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

TODO rename

Márton Donát Nagy

Master's Thesis

TODO rename

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

Co-Advisor Corinna Coadvisor

Author Márton Donát Nagy
Musterstr. 42
42424 Musterstadt

Submitted on July 20, 2021

Declaration of Authorship

I, Márton Donát Nagy, declare that this thesis titled "TODO rename" 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

This thesis is about ... This thesis shows that ...

Contents

1

Introduction

TECHNOLOGY is on the move and this topic is important because it will change the world.

1.1 Problem Statement

As a long term goal we would like to have ... The problem is that ... still does not work. So we will investigate the questions

- whether A
- or whether B

1.2 Scenario

The scenario and assumptions and the background of this thesis. We assume ..., we will focus on ... and we will exclude ...
on a more broader sense, the work is intended to be general enough that it could be applied for other iot/cps type applications as well

1.3 Task Definition

We will do

- try A
- try B

1.4 Structure of This Document

First, ...

1.5 Terminology

describe user, peer, agent, principal, node etc namings in research used according to domain of the paper
network
underlying value, trueness, ground truth,
reputation score, estimate, global rating
rating, opinion, review
Content, transaction, service, product, file, claim
advisor, witness, second hand reputation
internal vs agent-exposed: in the program, internal is how reputations, reviews, claims etc are stored and refers to [0,1] real agent-exposed is how the agent sees everything, typically [1,9] and discrete integers in the code _i vs _ae denotes which type a variable stores
score: review value, claim value, reputation etc. referred generally

2

State of the Art

This chapter gives an overview of ...

2.1 Domains of Trust and Reputation Research

discuss fields like WSN, p2p filesharing, MAS, mobile sensor systems, flying ad hoc networks, vehicular ad hoc networks, ecommerce, recommendation systems online stuff like imdb or hotel ratings....

2.2 Evaluation and Comparison

2.2.1 Theoretical Categorization

2.2.2 Simulation Frameworks

[1] does an ad hoc comparison of accuracy improvement / unfairness filtering methods. Maybe mention

Asd Example

2.3 Improvement Techniques for Reputation Systems

Various papers propose methods to make reputation systems more robust against exploitations and improve the accuracy of their reputation calculation. Works with this goal are most commonly in the e-commerce and online recommendation systems domains. A smaller amount also explore other domains.

2.3.1 Categorization

accuracy (better function) vs attack countering they can overlap
categories based on domain mentioned, but not the main driver of categorization here, because looking for general ideas to implement in IoT etc as cross domain better if domain dependent or independent
based on general idea primary categories for this collection
endo vs exogenous mention that it gained some traction in research Whitby et al. group methods to identify and remove unfair ratings into two categories: endogenous and exogenous approaches [2]. 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.
Endogenous could be expanded or generalized to say only the ratings, reviews, feedbacks, including their value and accompanying text, any data and metadata are analyzed.
[1] uses apart from the endo-exo categorization two other categories: 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 differentiated between 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 that 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. (E.g. this seller typically gets 4..5 so this 1-rating is suspicious) In a sense, opinions are isolated between agents. in global methods, advisor (interchangeable with rating, feedback, review etc...) trustworthiness is judged using ratings on all the agents in the system.
Although defined for e-commerce advisor-opinion scenario, these categories can be understood in a more general sense. In this case, sellers are to be understood as agents providing content (data, service, product, sensor reading...) and receiving a rating. Vice versa, buyers are receiving the content and providing the rating. Advisor refers to any third party contributing to the final aggregated reputation (rating, trust level) of a potential transaction partner. Opinion is any rating, review or feedback in a more broader sense.
The public-private and local-global distinction can be understood as two sides of the

transaction. In public-private, the rater is judging,
They also identify 4 capabilities a complete improvement approach should have: -majority: work if majority is unfair / malicious -flooding: function under sudden onset of ratings in short time -lack of experience: still work when agents don't have any connections or private XP -varying: agent behavior change should not pose a problem
reactive or proactive. The reactive solutions intend to identify the unfair feedbacks and the proactive solutions propose incentive to the buyers to encourage them to report fair feedbacks [3]
Accuracy of ratings is a crucial part of any reputation system. This is true whether transactions or directly the users are rated. Many attacks exploit exactly this vulnerable point of TRs. Slandering, promoting and other attack strategies, see attacks chapter...
Interpreting accuracy of ratings is difficult in systems where people rate based on their subjective experiences, or where there is a degree of uncertainty. E.g. e-commerce rep systems Since ratings in real-life situations are not accurately computable analogous term is honesty of users. some researchers (cite?) use a scale based on honesty to categorize users from purely malicious to purely honest. In this case this refers to given feedback, as other dimensions of "inhonesty" is also possible, like malice in transaction content. There was a paper that categorized malice into these two ways, cite mayb. Also this discussion mayb not here.
Accuracy's further interpretation is a sort of "trueness" value. This implies that an objectively true and absolutely accurate rating of a user's experience exists. This is then different from the feedback the user gives into the system. The user may or may not be aware of this "true rating." If not aware, maybe bias, subjectively very unpleasant experience. Some call this inherent bias. E.g. e-com the product is perfect but shipping time was bad and gives a 1-star rating. (There is a philosophycal aspect to this, as this is saying "you experienced this a 1/5 but we say it really was a 4/5" and what right or objective measure exists to permit this.) If the user is aware of the "true quality," giving a doctored rating out of malicious intent is still a possibility. In any case, the result is that the given rating differs from a hidden, "objectively true" rating. This situation is called an "unfair" rating by Josang.
Some other metrics measuring accuracy exist in evaluation frameworks. See chapter metrics.
(continuation of above) machine populated reputation systems where ratings are given by algorithms evaluating data, are more fitting to calculations of numerical accuracy

2.3.2 Statistical Reputation Systems

The Whitby et al. work with this line of thought in their paper in [2]. they extend the BRS from a previous paper They argue that unfair and fair ratings have different statistical patterns. Consequently, it is possible to identify and filter unfair ratings with

statistical methods.

TRAVOS is a bayesian trust and reputation method based on the beta probability distribution [4]. 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.

TRAVOS and the Beta Reputation System are very similar. They are both bayesian statistical methods using the beta probability distribution. However, the Beta Reputation System does not differentiate between direct observarions and advisor recommendations. Additionally, the two methods handle inaccurate (unfair, 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.

There is the question of what to do with opinions caught in the filter. Completely disregard, lower weight, or temporarily disregard...

Majority opinion methods rely on the assumption that the majority of the users are honest and accurate.

The Dirichlet Reputation System is worth mentioning as a multinomial generalisation of the Beta Reputation System [5]. 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 similar concepts to 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 Personalized Approach introduces in [6] ... in e-com, private opinion and global reputation of advisor both used, agents can weight these two, has aging, private: agent take opinion giver, see how they rate commonly rated fourth parties, public reputation based on how consistent the advisor's rating of other parties is

[7] Weighted Majority Algorithm advisors given a weight to sum their opinions, if incorrect based on future transaction, the weight is decreased, constantly wrong advisors are thus filtered eventually

The authors of [6] and [1] take a survey of previous improvement methods, introduce a new method (personalized approach) and compare them, including experimental

measurements. The methods compared are the Beta Reputation System, TRAVOS, Personalized Approach, Bayesian Network Approach and Weighted Majority Algorithm.
[8] "based on the assumption that a dishonest recommendation is one that is inconsistent with other recommendations and has a low probability of occurrence in the recommendation set. Based on this assumption, a new dissimilarity function for detecting deviations in a recommendation set is defined"
[9] in MANETs, dissimilarity function based
[10] uses randomly selected samples of all reviews to calculate global reputation. (e-com)
[11] uses a five-step process to make the reputation system more secure against unfair ratings: Evaluation based filtering, Time domain unfair rating detector, suspicious user correlation analysis, trust analysis based on Dempster-Shafer theory and malicious user identification and reputation recovery.
[12] also surveys methods against unfair ratings. Considered approaches are filtering based on majority opinion (Whitby et al), entropy-based filtering, reputation trees, (agents reputation is their rating's weight) controlled anonymity and agent clustering, clustering ratings and checking IP address, using trust rating in MANETs and WMA and belief function.
[13]
"Transactions are evaluated by both the source peer and the target peer in [9]. A source peer's feedback is considered consistent if it agrees with the target peer's self-evaluation. Assuming most of the peers are trustworthy and honest, most inconsistencies would be in the case that the source peer reports a trustworthy transaction as untrustworthy (badmouthing) or an untrustworthy transaction as trustworthy (collusion to boost the target peer's reputation). Thus, a source peer is suspected to be providing false feedbacks if the proportion of inconsistent feedbacks exceeds certain threshold, T. Feedbacks from inconsistent peers are no longer accepted. " WARNING, this is a quote of consistency based
"The reputation system in [15] uses a measurement called credibility factor. The credibility factor increases if a recommender provides a recommendation that matches the actual result of the transaction. From the credibility, discredibility factor can be derived. A recommender whose discredibility factor is higher than its credibility factor will be filtered out." WARNING quote of credibility based
[14] for e-com, introduces two extension ideas to TRs to combat unfair ratings: controlled anonymity to avoid unfairly low ratings and negative discrimination cluster filtering to reduce the effect of unfairly high ratings and positive discrimination
[15][16] work with the idea of normalizing the rating scale each agent is typically using. This approach comes from collaborative filtering systems.
[17] "finds similar agents and consider them as reputable source of information. Further, the agents compare their trust values with trust values of similar agents"
explicit inclusion of private experience as a system component possibility of maintaining

a trust network and getting recommendations (opinions) from them as advisors
both of these are done in a system primarily used by men. e.g. in e-commerce, once the reliability and quality of a provider is known through first-hand experience, reviews and ratings are not scrutinized as they would be by first-time buyers. Social proof mechanisms like getting recommendations and opinions from friends or trusted contacts is also very common without it needing to be part of any reputation system.
This is different in reputation systems made for and used by algorithmic agents. whether decisions are made by other machine agents or the system (algorithm) itself like in a traffic management scenario, it will only ever consider factors it was programmed to. There is a case for explicitly integrating these human mechanisms or similar methods inspired by them. The potential upside is better reputation estimate accuracy, or at the least better decision outcomes. On the other hand, the increased complexity provides new attack vectors and thus potentially enables new exploits.
Mechanisms like a list of trusted "friends," trust networks, third party advisors providing private opinions come from a trust-centric approach to the decision making problem. Three separate categories give themselves from this distinction: pure reputation systems, reputation systems infused with trust components and pure trust systems. Only the first two is of interest within the scope of this thesis.
With this, the categorization of local-global and public-private is also easier to define in a broader context.
Provide incentives to prevent or mitigate unfair ratings: [3] [18]
[19] recovers the temporary damage of sellers who were given a lower rating by a buyer by mistake, and the buyer willingly corrects it. The seller's reputation is temporarily inflated to revert reputation damage.
[20] aims to improve unfair rating detection by analyzing selected factors of the environment the detection is deployed in and selecting the most suitable approach from a multitude of methods.
learning approaches: [21] [22] Bayesian Network-Based Trust Model, p2p, reinforcement learning [23]
approaches combining first hand trust observations with reputation in some way: [24] for manets, everyone maintains a first-hand reputation (i.e. 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 re-evaluation to enable reputation redemption. [25]
regarding recommender systems: [26]

2.4 Attacks and Exploits

3

Approach

3.1 Improving Reputation Estimates

3.1.1 Categorization

other cateogry: what is rated
agents content links could call this rating dimension aka who/what is rated exactly can represent as graph (vertices, edges, directed boxes as content) discuss special cases, e.g. in e-com, rating the very same physical products between different sellers, an extra rating dimension is neede for service quality (packaging, mayb its white/gray label, shipping time, customer service interaction's quality etc... this is different than the product rating and also diff from the agent rating itself also accuracy rating

3.1.2 next todo

there is a case to be made about applying improvement techniques depending on the environment of the reputation system. e.g.
rating aging makes sense only if behavior are known to change over time. if for sure constant (like machines which receive no updates whatsoever) then it has no point
frequency of feedback rating given after transaction.. means only something maybe if its not mandated, which in a fully machine network might as well just be
outlier filtering makes sense only if you know that significant (extreme) differences in transaction experiences are unlikely to the point that such claim becomes suspicious. Once highly subjective experiences come into play, it is wrong to say a nonmajority opinion is unfair. Or in a highly dynamic agent population, where behavior changes often and rapidly, extremity filtering is contraproductive since it introduces reputation lag.
differenciate based on how many ratings contributed to a final reputation score, i.e. confidence. like if a new entrant gets a default of 4, and another has 4 based on a 1000

ratings, this difference should be noticeable and agents should be able to make decisions based on that

3.1.3 Outlier Filtering based on Majority Opinion

3.1.4 Aging Ratings

3.1.5 Normalizing Rating Range

3.1.6 Weighing based on Rater Reputation

3.1.7 Anonymous ratings

outlier filtering based on something: statistical, entropy, dissimilarity function, similarity dis/similarity is also a kind of stereotype-based TR, also collaborative filtering ties in here normalizing rating range weighing based on rater reputation anonymous ratings adding local trust use random subsets to calculate reputation Weighted Majority Algorithm simplifying rating space (some believe that +- or only + or - helps eliminate attacks) feedback consistency-like solution – in intersection, car estimates ETA, later rates its own ETA, and see if others rate the same ETA consistently – inspired by one menitoned in [13]
pair ETAs with uncertainty (generally: stake) if I can say along with the claim how sure i am in the claim or maybe once when I commission a car, I can say how accurate the sensor reading is, so this accuracy rating remains static for a long time then if this uncertainty is larger, I have more room to manipulate the ETAs within plausible deniability, but on the other hand the other cars can say my claim is more worthless because I have a large uncertainty range..
also in this part: author reviews

3.2 Pyrepsys Evaluation Framework

check how to implement these can use existing eval frameworks? or need own one in python if latter, what components a diagram can be good also for the presentation
The Pyrepsys Reputation System was designed in order to allow simulation, comparison and evaluation of selected reputation schemes with various environment conditions. This chapter gives a conceptual introduction for Pyrepsys, including main design goals, simulation flow and core features. For a description of the concrete implementation of Pyrepsys, see Chapter 4.
Overall requirements and unique approaches warranted the inception of a new reputation simulation framework. These are among others:

- domain-independence

- the use of one or more reputation improvement methods
- ability to represent claims
- possibility of second-order rating schemes like stakes
- full simulation of agents
- flexible review values (to support custom integer ranges)
- extendability and ease of modification for fast prototyping

3.2.1 Design Goals

ASD

Flexibility and extendability 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.

3.2.2 Scenarios

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

- 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 organizing combinations of simulation parameters and comparing 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.
In the Pyrepsys implementation, it is possible to describe scenarios without a full specification. This means not all needed parameters are listed, just some selected ones. In this case, the missing parameters are taken from a default scenario configuration. This is useful when the effect of one or more parameter changes are needed, since those

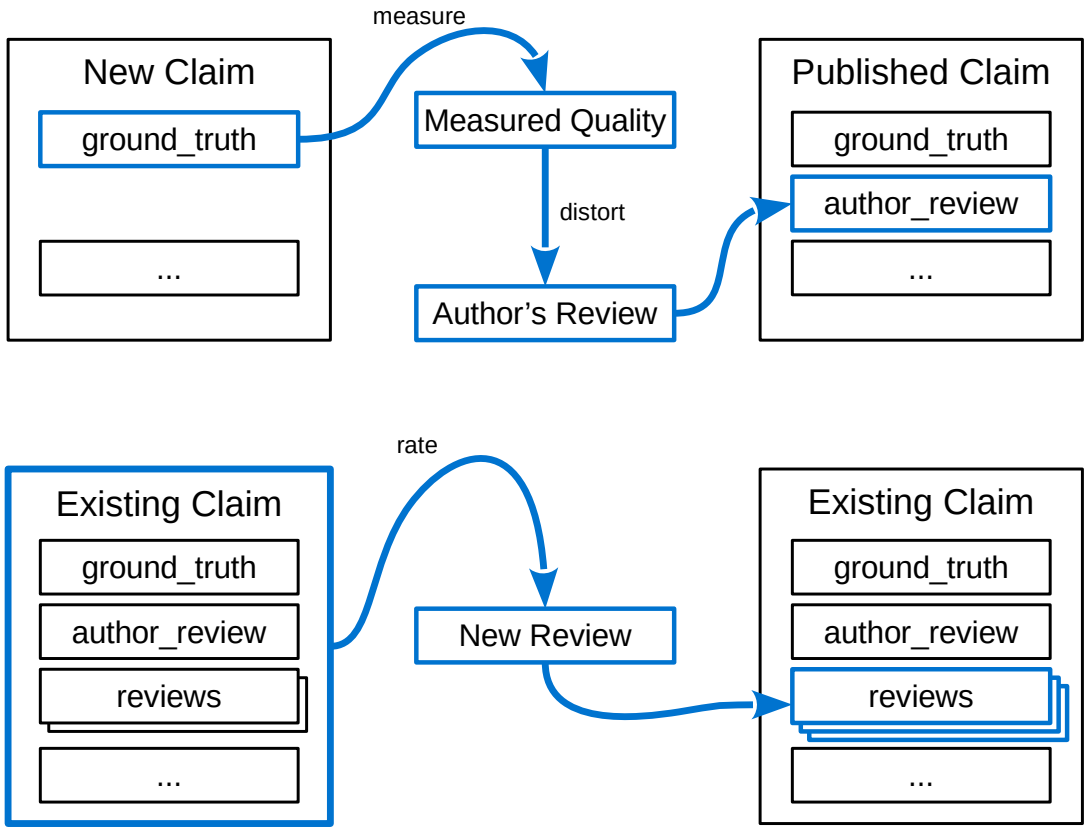

Figure 1: Schematic representation of claiming (above) and rating (below) processes with the involved claims and reviews. *Claiming* (above) spawns a new claim without any reviews, containing a random ground truth. The claimer agent measures this hidden quality and distorts it, resulting in a value representing the clamer's opinion of his own claim. This is attached as an author review to the claim to be published. *Rating* (below) takes an existing published claim. The rater agent takes all data relevant to the claim and produces a review. This is then appended to the other reviews on the same claim.

can be simply listed alone in their scenario configuration. This is described along the implementation in chapter 4. For all conceptual discussions, a scenario refers to all the simulation environment fully specified, regardless of where they come from in the implementation.

- idea of author review - what each thing represents - accuracy is the agents competency "expertness" / unwilling inaccuracy - distortion is honesty / willing inaccuracy - author review is a declaration on what the claimer thinks his claim is worth

Author reviews are special reviews appended to each claim by the claiming agent, the author.

3.2.3 Claims

describe claims and what means what

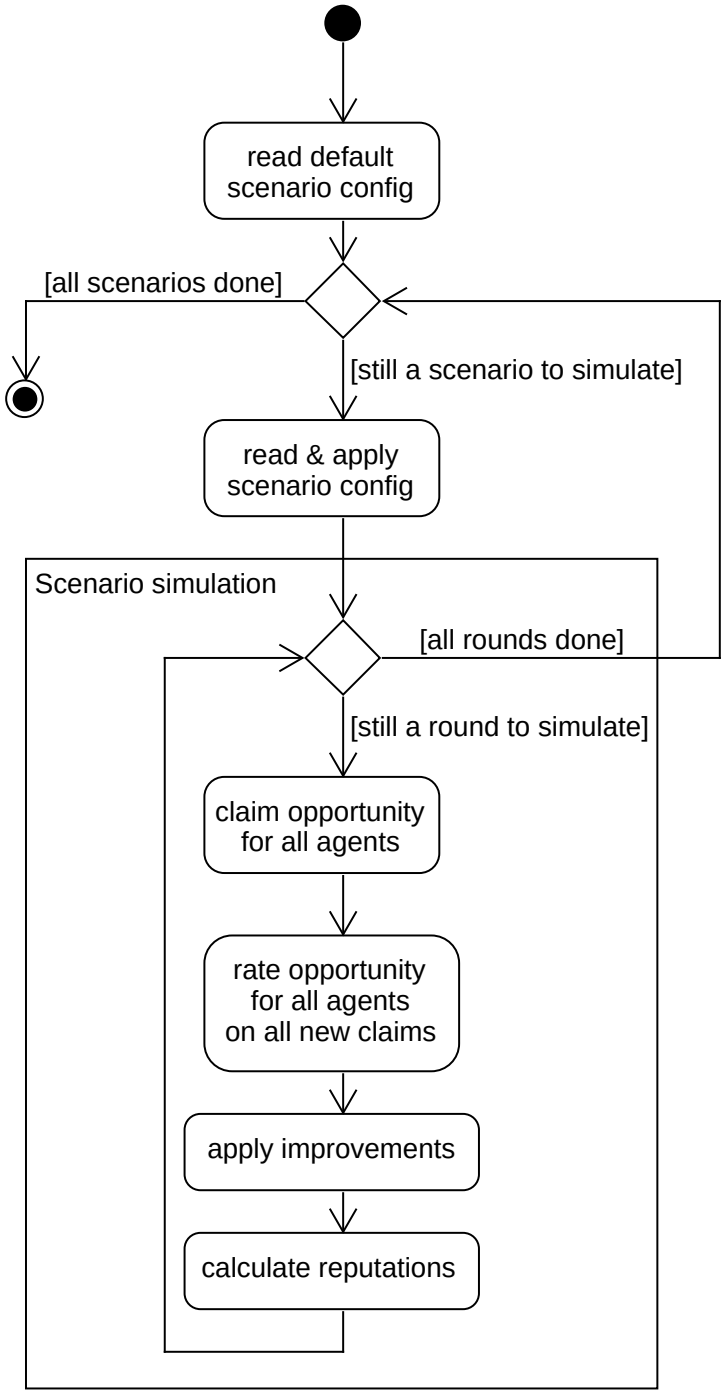

Figure 2: Simulation flow in Pyrepsys. A single simulation unit is the scenario. Each scenario is simulated independently after one another. Rounds make up the simulation within a scenario. Each round consists of four distinct phases: claiming, rating, improvement and reputation calculation. The number of rounds is specified in the configuration.

3.2.4 Reviews

describe reviews and what means what

3.2.5 Measuring Claims

measuring claims

3.2.6 Resolution

TODO THIS INTRO... Resolution gives control over what values are accepted at various parts of the simulation.
Used together with the rating range, resolution defines what values are possible in the system. For example, if reviews have a resolution of 1 and the minimum and maximum values are 1 and 9, then the result of ratings can be integers from 1 to 9.
There are two main uses for resolution during simulation. 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 is the implemented representation of this thought process. If it would return for example 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 as per Claim Truth Assessment Inaccuracy (CTAI). 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 a way to algorithmically assess a claim, it is reasonable to allow

more freedom for claim measurements.
A third resolution domain could also be created for reputations. This could limit the granularity of global reputations. The reason this was not included is that reputation calculation strategies serve as a single source of reputation values. They can adhere to specific resolution limits as part of their calculation if it is 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 behavior strategies for agents. Designing different versions of each with different resolutions unreasonable, hence the review resolution domain.

3.3 Reputation Calculation

BasedOnAvgDifferenceOfClaimsAndReviews ReputationAverageStrategy
why these two: - simple - thus easy to know what is happening - avg is a common-sense approach that a common user would expect when seeing a star-system (even tho that is often not the case in the rep calc, just implied) - BasedOnAvgDifferenceOfClaimsAndReviews uniquely uses the author-review system of claim rating the difference in their interpretation ReputationAverageStrategy: a rating becomes a quality of the claimer other one: review rates what this claim should have been rated by the author ability to easily incorporate weights

3.4 Distortion Strategies

[7] has the four static in the graph.. static: "pessimistic" rater (rates in the lower range) bad mouthing "optimistic" rater (rates in the higher range) ballot stuffing inverted rating (rates good as bad and vice versa) random rater (rates random all the time)
user tendentially rating extreme ... conservative random error rater (occasionally makes a significantly inaccurate rating, most ratings honest to a degree of noise, i.e. not perfect!) big inaccuracy e.g. erroneous measurement,sensor
Seller sells most products with high quality, but some with low quality for benefit. The honest bad reputation report from victim may be submerged, or be mistakenly regarded as unfair rating. [27]
dynamic: build up honest reputation and attack after that single out an agent and distort their rating only, honest otherwise mayb combine negative and positive attacks in collusion scenario, where attackers uprate each other and downrate their target mayb make difference bw. targeted attack on someone or group and just wrecking havoc on all individual vs collusion scenarios

3.5 Rating Strategies

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 Section 3.2.

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. They are data processors that automatically generate graphs and export data from simulations.

Flexibility and extendability 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.

Additional references for using Pyrepsys are provided outside this chapter. For a description of the Pyrepsys command-line interface, refer to Appendix B. A complete list and explanation of possible scenario configuration parameters are found in Appendix A.

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

Module	Notable Classes	Responsibility
agent	*Agent*, *Claim*, *Review*	Define agents with the simulation da
behavior		Submodule. Define agent rate and d
config	*Configurator*	Scenario configuration related. Read
errors		Custom exceptions
helper_types	*SimulationEvent*, *ResolutionDomain*, *LocalConfig*	Accessory type definitions
helpers		Accessory function definitions. Resol
instantiator	*Instantiator*	Creation of metric, reputation strate
main		Program entry point, preparation of
metrics		Submodule. Defines data export and
paths		Directory paths
reputation		Submodule. Define improvement har
results_processor	*ResultsProcessor*	Containment and coordination of me
scenario_creator		scenario creator utility
scenario_simulator	*ScenarioSimulator*	Simulate one single scenario. Aka Sy

Table 4.1: Modules making up Pyrepsys. Modules `behavior`, `reputation` and `metrics` form submodules. TODO FIX THE TABLE PARTS ARE OUT OF THE WORLD

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 processor module exports collected data and draws graphs based on the selected metrics. The simulation part's schematic flow is shown in figure 2.
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 3.2.5. With the measured truth, the agent executes its distortion strategy. It essentially applies one of the distortion methods found in section 3.4. 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 1.

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 show in equation 4.1.

$$\max(N^r_{reviews}) = N^r_{claims}(N_{agents} - 1) \tag{4.1}$$

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 1. 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 phase, all claims and reviews of the current round have been published. These were added to their appropriate places into the data structure formed by the list of agents. As discussed in detail in section 4.3, it contains all simulation data from the current scenario. This is passed to the selected improvement methods by the *ScenarioSimulator* for processing. All modifications are done 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 3 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 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)`

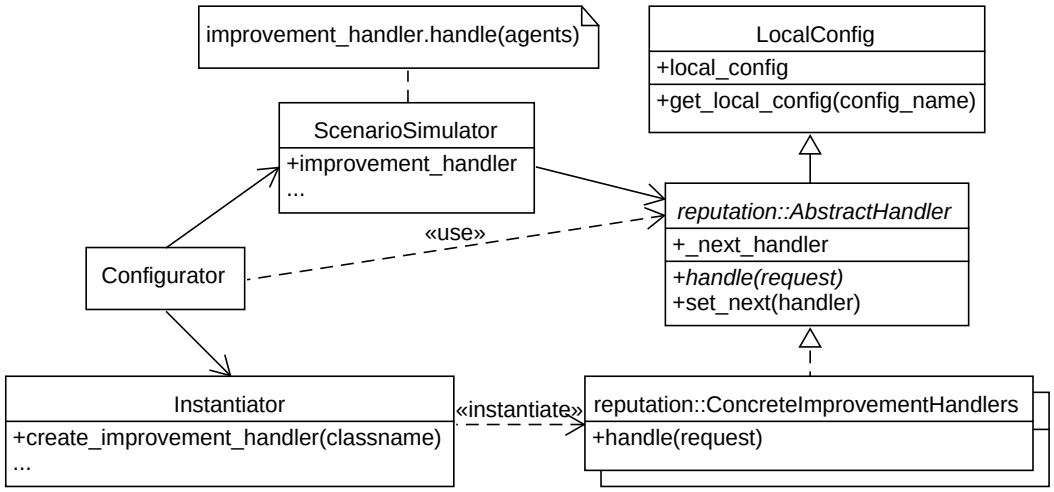

Figure 3: UML class diagram of improvement handlers. *Configurator* creates instances of the required improvement handlers via the *Instantiator* and sets the handler chain's entry point in *ScenarioSimulator*. When improvements are due, *ScenarioSimulator* calls the first handler with the agents data, 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.

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 4 shows the UML model of reputation strategies. implemented with the strategy pattern

4.2 Agents

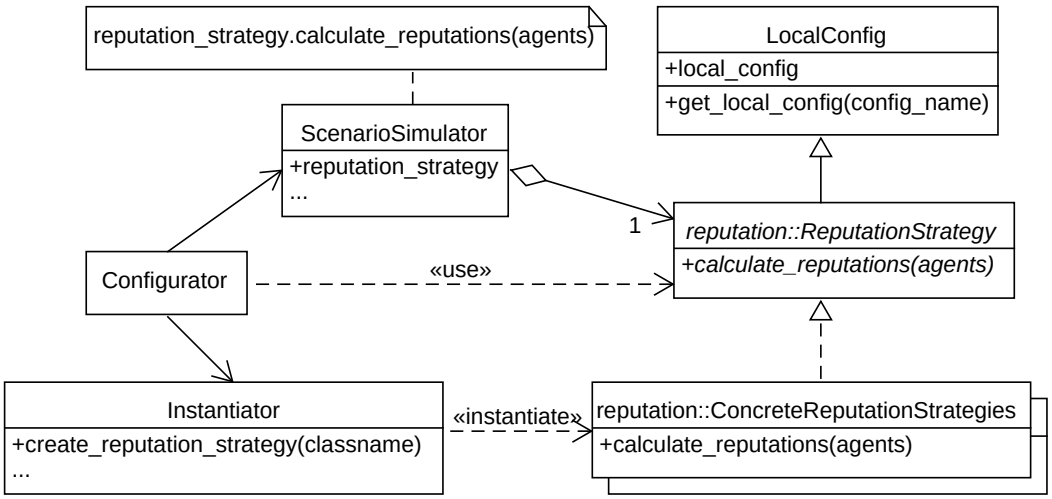

Figure 4: UML class diagram of reputation strategies. *Configurator* creates an instance of the selected reputation strategy via the *Instantiator* and gives it to *ScenarioSimulator*. At reputation calculation, *ScenarioSimulator* calls its reputation strategy object with the agents data, which includes all claims and reviews. *LocalConfig* provides custom configuration for strategy instances. This is discussed in section 4.5.3 in detail.

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 5. 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.

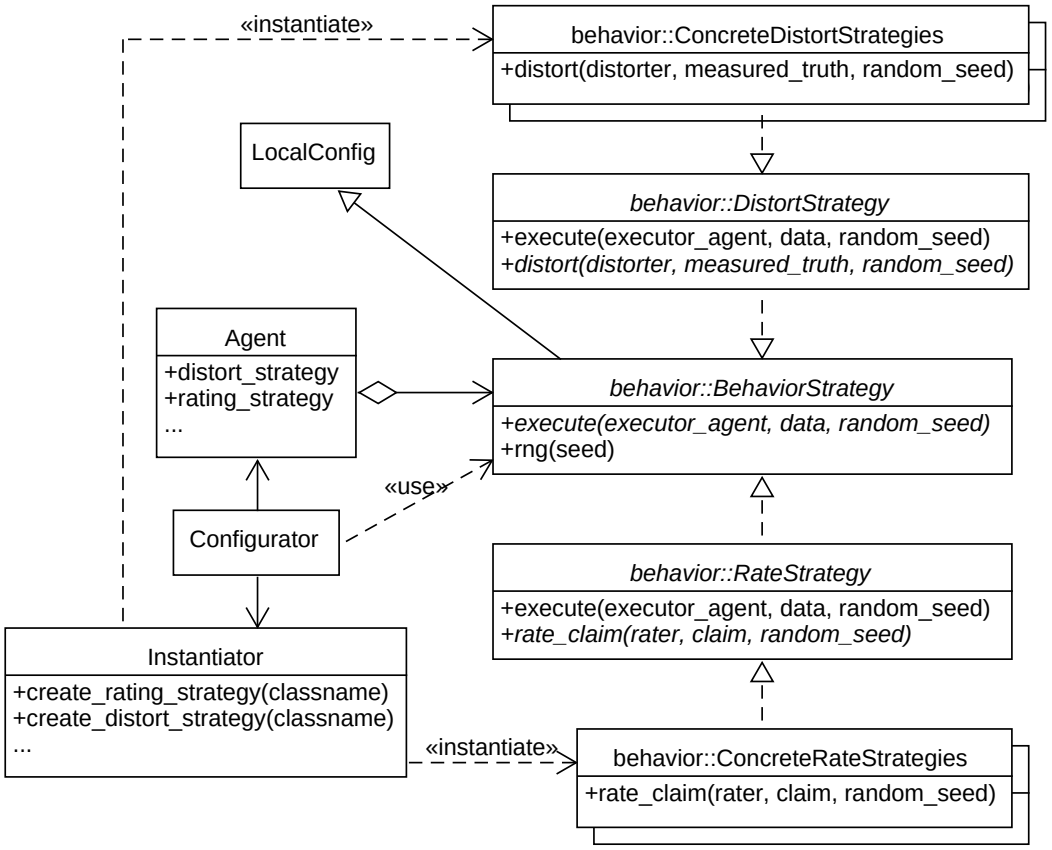

Figure 5: UML class diagram of agent behavior strategies. The abstract class *behavior::BehaviorStrategy* serves as a common base for both distort and rate strategies. Two interface classes *behavior::RateStrategy* and *behavior::DistortStrategy* both inherit from it. Concrete strategies are descendants 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.

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 a claim is the representation of trying out a product or content and making an opinion of what its quality could be. In more concrete terms, 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 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 4.2.

$$C_{gr\ truth} - \text{CTAI} <= \text{measured quality} <= C_{gr\ truth} + \text{CTAI} \tag{4.2}$$

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

$$\text{measurement error} = \text{rng.uniform}(-\text{CTAI}, \text{CTAI}) \tag{4.3}$$

$$\text{measured quality} = C_{gr\ truth} + \text{measurement error} \tag{4.4}$$

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 6.
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

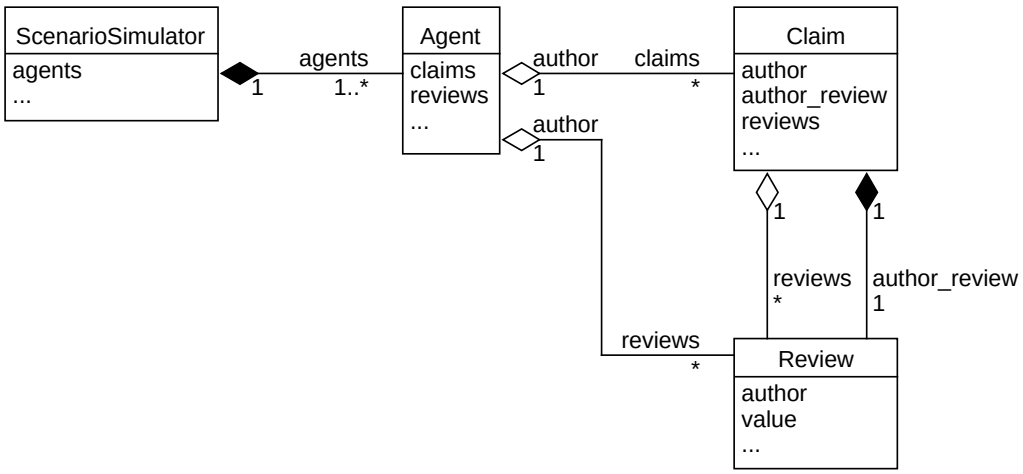

Figure 6: UML class diagram detailing how data is stored during simulation. *Agent* instances representing agents in the scenario are held in a list in *ScenarioSimulator*. Each agent holds a number of claims made by that agent. Claims have exactly one author review attached to them. Other reviews are 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.

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 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 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 4.5 and 4.6.

$$v_{internal} = \frac{v_{ae} - \text{minimum rating}}{\text{rating span}} \tag{4.5}$$

$$v_{agent\ exposed} = (\text{minimum rating}) + (\text{rating span}) * v_i \tag{4.6}$$

Used together with resolution, it is possible to reproduce schemes like the example ones at the beginning of this section. See sections 3.2.6 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.

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.

4.3.3 Resolution

Resolution settings control what exact values are allowed for select variables during simulation. This section discusses implementation details. See section 3.2.6 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 7 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 7.

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.

```
1 diff_left = round(abs( steps_sorted_list[i-1] - number ), 8)
2 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

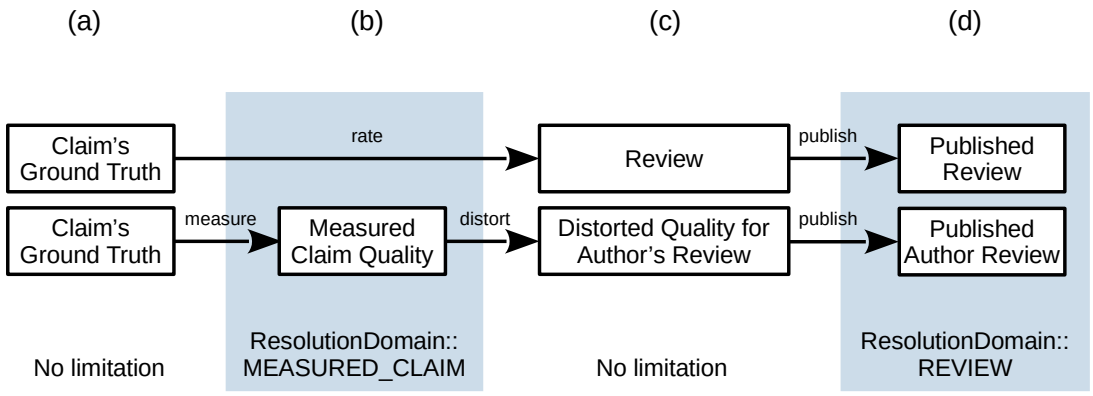

Figure 7: Resolution domains in various stages of rating (top) and claiming (bottom). (a) Ground truths have no limitation and can take up any value within the rating span, stored as internal representation. (b) Claim quality measurements are produced in the *measured claim resolution domain*. (c) In 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*.

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 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 provides a scenario creator utility, which uses the same approach to create scenarios with combinations of given parameter values. See appendix C for documentation.

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.

```
1 improvement_handlers:
2   - name: "Aging"
3     limit: 6
4   - "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 *Instantiator* 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,

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.

metrics and *ResultsProcessor* form a publish-subscribe architecture. The UML diagram of these classes is found in figure 8.

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.

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.

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.

Every metric is defined as a child of the abstract *metrics::Metric* class, as seen in figure 8. 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.
The benefit of metrics is the convenience of automatic reports and graphs exported based on simulation data. Flexibility and extendability 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.

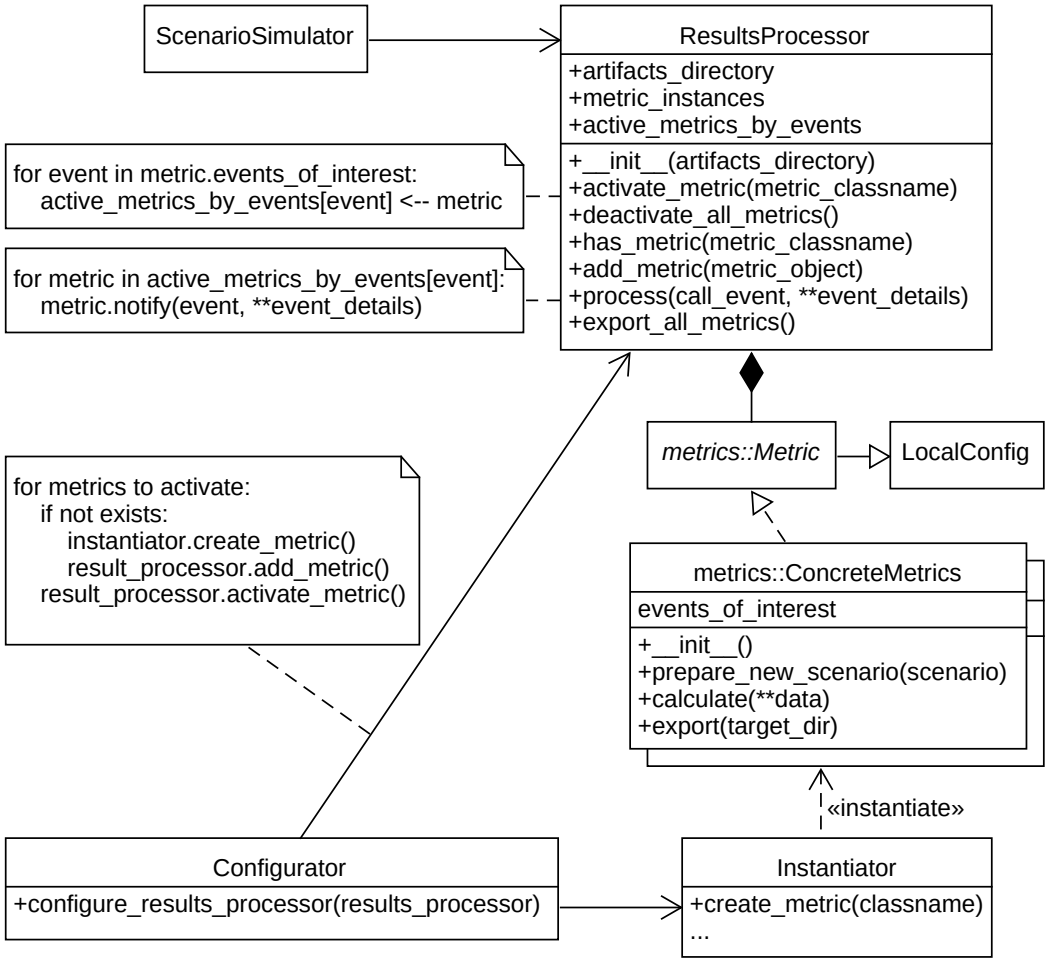

Figure 8: UML class diagram of the results processor and metrics. Metric instances are created by *Configurator* via *Instantiator*. *ResultsProcessor* registers metrics according to what event they are listening to. *ScenarioSimulator* and the `main` package trigger events on the results processor. Metrics are notified of events that interest them. *LocalConfig* provides custom configuration for each individual metric. This is discussed in section 4.5.3.

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.

5

Evaluation

5.1 Metrics

- reputation - avg tot claim inaccuracy - also other metrics from my notebook

5.2 Setup

simulation setup – used settings, scenarios describe different simulation bundles (of scenarios) and their rationale why they are interesting and what's expected

5.3 Results

5.4 Discussion

6

Conclusion

We successfully ...

6.1 Outlook

But we still need to ...

A

Scenario Configuration Options

TODO comprehensive list of all scenario parameters

B

Command Line Interface and Invocation

TODO extensive CLI description

C

Scenario Creator

TODO SC desc

Acronyms

CTAI Claim Truth Assessment Inaccuracy. 14, 24

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. 28, 29, 35, 55

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.. 39, 41

author review TODO. 12, 19, 25, 26

branch-off random chain TODO. 22, 24, 32

claim TODO. 11, 18

claim range TODO. 19

claiming probability A percent likelihood determining whether an agent initiates a claiming process if given an opportunity. Specified in scenarios for each agent.. 18

distorted claim quality TODO a value produced by the distort strategy from the measured claim q. 19

distortion strategy TODO. 19

ground truth TODO. 12, 19, 23

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. 28, 31, 35, 55

main random chain TODO. 18, 19, 32

measured claim quality TODO. 19, 22, 23

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.. 19

rating strategy TODO. 19

rating span The size of the rating range, calculated as the difference of the maximum and minimum review values.. 28, 31

review TODO. 12, 19

scenario Description of a complete simulation environment for Pyrepsys. Specifies the reputation scheme, improvement methods, agents, possible review values etc. See section 3.2.2 for details.. 11, 13, 17

Bibliography

[1] 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. 3, 4, 6

[2] 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. 4, 5

[3] 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. 5, 8

[4] 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

[5] 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. 6

[6] J. Zhang and R. Cohen, "A Personalized Approach to Address Unfair Ratings in Multiagent Reputation Systems," p. 10, 2006. cited on p. 6

[7] B. Yu and M. P. Singh, "Detecting Deception in Reputation Management," p. 8, 2003. cited on p. 6, 15

[8] 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. 7

[9] 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. 7

[10] 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. 7

[11] 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. 7

[12] L. Ngo, "A survey of unfair rating problem and detection methods in reputation management systems," p. 12, 2007. cited on p. 7

[13] F. Azzedin, "Identifying Honest Recommenders in Reputation Systems," p. 7, 2010. cited on p. 7, 10

[14] 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. 7

[15] 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. 7

[16] 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. 7

[17] 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. 7

[18] 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. 8

[19] 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. 8

[20] 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. 8

[21] 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. 8

[22] 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. 8

[23] 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. 8

[24] J.-Y. L. Boudec and S. Buchegger, "A Robust Reputation System for Mobile Ad-hoc Networks," tech. rep., 2003. cited on p. 8

[25] T. D. Huynh, N. R. Jennings, and N. R. Shadbolt, "On Handling Inaccurate Witness Reports," p. 15, 2005. cited on p. 8

[26] 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. 8

[27] Ping Xu, Ji Gao, and Hang Guo, "Rating Reputaion: A Necessary Consideration in Reputation Mechanism," in *2005 International Conference on Machine Learning and Cybernetics*, (Guangzhou, China), pp. 182–187, IEEE, 2005. cited on p. 15