ABM Supplement for A Framework to Develop Interventions to Address Labor Exploitation and Trafficking: Integration of Behavioral and Decision Science within a Case Study of Day Laborers

Model Overview

The AnyLogic model is utilized for the agent-based-modeling approach to address the exploitation and trafficking issue in the labor supply network. It shows the impact of two different kinds of intervention agents that model high-level diffusion through a broadcasting advertising campaign and a more direct "word-of-mouth" approach, as well as other features relevant to this scenario such as a laborer reporting feature when a laborer has faced exploitation.

Simulation

The model upon start contains a simulation window with a graphical user interface so that a user can input or change parameters before the simulation run. This is helpful for running multiple simulations and comparing the outputs of the log files (Excel spreadsheets) without having the simulation runner go into the backend code and change parameters manually. The parameters are grouped by the agents that are directly affected by them.

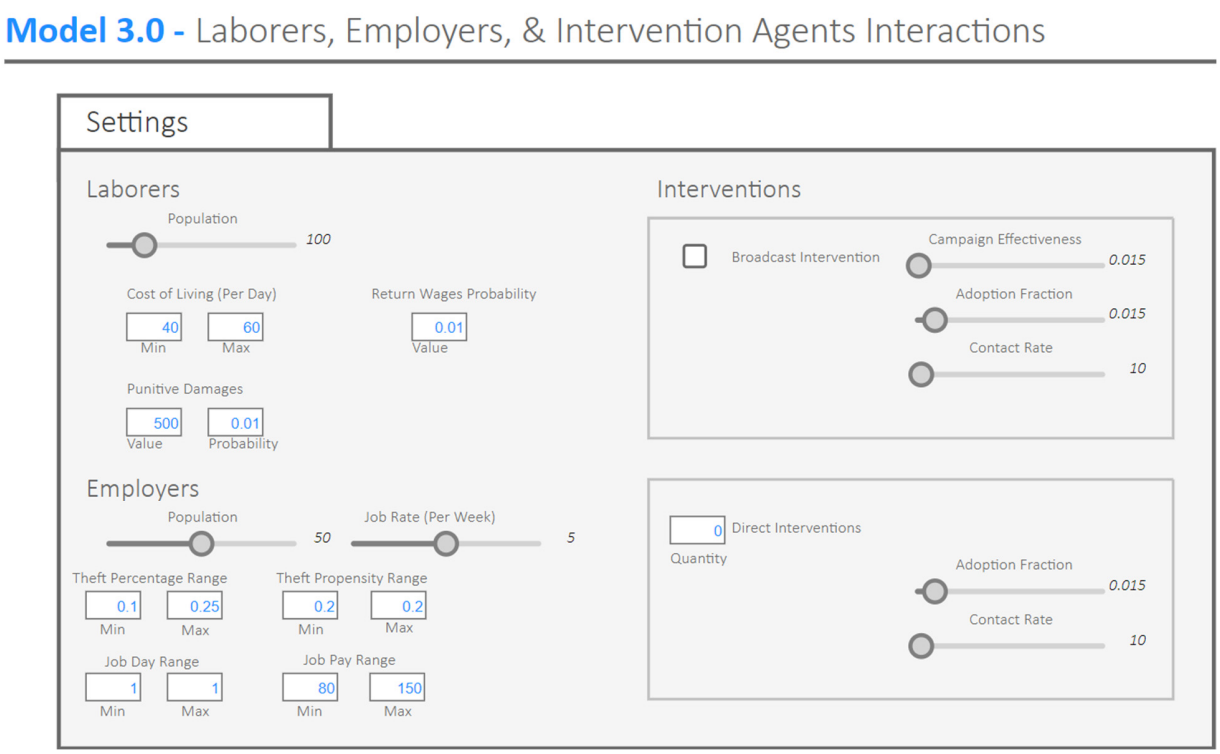

Figure S1: AnyLogic simulation window upon startup showing the interface in which the users can change parameters.

Laborers currently have 4 parameters that can be changed with the graphical user interface.

1) **Population:** Can be changed through a slider mechanism. The default value is set to 100. Value is used to populate the Main window's simulation social network visual with laborer agents.
2) **Cost of Living (Per Day):** Can be changed through Min/Max input fields with a default value set to Min: 40, Max: 60. Setting a value in the Min input field and a different value in the Max will cause the simulation to generate a uniform random distribution of Cost of Living for each laborer upon initialization. To have a constant singular value for the entire laborer population, the user can set the Min and Max input field to be the same value. An error will occur if the Max value is less than the Min value. The above-documented explanation of the Min/Max input fields holds for all other parameters that use this feature. The cost of living for a Laborer will be subtracted from their total money variable after a day (Or every 24 simulation hours.) to simulate the value that the laborer tries to subset through working jobs created by employers.
3) **Return wages Probability:** Can be changed through a user input field that takes in a value between 0.0 and 1.0. The default value is set to 0.01. This value is utilized in the random draw to determine if the report is successful and if the total wages stolen would be returned to the laborers.
4) **Punitive Damages:** This parameter has a nominal value and a probability. The default value of the probability is set to 0.01. This value is utilized in the random draw to determine if a successful report also includes a return of punitive damages to the laborer. The nominal value of the punitive damages represents the amount the employer would have to return to the laborer along with the stolen wages.

Employers currently have 6 parameters that can be changed with the graphical user interface.

1) **Population:** Can be changed through a slider mechanism. The default value is set to 50. Value is used to populate the Main window's simulation social network visual with employer agents.
2) **Job Rate (Per Week):** Can be changed through a slider mechanism. The default value is set to 5.0. The job rate can be set anywhere from 0.0 to 10.0 and represents the number of jobs an employer can create and give out during a given simulation week.
3) **Theft Percentage Range:** Can be changed through Min/Max input fields. This represents a value between 0.0 and 1.0 or in other words the percentage of perceived pay that an employer will withhold from their employed laborer agent if the employer agent decides to commit wage theft. The default value is set to Min: 0.1, Max 0.25.
4) **Theft Propensity Range:** Can be changed through Min/Max input fields. Like the Theft Percentage parameter, this represents the percentage of jobs in which an employer will decide to commit wage theft. Values of 0.0 to 1.0. The default value is set to Min: 0.2, Max: 0.2.
5) **Job Day Range:** Can be changed through Min/Max input fields. A random uniform distributed variable with default values of Min: 1, Max:1 day that is initialized every time an employer creates a job.
6) **Job Pay Range:** Can be changed through Min/Max input fields. A random uniform distributed variable that represents the perceived daily pay a laborer will get upon completion of their respective jobs. Default values set to Min: 80, Max: 150.

There are 2 main interventions: Broadcast Intervention and Direct Intervention. Interventions currently have 5 parameters that can be changed with the graphical user interface. Three of the parameters affect one of two types of intervention agents and the other two affect the other. All the parameter values are extracted through AnyLogic Java code and stored into variable placeholders that are to be used in Main and throughout the simulation run.

1) **Broadcast Intervention:** Has a checkbox to toggle the single Broadcast Intervention agent which functions similarly to an advertising campaign. Given the simulation social distance network area

is limited, there is a presumption that only one is needed currently. There is a possibility in the future to include multiple broadcast interventions given parameters such as funds, dosing effects, and the respective AnyLogic implementation that would be needed to make those features possible.

- ***Broadcast Campaign Effectiveness:*** Represents the effectiveness the advertising campaign has on the laborer population. The actual implementation of this particular advertising mechanism is borrowed from the AnyLogic Bass Diffusion model example and functions as a countdown mechanism that all laborers will eventually reach an informed state given a certain amount of time reflective of real-world campaigns. The default value is set to 0.015.
- ***Broadcast Adoption Fraction***: The value that represents the likelihood a laborer when seeing the advertising of a campaign will take/retain the information being diffused. This value is utilized/multiplied with the campaign effectiveness value when an informed laborer decided to contact another laborer. (Diffusing information through word of mouth.) The default value is set to 0.015.
- ***Broadcast Contact Rate:*** The number of other laborer agents an informed laborer agent will contact in a simulated month time. This is multiplied with the adoption fraction to simulate that not all laborers will diffuse the information but rather a percentage of those who have received the information. The default value is set to 10.

2) **Direct Intervention:** This type of intervention has an input field where the user can change the number of direct intervention agents that populate the Main social distance network area with the default value set to 0. They diffuse the information about worker rights through the word-of-mouth message mechanism rather than the Broadcast intervention agent's advertising mechanism. All the direct intervention agents will have the following parameter values when initialized.
   - ***Direct Adoption Fraction:*** Functions as the Broadcast adoption fraction parameter but is only applied to the word-of-mouth messaging mechanism. The default value is set to 0.015.
   - ***Direct Contact Rate:*** Functions as the Broadcast contact rate parameter but is only applied to the word-of-mouth messaging mechanism. The default value is set to 10.

Agents

The model includes agents for employers and Laborers. These are described next.

Employer

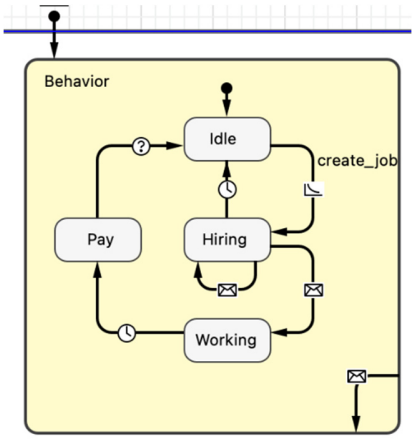

Figure S2: Employer state chart depicting the agent's behavior.

The Employer agents' behavior is depicted in the above state chart which contains 4 states: *Idle, Hiring, Working, Pay,* and 7 notable transitions: *create_job, job_timeout, laborer_declined, hired_laborer, job_completion, reset,* and *reported*. All of which have varying Java code implementations and function calls from the Java class files.

1) **Idle**: Once an Employer agent is initialized at the start of the simulation, they begin their behavior in the Idle state. They wait there until the create_job transition is taken at a rate of jobs per week. The parameter is set in the simulation parameter interface.
    - **create_job**: Once the transition is taken, the Java code constructor from the Job.java file is called with the parameters Employer agent creating the job, the current time (Simulation time in days passed since simulation start), and Main (to access the parameters set in the simulation parameter interface.) The variable current_job is now set to the job object that the constructor returns.

Feedback Loop: Create_job moves the Employer from an Idle state to a Hiring one. The Job_timeout transition moves the Employer from Hiring back to Idle, while a Labor_declined transition keeps the Employer in a Hiring Loop until Job_Timeout or Hired_Laborer, in which case the Employer moves to the Working State.

2) **Hiring**: Once the current job has been set the employer transitions into the Hiring state. In this state, the Employer agent will send out job offers in the form of Job object messages to randomly connected Laborer agents. The messages are sent one at a time and the employer waits for a response which corresponds to 2 of the 3 transitions leading out of Hiring. If a Laborer declines an Employer, the Employer will not send another job offer of the same job object to them.
    - **job_timeout**: If after 3 simulation days and the Employer has not gotten a Laborer agent to accept his job offer, the job object expires and gets added to his job history. The Employer loops back to the Idle state.
    - **laborer_declined**: The Employer agents wait to receive a message object from the potential hire Laborer agent. If the Employer receives the message "Decline Job" then the Employer will loop back to the Hiring state and repeat the process with the decliner added to a do not hire list which is reset upon a new current job.
    - **hired_laborer**: If the Employer agent receives the message "Accept Job" then they will take the hired_laborer transition. They call the Job function hireLaborer() which takes the parameters: Laborer agent reference and the current simulation time. The laborer gets added as the worker for the job object and time is saved for logging purposes.
3) **Working**: The Employer then moves to the working state which is essentially a delay state to reflect the time taken for the job to complete.
    - **Job_completion**: This transition is taken after the current job's days have passed in simulation time to reflect the time taken for the current job to complete.
4) **Pay**: In this state the Employer agents' variables theft_propensity, percent_theft, and decision are taken to compute whether the Employer commits wage exploitation on the Laborer agent or if they pay them the total amount of perceived pay. The job object is updated appropriately along with the variables for specific agent stat tracking.
    - **reset**: This transition is to reset the current job reference to null and add the job to the agent's history. They then return to the Idle state after the code is run within this transition.

reported: A continuous transition that waits for a message object with the message "Reported". When an agent receives said message their theft propensity variable is lowered. This is effectively a feedback loop. Once the employer is reported, the propensity is lowered, as more reports occur, the employer is less likely to commit wage theft, eventually tending to zero. As more laborers become educated, they are more likely

to report, and the likelihood of wage theft continues to decrease. The degree of decreased theft propensity depends on the effectiveness of reporting and the associated penalty.

Laborer

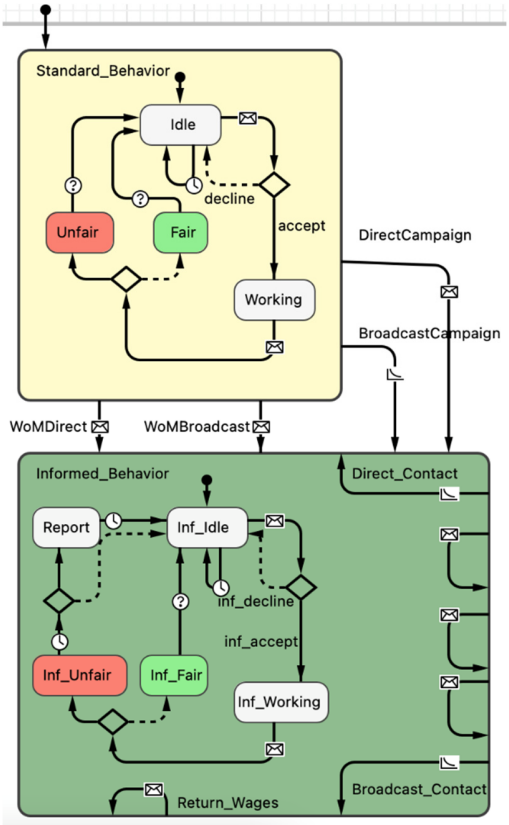

Figure S3: Laborer state chart depicting the agent's behavior. Note the two different behaviors: Standard and Informed

The Laborer agents' behavior is depicted in the above two state charts labeled Standard Behavior and Informed Behavior. The Standard Behavior state chart is composed of 4 states: *Idle, Working, Fair, Unfair;*

Transitions: *job_offer, accept, decline, idle_waiting, job_completion, paid_fairly, paid_unfairly, fair_reset, unfair_reset*; and 2 branches: *decision_making, job_making*. The Informed Behavior state chart is composed of 5 states: *Inf_Idle, Inf_Working, Inf_Fair, Inf_Unfair, and Report.*

There are 17 transitions: *inf_job_offer, inf_accept, inf_decline, inf_idle_waiting, inf_job_completion, inf_paid_fairly, inf_paid_unfairly, to_report_decision, inf_fair_reset, do_nothing_reset, report, report_reset, Direct_Contact, Receive_Direct_Message_WOM, Receive_Direct_Message, Receive_Broadcast_Message, Broadcast_Contact*; and 4 State Chart Transitions: *BroadcastCampaign, DirectCampaign, WordOfMouthBroadcast, and WordOfMouthDirect.*

Behavior State Chart States

1) **Idle/Inf_Idle**: The initial starting states (With Idle being the initial starting state for all Laborer agents at the beginning of a simulation run.) where the Laborer agents wait for a Job Message from an employer.
2) **Working/Inf_Working**: A waiting state to simulate the time that is needed to complete the current_job based on the job's labor_days variable.
3) **Fair/Inf_Fair**: current_job is revealed to be fair, i.e no payment is stolen. Labor returns to Idel/Inf_Idle state.
4) **Unfair/Inf_Fair:** current_job is revealed to be unfair, i.e payment is stolen and subtracted from the laborer's total wages. Transition into inf_paid_unfairly and the worker has the option to report or reset back to Idle/Inf_Idle state.
5) **Report**: state within the informed state chart. Sends a reporting message to the employer to trigger the reporting functionality. Also appends information within the job memory code.

Branches

1) **decision_making/inf_decision_making**: The branch that contains the memory "sifting" code in which a Laborer iterates through their memories data structure looking for previous instances of the Employer who sent them the Job message. Once a memory instance is found the Laborer will call on the JobMemory function access and tolerate to simulate if they can recall the memory given the time that has passed since the instance and checks if there was any wage exploitation that may have occurred. This is stored in the Laborer agent's accept_job variable and is the deciding factor if the accept/inf_accept to the Working/Inf_Working state or decline/inf_decline transition to the Idle/Inf_Idle state.
2) **job_making**: Creates job

State Chart Transitions

1) **BroadcastCampaign**: Transition from the Standard_Behavior state chart to the Informed_Behavior state chart based on the campaign_effectiveness_broadcast parameter of the BroadcastIntervention agent. Occurs if broad_enabled is set to true in Simulation.
2) **DirectCampaign**: Transition from the Standard_Behavior state chart to the Informed_Behavior state chart based on direct messages sent by the Direct_Intervention agent in Main. Occurs if direct_enabled is set to true in Simulation.
3) **WordOfMouthBroadcast**: Transition from the Standard_Behavior state chart to the Informed_Behavior state chart based on direct messages sent by Laborer agents to other Laborer agents that were informed initially by the Broadcast Campaign.
4) **WordOfMouthDirect**: Transition from the Standard_Behavior state chart to the Informed_Behavior state chart based on direct messages sent by the Laborer agents to other Laborer agents that were informed initially by the Direct Campaign.

Transitions

1) **job_offer/inf_job_offer**: The transition from Idle to decision_making/inf_decision_making. In this transition, the Laborer sets their current_job variable to the Job message received from the Employer agent.
2) **accept/inf_accept**: The transition taken if Laborer's accept_job variable is true. Sends a String message back to the Employer with "Accept Job" to alert the Employer that they have accepted the job and can begin "work". The Laborer and Employer agents transition into their respective Working states.

3) **decline/inf_decline**: The transition taken if Laborer's accept_job variable is not true. Sends a String message back to the Employer with "Decline Job" to alert the Employer to continue their job search. The Laborer goes back to Idle.
4) **idle_waiting/inf_idle_waiting**: The transition that is set to occur every simulation day in order to log the net loss of a Laborer agent if they have not received a job.
5) **Job_completion/inf_job_completion**: A transition state taken once the Employer sends the Laborer an int message representing the end of the job (The Employer sends the int message once the labor days of the current working job has passed) and the pay the Employer gives the Laborer.
6) **paid_fairly/inf_paid_fairly**
7) **paid_unfarily/inf_paid_unfarily**: The transition into the Unfair/Inf_Unfair state that does the calculations to see if the actual payout of the Job to the Laborer by the Employer is less than the agreed-upon pay. If not then the full amount had been paid and the paid_fairly/inf_paid_fairly state is taken.
8) **Unfair_reset** and 9) **Fair_reset / Report_reset** and 10) **do_nothing_reset**: Transitions used to house the logging functions and memory creation functions for the Laborer. The Laborer returns to the idle state with their current_job set to null.
9) **Direct_Contact**: Continuous transition in the Informed_Behavior state chart that sends "Direct WOM Information" messages to other laborer agents based on the contact_rate_direct and adoption_fraction_direct parameters set in Simulation.
11) **Broadcast_Contact**: Continuous transition in the Informed_Behavior state chart that sends "Broadcast Information" messages to other laborer agents based on the contact_rate_broadcast and adoption_fraction_broadcast parameters set in Simulation.
12) **Receive_Direct_Message_WOM**: Continuous transition to receive messages and update the total_messages variable accordingly.
13) **Receive_Direct_Message**: Continuous transition to receive messages and update the total_messages variable accordingly.
14) **Receive_Broadcast_Message**: Continuous transition to receive messages and update the total_messages variable accordingly.
15) **Report branch**: the branch that leads to Report or Inf_idle. Currently has if-else statements to determine if a laborer will report or do nothing. The conditions are if the net income of the laborer at the conclusion of a job puts them in the negative, messages compared to theft percentage, otherwise, the laborer will do nothing.

Java Class

The model includes three custom Java classes: Job, Memory, and Job Memory. These are described next.

Job

The Job Java class file was created separately from the agents in order to consolidate variables and methods that are used/accessed extensively by the Employer and Laborer agents. It also helps with simplifying the clutter in the code blocks of state charts and transitions.

```
public class Job implements Serializable {

    Employer employer;
    Laborer laborer;
    //Creation of job/constructor call by employer within the create_job path.
    double create_date;
```

```
//Time when a laborer gets hired.
double start_date;
//Time when laborer gets paid and job completes.
double end_date;
//Int of days to pass for job to complete and laborer to get paid.
int labor_days;
//Per day.
int perceived_pay;
//Perceieved pay * labor days.
int perceived_payout;
//Actual pay out given to laborer from employeer.
int pay_out;
//Double represent % of total payout withheld by employer.
double percent_theft;
String info;
int job_id; // job id for this job
```

The above snippet of code is from the Java class file Job depicting its variables.
Methods and variables are used by the Employer and Laborer agents. All the variables are currently made public for ease of access but can be transitioned into getters and setters later on. Unknown references are set to unknown and unknown integer/double values are set to -1 initially. Below is a detail of the Java class file Job variables.

- **Employer employer**: a reference to the Employer agent that created the current job. Should not be null initially due to all jobs being created by an Employer agent.
- **Laborer laborer**: a reference to the Laborer agent that will be hired to "work" on the job. Should be null initially until an Employer hires one through the messaging mechanism.
- **double create_date**: A double representing the simulation time in days when the job's constructor was called by the Employer agent.
- **double start_date**: A double representing the simulation time in days when a Laborer has been hired and both the Laborer and Employer agent transition into the working state.
- **double end_date**: A double representing the simulation time in days when the job completes (both simulation time and based on the labor_days variable.) and when the Laborer gets paid by the Employer.
- **int labor_days**: An integer that is initialized upon job constructor call that represents how long it takes (in terms of simulation time) for the job to complete once a Laborer has been hired. This value is determined by the parameters set in the simulation parameter interface at the beginning of a simulation run.
- **int perceived_pay**: An integer that is initialized upon job constructor call that is the pay rate per day the Laborer perceives to receive upon job completion. This value is determined by the parameters set in the simulation parameter interface.
- **int perceived_payout**: An integer that is initialized upon job constructor call that is the total amount of money the Laborer can expect to receive in an int message object by the Employer agent referenced in this job upon completion. The value is determined by multiplying perceived_pay by labor_days.
- **int pay_out**: An integer that is initially -1 and represents the actual payout that the Laborer receives upon job completion from the Employer. Cannot be greater than perceived_payout and is set upon job completion.
- **double percent_theft:** A double that represents the percentage amount of theft from the perceived_payout that a Laborer experiences upon pay/job completion. Initially set to -1.
- **String info**: A string that appends any key information that the job may have for logging purposes. Such as if at the conclusion of this job there was wage exploitation, a report, or if the job simply expired without any Laborer taking the job.

Job Method Class

Below is a snippet of code depicting the 6 methods in Job Java class code, including the constructor, that is called throughout the Employer and Laborer state charts.

```
//Standard Constructor based on simulation initial parameters.
public Job(Employer employer, double create_date, Main main) {
        this.employer = employer;
        this.laborer = null;
        this.create_date = create_date;
        this.start_date = -1;
        this.end_date = -1;
  //Uniform random distribution based on parameters.
        this.labor_days = (employer.rand.nextInt(( /*Max*/ main.job_day_range_max - /*Min*/ main.job_day_range_min) +
1) + /*Min*/ main.job_day_range_min)
  //Uniform random distribution based on parameters.
        this.perceived_pay = (employer.rand.nextInt(( /*Max*/ main.job_pay_range_max - /*Min*/
main.job_pay_range_min) + 1) + /*Min*/ main.job_pay_range_min);
        this.perceived_payout = perceived_pay * labor_days;
  //-1 if the number is unknown at the time of initialization.
        this.pay_out = -1;
        this.percent_theft = -1;
        this.job_id = main.entries++; }

//Adds the laborer to the job information if the laborer accepts/
public void hireLaborer(Laborer laborer, double start_date) {
        this.start_date = start_date;
        this.laborer = laborer; }

//-Reported text is appended if employer is reported by labor
public void reported() {
        this.info += " - Reported"; }

//If no laborer takes the job after a certain amount of time.
public void expire() {
        this.info = "Not Completed - Job Expired"; }

//Fills out information if the employer decides to fully compensate laborer.
public void fair_pay(double end_date) {
        this.pay_out = this.perceived_payout;
        this.employer.job_value -= this.pay_out;
        this.employer.send(pay_out, this.laborer);
        this.percent_theft = 0.0;
        this.info = "Completed - Fairly Paid";
        this.end_date = end_date; }

//Fills out information if the employer decides to pay laborer any amount less than the agreed upon pay.
public void unfair_pay(double percent_theft, double end_date) {
        this.percent_theft = percent_theft;
        this.pay_out = this.perceived_payout*(1 - this.percent_theft);
        this.employer.job_value -= this.pay_out;
        this.employer.send(pay_out, this.laborer);
        this.info = "Completed - Wage Theft";
        this.end_date = end_date; }
```

- **Job(Employer employer, double create_date, Main main)**: The first method as shown in the above snapshot of Java code is the constructor called when initializing a Job in the Employer's create_job

transition. It needs three parameters: 1) a reference to the employer who called the constructor, 2) the current simulation time to fill the create_date, and 3) a reference to the main to access its parameters (the parameters set by the user in the simulation parameter interface) to fill the rest of the variables.

- **hireLaborer(Laborer laborer, double start_date)**: The method called by the Employer during the hired_laborer transition which sets the job's Laborer variable as the laborer reference parameter. The start date is also set and the laborer being hired also starts the working state.
- **reported()**: If the Employer is reported by the Laborer at the end of the job, info gets appended with the "- Reported" text.
- **expire()**: If the job reaches its expiration time (3 Days) then this method appends that information into the info variable for logging purposes.
- **fair_pay(double end_date)**: Fills out the rest of the unknown variables when an Employer decides to fully pay the Laborer the perceived amount. Pay_out is set to perceived_out. The employer sends that amount as an integer object to the Laborer. The percent_theft is set to 0.0. The info variable is appended with "Completed – Fairly Paid" and the end date is set to the current time parameter.
- **unfair_pay(double percent_theft, double end_date)**: Fills out the rest of the unknown variables when an Employer decides to commit wage exploitation against the Laborer. Percent_theft is set to the parameter. Payout is calculated by the perceived_payout * (1 – percent_theft). The employer sends that amount as an integer object to the Laborer. The info variable is appended with "Completed – Wage Theft" and the end date is set to the current time parameter.

Memory

Similar to the Java class file, Job, the functions and variables of memories used by the Laborers are separated and consolidated into the Memory Java class file. Below is a snippet of code from the Java class file Memory depicting its variables and main functions. Methods and variables are used by the Laborer agents.

```
public class Memory implements Serializable {
        //At completion of event.
        double time;
        //Reference to Main of simulation in order to access parameters.
        Main main;
        double k = 1.0;
        //Initializing a basic memory.
  public Memory(double time, Main main) {
        this.time = time;
        this.main = main;
  }
  //Gets the time difference since the event of the memory was completed to the function call.
  public double timeSince(double current_time) {
        return current_time - this.time;
  }
  //Returns a double representing memory strength based on a decay curve.
  public double strength(double current_time) {
        return (Math.exp(-k*(current_time - time)));
  }
```

```
  //Returns a boolean if the memory's information is accessible to the agent calling it based on memory strength and random uniform distribution.
  public boolean access(double current_time) {
        Random rand = new Random();
        double recall_value = rand.nextDouble();
        double memory_strength = this.strength(current_time);

        if (recall_value < memory_strength) {
              return true;
        } else {
              return false;
        }
  }
```

There are two variables that every Memory object will have.

- **Double time**: a double that represents the time in which the memory occurred in simulation time units. Since the simulation is in days a time variable would for example look like 2.00 to represent that the memory occurred on the second day of the simulation.
- **Main main**: a reference to the main window of Anylogic to access variables from the simulation.

There are four main functions, including the initial constructor, that every Memory has.

- **Memory (double time, Main main)**: the initial constructor called in the transitions to Idle by the Laborer to create the memory and store it within their memory ArrayList data structure. The time here is at the conclusion of the event, typically a job after work and payment.
- **timeSince(double current_time)**: a function to find out how much time, in a double format, has passed since the memory occurred. Returns the difference between the two times (current time – time of memory).
- **Strength(double current_time)**: using current_time as the parameter and input t, returns the strength calculated by the exponential decay curve equation.
- **Access(double current_time)**: function used to see if the agent (Laborer) can recall the current memory object with a double input current_time. Using a random uniform distribution paired against the memory strength of the variable calculated with the current time returns a Boolean.

Job Memory

The Memory Java class was created as an umbrella class for all other possible memories that could be in the Laborer agent's memory data structure. Currently, there is one memory derivative that extends the Memory Java class: Job Memory. Below is a snippet of code from the Java class file Memory depicting its variables and main functions. Methods and variables are used by the Laborer agents.

```
public class JobMemory extends Memory implements Serializable {
  Job job;
        //Constructor for a memory of subclass job.
  public JobMemory(double time, Job job, Main main) {
        super(time, main);
        this.job = job;
  }
  //Returns a boolean if the laborer was exploited based on the job in question's data.
   public boolean exploited() {
        if (this.job.perceived_payout > this.job.pay_out) {
              return true;
        } else {
              return false;
        }
  }
```

```
//Returns a double based on the amount of pay thefted by an employer.
public double theftPercent() {
        if (this.exploited()) {
                return this.job.percent_theft;
        } else {
                return 0.0;
        }
}

//The laborer doesn't tolerate every time there's been theft in a particular job memory when making a decision to accept a job.
public boolean tolerate(){
        if (this.job.percent_theft > 0.0) {
                return false;
        } else {
                return true;
        }
}
```

Job Memory has one new variable (as well as the time and Main which is inherited from the Memory Java class.) which is a Job object that contains all the information/variables of that specific job. There are four main functions, including the Job Memory-specific initial constructor.

- **JobMemory(double time, Job job, Main main)**: the constructor that takes in the same inputs as the parent class Memory with the additional Job object reference.
- **Exploited()**: a Boolean returning function that checks the Job object and returns true if the Job Memory is of a job in which the Laborer has a portion of their wages withheld by their employer.
- **theftPercent()**: a function that returns a double of the percent theft (0 to 1.0) that occurred with the Job.
- **Tolerate()**: a function that checks if there's been a wage theft in a particular job memory when making a decision to accept a job. Returns a Boolean to see if the Laborer accepts/declines a job in their state charts.

Bass Diffusion Model

We incorporated the training manipulation examined in the case study as a third category of agent implemented as a type of Bass diffusion model (BDM). Within the BDM approach, we implemented a modality to include the effect of advocates working directly with laborers in the field and a modality associated with a public awareness campaign (PSA).

The BDM captures the dynamics of the response and rationale of a population when presented with a new product and documents the adoption process of the product. For our study, the 'products' introduced to the labor agents are anti-exploitation interventions: advocates and public service campaigns. The product adoption rate throughout the population in a BDM depends on exposure to the product and feedback from adopters to non-adopters. The dynamics of the model are defined by a differential Equation (S1) which describes the process of new product, or information, diffusion into a population .

$$\frac{dN(t)}{dt} = \left(m - N(t)\right)\left[p + \frac{q}{m}N(t)\right], t \geq 0$$	(S1)

where

$N(t)$ ≡ number of people converted at time t as part of the campaign.
m ≡ size of population
p ≡ conversion as a direct consequence of the campaign (innovation)

$q \equiv$ conversion from word of mouth

The degree of innovation, p, represents the external motivation for adoption, while the degree of imitation, q, is the effect of word of mouth in the diffusion of a product. In the present case study, p and q are coefficients that were determined though reviewing previous research. The quantity $m - N(t)$ is the number of members of the population who have not adopted at time t. The quantity $p + q*(N(t)/m)$ is the probability of adoption at time t, and incorporates the direct effect of the campaign, p, and word of mouth, $q*N(t)/m$, from those who have adopted.

In the current study, the ABM was developed to allow for two forms of intervention to be tested: direct training by advocates and media campaigns. Our case study's educational manipulation informed how direct training was incorporated in the model.

Coefficient Estimates

The value estimates in the model for p (coefficient of innovation, external influence or advertising effect) and q (coefficient of imitation, internal influence or word-of-mouth effect) were found from estimates in reviewed studies. Below is a table documenting p and q value estimates used in deciding the p and q values used in our model.

p-value	q-value	Description	Additional Information	Source Paper
Cohort A: p= 0.021 Cohort B: p=0.015 Cohort C: p=0.015 All smokers over 20: p=.005 p=early adopters from media exposure	Cohort A: q= 0.149 Cohort B: p=0.162 Cohort C: p=0.222 All smokers over 20: q=.038 q= word of mouth communication	p and q effected by attractiveness and affordability of innovation estimated using the procedures outlined in Takada and Jain (1991) on SPSS NLIN software	$P_t = p+q\,[Y_{t-1}/m]$ Y_{t-1}= cumulative number of adopters through period t -1 P_t= probability of an initial purchase in period t P[=innovation rate q=imitation rate To estimate parameters of diffusion models, percentage data of quitting were converted to numbers of quitting persons by census population figures	*Original:* Redmond, W. H. (1996). Product disadoption: Quitting smoking as a diffusion process. *Journal of Public Policy and Marketing*, *15*(1), 87–97. https://doi.org/10.1177/074391569601500108 *References:* Takada, Hirokazo and D. Jain (1991), "Cross-National Analysis of Diffusion of Consumer Durable Goods in Pacific Rim Countries," Journal of Marketing, 55 (April), 48-54
p= coefficient of internal influence mass media communications	q= coefficient of external influence (word of mouth)	Value of q is very high compared to other BRIC countries in the sample	External Influence Models: $d(N(t)/d(t)=p(m-N(t))$ N(t): cumulative number of subscribers til period t M = total number of potential adopters p= constant (coefficient of external influence)	*Original:* Kumar, Ashish & Baisya, R.K. & Momaya, Kirankumar. (2007). Diffusion of mobile communications: Application of bass diffusion model to BRIC countries. Journal of Scientific and Industrial Research. 66.

p= 0.0019 p=0.00322 p=0.00013, 0.000274 p= 0.0005832, 0.000198 p=0.0007 p=0.00089	q= 1.2513 q=0.8591 q=0.999, 0.302 q=0.553,0.264 q= 0.2422 q=0.2318	Endogenous Potential KBA Annual Data Endogenous Potential KBA Monthly Data Exogenous Potential KBA annual data Exogenous Potential KBA monthly data Toyota Hybrid worldwide Toyota Hybrid Europe	Parameter estimates based on market data for Germany on the diffusions of different car technologies. Considers parameter estimation for p &q from literature and compares to data estimation. $nt=dNtdt=pXt+qYt$ $X_t = M-N_t$ Yt=1MNt(M−Nt) If M and N_t are given, both X_t and Y_t can be calculated. This makes it possible to estimate p and q.	*Original:* Massiani, J., & Gohs, A. (2015). The choice of Bass model coefficients to forecast diffusion for innovative products: An empirical investigation for new automotive technologies. *Research in Transportation Economics*, *50*, 17–28. https://doi.org/10.1016/j.retrec.2015.06.003 *References:* T.A. Becker, I. Siduh, B. Tenderich **Electric vehicles in the United States a new model with forecasts to 2030** University of California, Berkeley (2009) M. Davidson, D. Cross, M. Craig, A. Bharatkumar **Assessing options for accommodating electric vehicles in Santa Delano Valley** USAEE Case Competition 2013. T. T. a. P. Group: 49 (2013)
q = coefficient of innovation Method 1 Estimate: p = 0.02496 Method 1 Value: p = 0.0048 Method 2: p = 0.017	q = coefficient of innovation Method 1 Estimate: q = 0.14026 Method 1 Value: q = 0.7817 Method 2: q = 0.439	Use Bass Diffusion Techniques to forecast a Neurostimulator Device with No Historical Data Method # 1: Calculate p and q from an analogous product's past sales	Determine probability of purchase at time (t) given that individual has not purchased product before $n(t) = p(0) + q/m\ (N(t)$ n(t) = probability of purchase time t p(0) = coefficient of innovation at time t = 0	*Original:* Farnaz Ganjeizadeh, Howard Lei, Preetpal Goraya, Erik Olivar, Applying Looks-like Analysis and Bass Diffusion Model Techniques to Forecast a Neurostimulator Device with No Historical Data, Procedia Manufacturing, Volume 11(2017) Pages 1916-1924, ISSN 2351-9789, https://doi.org/10.1016/j.promfg.2017.07.334. (https://www.sciencedirect.com/science/article/pii/S2351978917305425)

		Method #2: Calculate p and q from analogous products whose p and q estimates are provide.	q = coefficient of imitation m = total number of buyers in market N(t) = number of people who have adopted the product at time t Continuous density function over time $n(t) = pm + (q-p)N(t) - (q/m)(N(t))^2$ n(t) = sales in period t N(t) = cumulative sales to period t m = total number of buyers in the market p = coefficient of innovation q = coefficient of imitation Discrete form $N(t) = a+bN(t-1)-c(N(t-1))^2$ $M = (-b\sqrt{b^2} - 4ac)/2c$ $p = a/m$ $q = p+b$ a, b and c can be estimated using least squares regression	

p = coefficient of innovation p_2= fraction of tasters from promotional activities multiplied by barrier towards tasting fraction of potential tasters from promotional activities = 0.0036 1/year	q = coefficient of imitation q_2= strength of the word of mouth multiplied by barrier towards tasting strength of word of mouth = 0.151 Dmnl	Modifying the Bass Diffusion Model for the adoption of insect-based foods in the Netherlands. Mass media = external influence Word of mouth = internal influence	$\frac{dT(t)}{dt} = (p_2 + \frac{q_2}{m}(T(t) + R(t) + N(t)) * (m - T(t) - R(t) - N(t)) - c(t)T(t) - d(t)R(t)$ Barrier towards tasting: average disgust level from 0 to 1	*Original:* Horvat A, Fogliano V, Luning PA (2020) Modifying the Bass diffusion model to study adoption of radical new foods–The case of edible insects in the Netherlands. PLoS ONE 15(6): e0234538. https://doi.org/10.1371/journal.pone.0234538 *References:* Goldenberg J, Shapira D. Marketing: Complexity Modeling, Theory and Applications in. In Meyers AR, editor. Encyclopedia of Complexity and Systems Science. New York: Springer New York; 2009. p. 5377–5392. Tan HSG, Fischer ARH, van Trijp HCM, Stieger M. Tasty but nasty? Exploring the role of sensory-liking and food appropriateness in the willingness to eat unusual novel foods like insects. Food Qual Prefer. 2016;48: 293–302.