: weights. The ratio of dishonest claimer-raters increases downwards. Both columns have aging with limit 30, right side also has weights. Honest agent reputations gradually recover with weights.

kind of environments was possible and produced many of the expected results. The simulator also manages to stay flexible and extendable, so it provides a viable alternative to other previous systems.

Aging

Aging was shown to make the reputation system's memory shorter. While this was not particularly useful in tested experiments with static behaviors, it can have its uses. When agent behaviors or qualities change, a reputation system should recognize this and adjust the agents reputation accordingly. Examples for such change of behavior can be switching to a different claiming method, or the agent gets more accurate or honest over time, or the other way around. This is a realistic, since static behavior is rarely the case, even algorithmic agents can change abruptly with an update for example. Aging's leniency to forget agent history could make a reputation system react faster to changes. When selecting the aging time limit, the goal is to still produce more or less stable aggregations, so that reputation does not become erratic, as happened on Figure 13. At the same time, the system should still allow changes over some time frame. Setting the aging limit needs to be tested for this in each different environment.

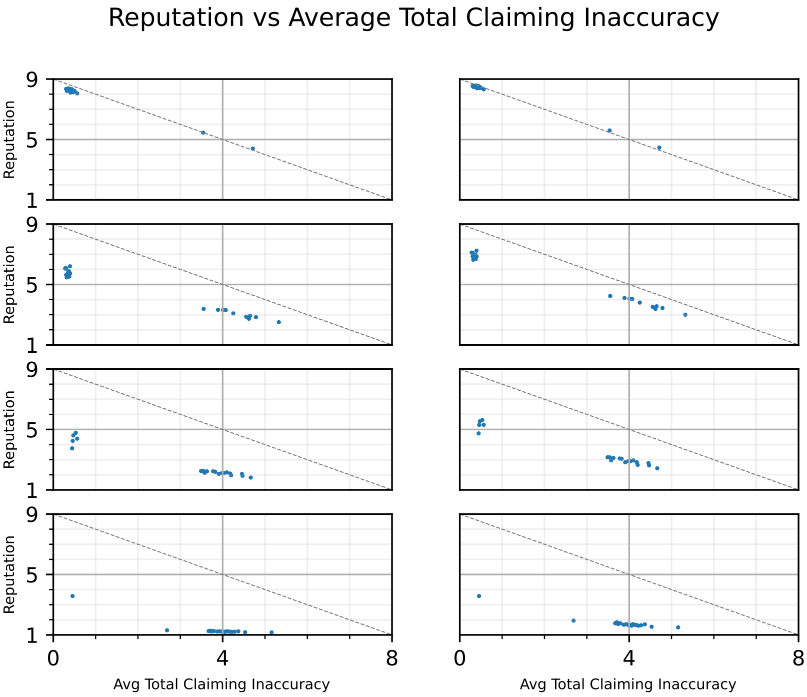


Figure 24: Average Total Claiming Inaccuracy and reputation scatter diagram of experiment D/III: weights. The ratio of dishonest claimer-raters increases downwards. Both columns have aging with limit 30, right side also has weights. Honest agents rate from their own experience.

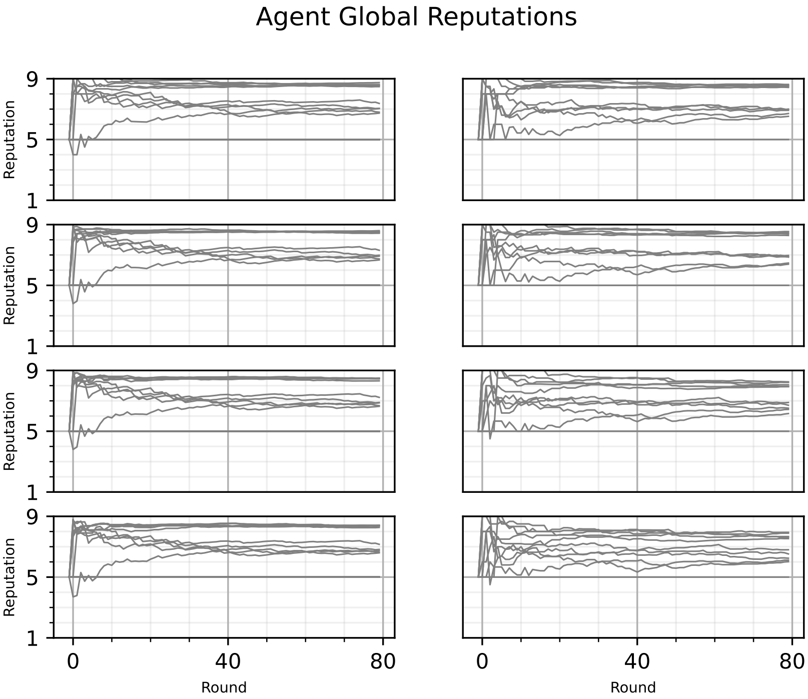


Figure 25: Progression of agent reputations per rounds in experiment E: measurement accuracy. CTAI increases downwards. Left: 10 raters, right: only 1 rater.

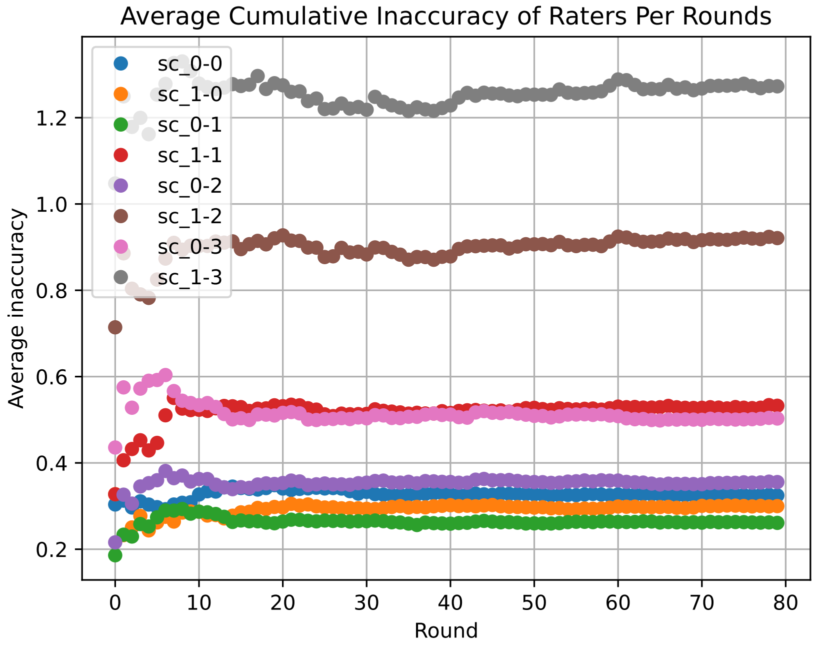


Figure 26: Averaged cumulative rater inaccuracy per rounds in experiment E: measurement accuracy. In the legend, first index means there are 10 raters if 0, only 1 rater if 1. The second index increases along with CTAI. More raters can balance measurement inaccuracy.

Weights

Weighting was shown to be helpful, but only under very specific configuration. The weights implementation simply takes the agent's reputation and gives a proportional weight. Since only claimers get reputation, purely rater agents never adjust their starting reputation, and so nor do they get weights. When these same agents rate, their reviews will mostly be taken equally, because they all have default weights.

This was not the case with an agent population representing an algorithmic environment, where most raters are claimers as well. Most other cases have this problem in the used model.

Either the weights method or the reputation calculation could change to resolve this. If a reputation scheme can give reputation to raters who do not claim, the same simple weights can be used. If the weight calculation is different, non-claiming raters should get weights assigned as well.

Reputation Calculation

`BasedOnAvgDifferenceOfClaimsAndReviewsWithWeight`, the reputation calculation method used, was unable to handle scenarios with a high percentage of raters who never claim. Such strict consumers are often seen in human-used reputation systems, where most users never enter the system as producers. The lack of reputation assignment for consumer-only raters enables malicious or dishonest behavior for them, because there is no basis to filter, weight, etc. their reviews.

This could be remedied with a different reputation method and more elaborate agent behavior strategies. If a method assigns reputation to agents based on their reviews as well, pure raters can no longer evade reputation penalties or rewards.

Alternatively, to get the already used reputation calculation method to work on rater side as well, the model would need some kind of second order rating, where reviews are rated as well. Such systems exist already, for example online markets sometimes ask if a review was helpful or not.

6

Conclusion

We successfully set up a generic reputation system model and provided a Python implementation of it named Pyrepsys. Pyrepsys is the first evaluation framework to support modular reputation improvement methods. We collected a selection of promising techniques for improving reputation system accuracy, and building robustness against attacks and malicious behavior. Two of these were evaluated with Pyrepsys.

In cases were agent behavior never changes, aging was shown to be useless or possibly even counterproductive. Due to forgetting older data, it increases the spreads of aggregations like review averages and reputation calculation.

Weighting worked only in specific circumstances, namely when agents are both claimers and raters at the same time, and rating is dogmatic, meaning honest agents rate better reputationed agents with disproportionately better ratings, and vice versa. Weights did not bring an improvement when most raters never claimed. We attribute this to the used reputation calculation method, which assigns reputation values based on honesty in claiming, so that agents who never claim do not get reputation adjustments.

All in all, accurately modeling reality within a reputation system is hard. The diversity of agents in intent, behavior and ability, and many circumstances of domains need to be accounted for. The difficulty rises even more if the aim is to stay generic.

6.1 Outlook

Many opportunities remain both in the introduced model and implementation, and in trust and reputation research.

The Pyrepsys model and implementation can be continued by

- Evaluating more improvement methods
- Including complex threats and attack methods
- Implementing known reputation calculation methods

- Exploring dynamic agent behavior and agent collaborations

As for trust and reputation systems, researchers have been aiming to create a generic and widely accepted evaluation framework for decades now, with only partial and intermittent successes. Reputation systems are deployed in many different environments, each environment with its own circumstances. Because of this, it is difficult to find an approach that fits all. Nonetheless, as more and more people and devices join networks, the need for reputation systems will increase, necessitating ways to compare different approaches.

A

Scenario Configuration Options

Table A.1 lists top-level scenario parameters. Agent-specific parameters are listed in Table A.2.

Parameter	Type	Function
reputation_strategy	string (classname)	Reputation calculation method
improvement_handlers	list of strings (classnames)	Active improvement techniques
seed	any or null	Seed for random number generation
agent_base_behaviors	agent definition	Agent type templates
agents	agent definition	Concrete agents in the scenario
metrics	list of strings (classnames)	Active metrics
MIN_RATING	float	Bottom of the rating and reputation range
MAX_RATING	float	Top of the rating and reputation range
INITIAL_REPUTATION	float	Starting reputation for agents
SIM_ROUND_MAX	integer > 0	Number of rounds to simulate
MEASURED_CLAIM_RESOLUTION	float	Resolution of measured qualities
REVIEW_RESOLUTION	float	Resolution of published reviews

Table A.1: All top-level scenario configuration options in Pyrepsys.

Parameter	Type	Function
distort_strategy	string (classname)	Distortion strategy
rate_strategy	string (classname)	Rating strategy
claim_range	list of two floats	Min. and max. quality the agent publishes claims with
claim_probability	float $\in [0, 1]$	Chance of claiming in a round
rate_probability	float $\in [0, 1]$	Chance of leaving a review
claim_truth_assessment_inaccuracy	float $\in [0, 1]$	CTAI (measurement inaccuracy)
amount	integer > 0	Create this many concrete agents of the given type
name	string	Name of an agent base behavior.

Table A.2: All agent configuration options in Pyrepsys.

Acronyms

CPS cyber-physical system. 2

CTAI Claim Truth Assessment Inaccuracy. 16, 38, 40, 58, 61, 64–66, 68, 71, 72, 78

ETA estimated time of arrival. 2

IoT internet of things. 2–5

MAS multi-agent system. 9, 10, 12

p2p peer-to-peer. 3, 4, 10, 11, 14

Glossary

agent-exposed representation Refers to storing values in the configured rating range [minimum rating, maximum rating] in Pyrepsys. This is what agents and users see. See internal representation, sec. 4.3.2. 44, 45, 51, 61, 81

artifacts directory Directory in which the logfile and simulation results from metrics are saved. Located in the `simulation_artifacts` directory. Named after the date and time Pyrepsys was invoked.. 54, 55

author review A claimer's quality valuation of its own content. See sec. 3.2.3. 14, 16, 33, 41, 42

branch-off random chain A secondary sequence of random numbers that is thrown away after it is used. Seeded with a random value from the main chain. See sec. 4.4. 38, 40, 48

claim The representation of a content provided by an agent. See sec. 3.2. 29, 33

claim range Claim author reviews, reviews, ground truths, measured qualities and reputations fall within this range. 33

claiming probability A percent likelihood determining whether an agent initiates a claiming process if given an opportunity. Specified in scenarios for each agent. 33

distorted claim quality The value produced by the distort strategy from the measured claim quality. 33

distortion strategy Claimers calculation method of author review values. See sec. 3.6. 33

ground truth The real and hidden objective quality of a content. Unknown to agents. See sec. 3.2.2. 14, 15, 33, 35, 38, 58

internal representation Refers to storing values in the [0,1] range by Pyrepsys. Meant for internal use in data structures. See agent-exposed representation, sec. 4.3.2. 44, 47, 50, 81

main random chain Top-level sequence of random numbers. See sec. 4.4. 33, 35, 47

measured claim quality An agent's approximation of a claim's ground truth. See sec. 3.2.4. 33, 38

rating probability A percent likelihood determining whether an agent leaves a review on a claim if given an opportunity. Specified in scenarios for each agent. 35

rating strategy Agents calculation method of review values. See sec. 3.7. 35

rating span The size of the rating range, calculated as the difference of the maximum and minimum review values. 44, 47

review Represents a quality valuation of a content (claim). Also called regular review. See sec. 3.3. 35

scenario Description of a complete simulation environment for Pyrepsys. Specifies the reputation scheme, improvement methods, agents, possible review values etc. See Section 3.9 for details. 28, 31, 34, 48

Bibliography

[1] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, "The EigenTrust Algorithm for Reputation Management in P2P Networks," *Proceedings of the 12th international conference on World Wide Web*, pp. 640–651, 2003. cited on p. 3, 4

[2] A. Jøsang, "Trust and Reputation Systems," *Foundations of Security Analysis and Design*, vol. IV, pp. 209–245, 2007. cited on p. 3

[3] A. Jøsang, R. Ismail, and C. Boyd, "A Survey of Trust and Reputation Systems for Online Service Provision," p. 44, 2017. cited on p. 3, 4

[4] D. Gambetta, "Can We Trust Trust?," *Trust: Making and Breaking Cooperative Relations*, p. 17, 2000. cited on p. 4

[5] H. A. Kurdi, "HonestPeer: An enhanced EigenTrust algorithm for reputation management in P2P systems," *Journal of King Saud University - Computer and Information Sciences*, vol. 27, pp. 315–322, July 2015. cited on p. 4

[6] M. Keshavarz, M. Gharib, F. Afghah, and J. Ashdown, "UASTrustChain: A Decentralized Blockchain-based Trust Monitoring Framework for Autonomous Unmanned Aerial Systems," *IEEE Access*, pp. 1–1, 2020. cited on p. 4

[7] F. Ahmad, F. Kurugollu, A. Adnane, R. Hussain, and F. Hussain, "MARINE: Man-in-the-Middle Attack Resistant Trust Model in Connected Vehicles," *IEEE Internet of Things Journal*, vol. 7, pp. 3310–3322, Apr. 2020. cited on p. 4

[8] G. Primiero, A. Martorana, and J. Tagliabue, "Simulation of a Trust and Reputation Based Mitigation Protocol for a Black Hole Style Attack on VANETs," in *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, (London), pp. 127–135, IEEE, Apr. 2018. cited on p. 5

[9] D. O. Airehrour, "A Trust-Based Routing Framework for the Internet of Things," p. 306, 2017. cited on p. 5

[10] S. Ganeriwal and M. B. Srivastava, "Reputation-based Framework for High Integrity Sensor Networks," p. 12, 2008. cited on p. 5

[11] A. I. A. Ahmed, S. H. Ab Hamid, A. Gani, S. khan, and M. K. Khan, "Trust and reputation for Internet of Things: Fundamentals, taxonomy, and open research challenges," *Journal of Network and Computer Applications*, vol. 145, p. 102409, Nov. 2019. cited on p. 5

[12] J. Zhang, M. Sensoy, and R. Cohen, "A Detailed Comparison of Probabilistic Approaches for Coping with Unfair Ratings in Trust and Reputation Systems," in *2008 Sixth Annual Conference on Privacy, Security and Trust*, (Fredericton, Canada), pp. 189–200, IEEE, Oct. 2008. cited on p. 5, 6

[13] A. Whitby, A. Jøsang, and J. Indulska, "Filtering Out Unfair Ratings in Bayesian Reputation Systems," in *Proceedings of the Workshop on Trust in Agent So- cieties, at the Autonomous Agents and Multi Agent Systems Conference*, p. 12, 2014. cited on p. 5, 6

[14] S. Thakur, "A reputation management mechanism that incorporates accountability in online ratings," *Electronic Commerce Research*, vol. 19, pp. 23–57, Mar. 2019. cited on p. 6, 9

[15] W. T. L. Teacy, J. Patel, N. R. Jennings, and M. Luck, "TRAVOS: Trust and Reputation in the Context of Inaccurate Information Sources," *Autonomous Agents and Multi-Agent Systems*, vol. 12, pp. 183–198, Mar. 2006. cited on p. 6

[16] A. Josang and J. Haller, "Dirichlet Reputation Systems," in *The Second International Conference on Availability, Reliability and Security (ARES'07)*, (Vienna, Austria), pp. 112–119, IEEE, 2007. cited on p. 7

[17] B. Yu and M. P. Singh, "Detecting Deception in Reputation Management," p. 8, 2003. cited on p. 7, 24

[18] J. Zhang and R. Cohen, "A Personalized Approach to Address Unfair Ratings in Multiagent Reputation Systems," p. 10, 2006. cited on p. 7

[19] N. Iltaf, A. Ghafoor, and U. Zia, "A mechanism for detecting dishonest recommendation in indirect trust computation," *EURASIP Journal on Wireless Communications and Networking*, vol. 2013, p. 189, Dec. 2013. cited on p. 8

[20] Zakirullah, M. H. Islam, and A. A. Khan, "Detection of dishonest trust recommendations in mobile ad hoc networks," in *Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, (Hefei, China), pp. 1–7, IEEE, July 2014. cited on p. 8

[21] C. Dellarocas, "Immunizing online reputation reporting systems against unfair ratings and discriminatory behavior," in *Proceedings of the 2nd ACM conference on Electronic commerce - EC '00*, (Minneapolis, Minnesota, United States), pp. 150–157, ACM Press, 2000. cited on p. 8

[22] E. Zupancic and D. Trcek, "QADE: A Novel Trust and Reputation Model for Handling False Trust Values in E-Commerce Environments with Subjectivity Consideration," *Technological and Economic Development of Economy*, vol. 23, pp. 81–110, Jan. 2015. cited on p. 8

[23] M. Rezvani and M. Rezvani, “A Randomized Reputation System in the Presence of Unfair Ratings,” *ACM Transactions on Management Information Systems*, vol. 11, pp. 1–16, Apr. 2020. cited on p. 8

[24] A. Baby, A. Kumaresan, and K. Vijayakumar, “A Secure Online Reputation Defense System from Unfair Ratings using Anomaly Detections,” *International Journal of Computer Applications*, vol. 93, pp. 17–21, May 2014. cited on p. 8

[25] F. Azzedin, “Identifying Honest Recommenders in Reputation Systems,” p. 7, 2010. cited on p. 8

[26] C. Wan, J. Zhang, and A. A. Irissappane, “A Context-Aware Framework for Detecting Unfair Ratings in an Unknown Real Environment,” in *2012 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, (Macau, China), pp. 563–567, IEEE, Dec. 2012. cited on p. 9

[27] R. Jurca and B. Faltings, “Minimum payments that reward honest reputation feedback,” in *Proceedings of the 7th ACM conference on Electronic commerce - EC ’06*, (Ann Arbor, Michigan, USA), pp. 190–199, ACM Press, 2006. cited on p. 9

[28] S. Liu, C. Miao, Y. Liu, H. Fang, H. Yu, J. Zhang, Y. Chai, and C. Leung, “A Reputation Revision Mechanism to Mitigate the Negative Effects of Misreported Ratings,” in *Proceedings of the 17th International Conference on Electronic Commerce 2015 - ICEC ’15*, (Seoul, Republic of Korea), pp. 1–8, ACM Press, 2015. cited on p. 9

[29] J.-Y. L. Boudec and S. Buchegger, “A Robust Reputation System for Mobile Ad-hoc Networks,” tech. rep., 2003. cited on p. 9

[30] T. D. Huynh, N. R. Jennings, and N. R. Shadbolt, “On Handling Inaccurate Witness Reports,” p. 15, 2005. cited on p. 9

[31] A. Khoshkbarchi and H. R. Shahriari, “Coping with unfair ratings in reputation systems based on learning approach,” *Enterprise Information Systems*, vol. 11, pp. 1481–1499, Nov. 2017. cited on p. 9

[32] Y. Wang and J. Vassileva, “Bayesian network-based trust model,” in *Proceedings IEEE/WIC International Conference on Web Intelligence (WI 2003)*, (Halifax, NS, Canada), pp. 372–378, IEEE Comput. Soc, 2003. cited on p. 9

[33] A. Yazidi, B. J. Oommen, and M. Goodwin, “On Solving the Problem of Identifying Unreliable Sensors Without a Knowledge of the Ground Truth: The Case of Stochastic Environments,” *IEEE Transactions on Cybernetics*, vol. 47, pp. 1604–1617, July 2017. cited on p. 9

[34] A. Jøsang, G. Guo, M. S. Pini, F. Santini, and Y. Xu, “Combining Recommender and Reputation Systems to Produce Better Online Advice,” in *Modeling Decisions for Artificial Intelligence* (D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, V. Torra, Y. Narukawa, G. Navarro-Arribas, and D. Megías, eds.), vol. 8234, pp. 126–138, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Series Title: Lecture Notes in Computer Science. cited on p. 9

[35] D. Margaris and C. Vassilakis, “Improving Collaborative Filtering’s Rating Prediction Quality by Considering Shifts in Rating Practices,” in *2017 IEEE 19th Conference on Business Informatics (CBI)*, (Thessaloniki, Greece), pp. 158–166, IEEE, July 2017. cited on p. 9

[36] D. Margaris and C. Vassilakis, “Improving Collaborative Filtering’s Rating Prediction Accuracy by Considering Users’ Rating Variability,” in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, (Athens), pp. 1022–1027, IEEE, Aug. 2018. cited on p. 9

[37] V. Agate, A. De Paola, G. Lo Re, and M. Morana, “A Platform for the Evaluation of Distributed Reputation Algorithms,” in *2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, (Madrid), pp. 1–8, IEEE, Oct. 2018. cited on p. 10, 11

[38] V. Agate, *Reputation Management Algorithms in Distributed Applications*. PhD thesis, 2020. cited on p. 10, 11

[39] K. K. Fullam, T. B. Klos, G. Muller, J. Sabater, A. Schlosser, K. S. Barber, J. S. Rosenschein, L. Vercouter, and M. Voss, “A Specification of the Agent Reputation and Trust (ART) Testbed: Experimentation and Competition for Trust in Agent Societies,” p. 7, 2005. cited on p. 9, 10

[40] D. Jelenc, R. Hermoso, J. Sabater-Mir, and D. Trček, “Decision making matters: A better way to evaluate trust models,” p. 18, 2013. cited on p. 10, 11

[41] P. Chandrasekaran and B. Esfandiari, “Toward a testbed for evaluating computational trust models: experiments and analysis,” *Journal of Trust Management*, vol. 2, p. 8, Dec. 2015. cited on p. 10

[42] A. Salehi-Abari and T. White, “DART: A distributed analysis of reputation and trust framework,” *Computational Intelligence*, vol. 28, pp. 642–682, Nov. 2012. cited on p. 10

[43] T. P. Nguyen and B. J. d'Auriol, "EStarMom: Extendable Simulator for Trust and Reputation Management in Online Marketplaces," p. 10, 2014. cited on p. 10

[44] A. A. Adamopoulou and A. L. Symeonidis, "A simulation testbed for analyzing trust and reputation mechanisms in unreliable online markets," *Electronic Commerce Research and Applications*, vol. 13, pp. 368–386, Sept. 2014. cited on p. 10, 11

[45] A. A. Irissappane, S. Jiang, and J. Zhang, "Towards a Comprehensive Testbed to Evaluate the Robustness of Reputation Systems against Unfair Rating Attacks," p. 12, 2012. cited on p. 10

[46] M. Janiszewski, "Towards an Evaluation Model of Trust and Reputation Management Systems," *International Journal of Electronics and Telecommunications*, vol. 63, pp. 411–416, Nov. 2017. cited on p. 10, 12

[47] Z. Liang and W. Shi, "Analysis of ratings on trust inference in open environments," *Performance Evaluation*, vol. 65, pp. 99–128, Feb. 2008. cited on p. 10

[48] C. J. Hazard and M. P. Singh, "Macau: A Basis for Evaluating Reputation Systems," p. 7, 2013. cited on p. 10

[49] A. Celestini, R. De Nicola, and F. Tiezzi, "Network-Aware Evaluation Environment for Reputation Systems," in *Trust Management VII* (C. Fernández-Gago, F. Martinelli, S. Pearson, and I. Agudo, eds.), vol. 401, pp. 231–238, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Series Title: IFIP Advances in Information and Communication Technology. cited on p. 10

[50] A. G. West, S. Kannan, I. Lee, and O. Sokolsky, "An Evaluation Framework for Reputation Management Systems," in *Trust Modeling and Management in Digital Environments: From Social Concept to System Development* (Z. Yan, ed.), IGI Global, 2010. cited on p. 10

[51] A. Schlosser, M. Voss, and L. Bruckner, "Comparing and Evaluating Metrics for Reputation Systems by Simulation," p. 16, 2004. cited on p. 10

[52] G. Suryanarayana and R. N. Taylor, "SIFT: A Simulation Framework for Analyzing Decentralized Reputation-based Trust Models," tech. rep., 2007. cited on p. 10

[53] Y. Zhang, W. Wang, and S. Lü, "Simulating Trust Overlay in P2P Networks," p. 8, 2007. cited on p. 10

[54] R. Kerr and R. Cohen, "TREET: the Trust and Reputation Experimentation and Evaluation Testbed," p. 20, 2010. cited on p. 9, 10

[55] M. Janiszewski, "TRM-EAT - a New Tool for Reliability Evaluation of Trust and Reputation Management Systems in Mobile Environments," in *2020 20th*

IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), (Melbourne, Australia), pp. 718–727, IEEE, May 2020. cited on p. 10

[56] F. G. Mármol and G. M. Pérez, "TRMSim-WSN, Trust and Reputation Models Simulator for Wireless Sensor Networks," p. 5, 2009. cited on p. 10, 11, 69

[57] M. Youssef, E.-N. Abdeslam, and D. Mohamed, "Towards a Framework for the Analysis and Evaluation of Computational Trust Models in Multi-agent Systems," *Journal of Software*, vol. 12, no. 11, p. 14, 2017. cited on p. 10, 12

[58] K. Hoffman, D. Zage, and C. Nita-Rotaru, "A Survey of Attack and Defense Techniques for Reputation Systems," *ACM Computing Surveys*, p. 34, 2009. cited on p. 24

[59] G. Kalton, J. Roberts, and D. Holt, "The effects of offering a middle response option with opinion questions," *Journal of the Royal Statistical Society: Series D (The Statistician)*, vol. 29, no. 1, pp. 65–78, 1980. cited on p. 24

[60] J. Lee, "Outlier Aversion in Evaluating Performance: Evidence from Figure Skating," *IZA Discussion Papers*, no. 1257, p. 41, 2004. cited on p. 24

[61] V. Schoenmueller, O. Netzer, and F. Stahl, "The Extreme Distribution of Online Reviews: Prevalence, Drivers and Implications," *SSRN Electronic Journal*, 2018. cited on p. 25