Tim Righettini

3/31/13

Professor Wilczynski

CSCI-201

**Factory Design Document: Version 0**

**Purpose:** This piece of software will build the first agent based version of what will become part of a larger glass factory project in version 1.

**Requirements (as stated on the website):**

<http://www-scf.usc.edu/~csci201/factory/TheGlassLine.pdf>

For this version of the project, the following objectives have to be completed:

1. Formulate a Team Interaction Diagram for the whole factory normative scenario.

2. Compile an individual v0 Design Doc, complete with pseudo code, for the agent messaging system.

3. Write up the agent code for the design, and then Unit test it to make sure that it works.

**Acronyms:**

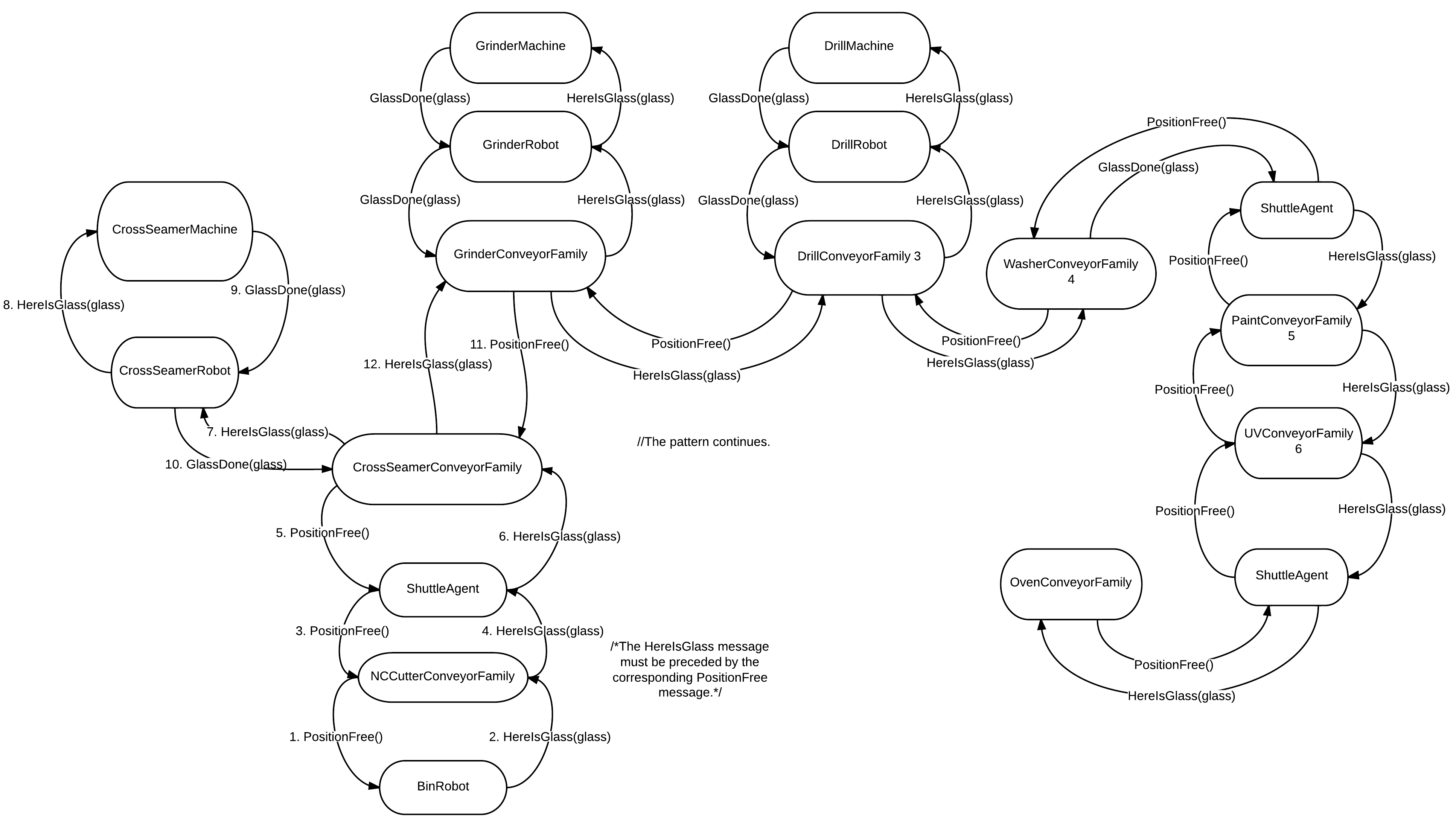
**\*****or t.e. = There Exists**

**\*s.t. = Such That**

**\*****or f.a. = For All**

***Same* acronyms apply for the rest of the document**

**Normative Interaction Diagram: Team**

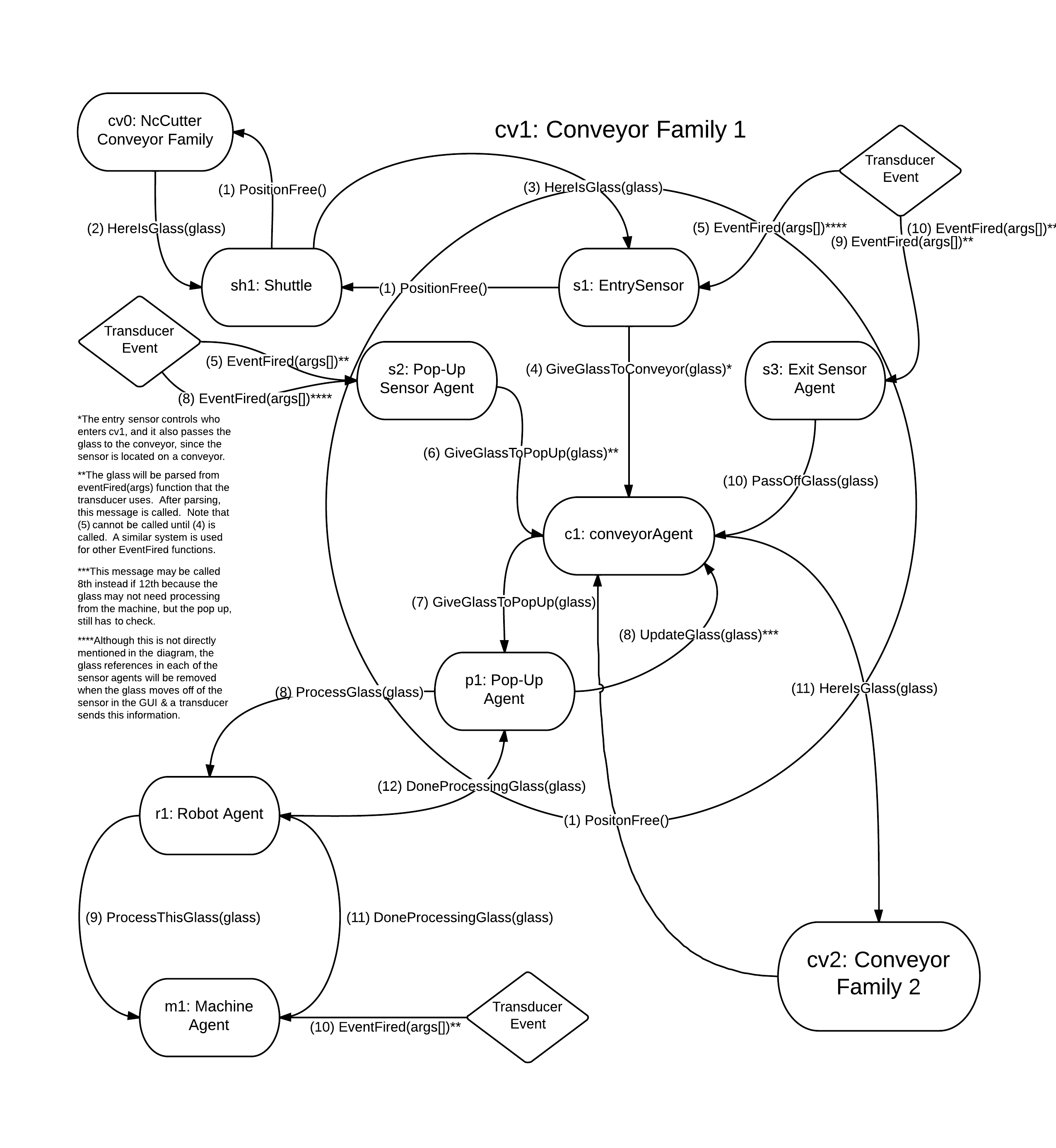


**Normative Interaction Diagram: Team** includes the simple normative scenario under one hood. This scenario will not be addressed in v0, but in v1.

This helps satisfy the following scenarios:

1. Formulate a Team Interaction Diagram for the whole factory normative scenario.

\*Note, the PNG for this interaction diagram is included full size within the submission so that it is easier to read.

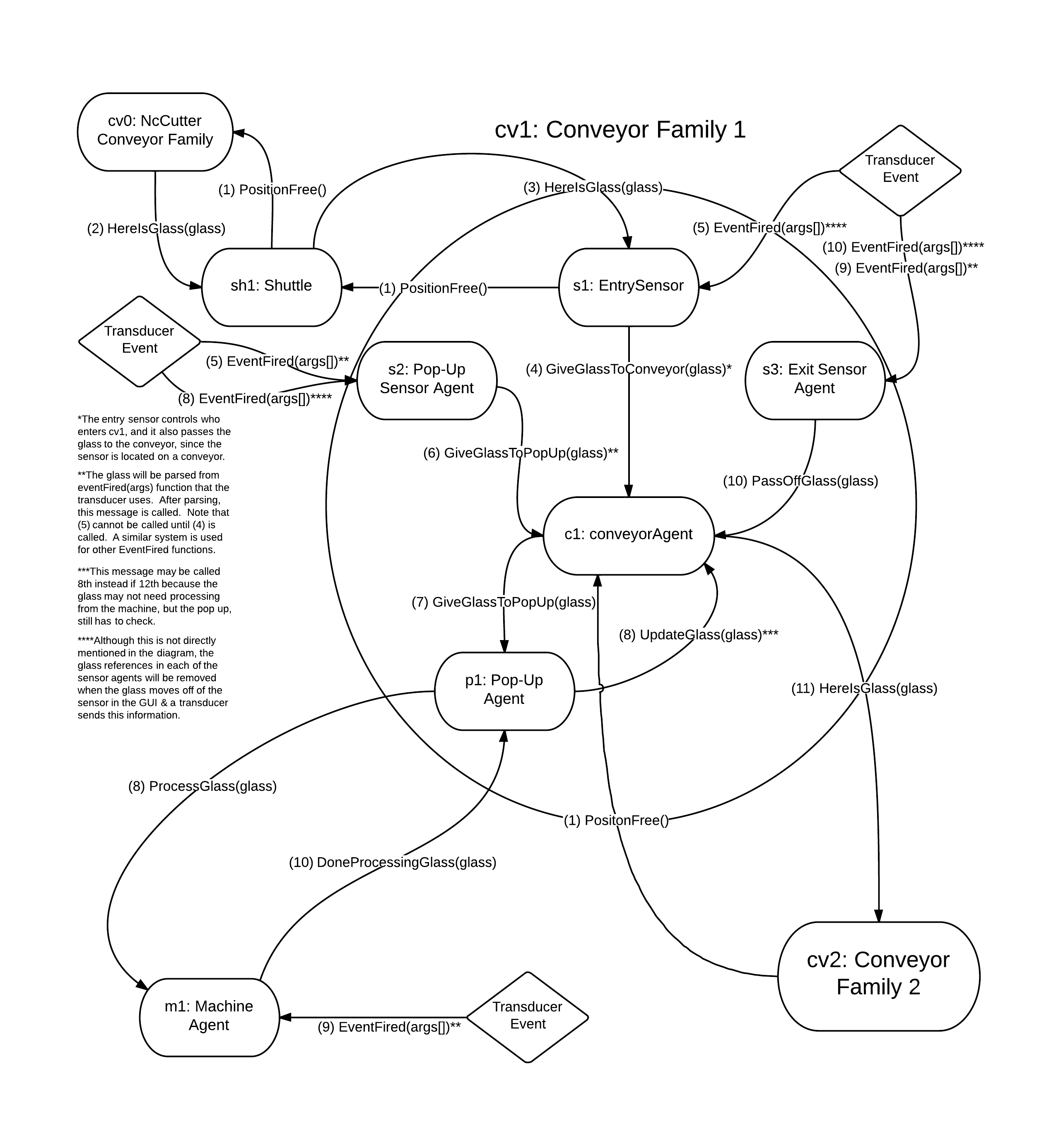
**Normative Interaction Diagram: Individual (Tim) – Version 1**

**Normative Interaction Diagram: Individual (Tim) Version 1** includes the normative scenario for the agent messaging system to be used for the conveyor family.

This helps satisfy the following scenarios:

2. Compile an individual v0 Design Doc, complete with pseudo code, for the agent messaging system.

3. Write up the agent code for the design, and then Unit Test it to make sure that it works.

**Normative Interaction Diagram: Individual (Tim) – Version 2**

**Normative Interaction Diagram: Individual (Tim) Version 2** includes the normative scenario for the agent messaging system to be used for the conveyor family. This design removed the robot agent, since the requirements for the project changed. The robot agent will still be described in this version of the pseudocode, though, just in case it is still needed (the references will be a little off, though, since the machine and popUp agents will be updated to remove the references to robots). This design still fulfills the same requirements of the previous one.

**Agents & Other Classes:**

**Name:** Agent (Base Class)

**Description:** The Base Class that all agents inherit from.

**Data:**

Semaphore stateChange; // Semaphore to use to put the agent thread to sleep

AgentThread agentThread;

Transducer transducer; // Will hold a reference to the transducer

TracePanel tracePanel; // A link to the tracePanel on the animation GUI

**Methods:**

run() { // This method will be used for the agent’s thread

while (agent is active) {

if (pickAndExecuteAnAction)

then performAction()

else

then sleep() until stateChanged()

}

}

stateChanged(); // Release a permit from the semaphore

stopThread();

startThread();

pickAndExecuteAnAction(); // Scheduler

eventFired(); // Used for the transducer

---------------------------------------------------------------------------------------------------------------------

**Name:** Glass

**Description:** The class with values relating to a piece of glass – this is more like a struct.

**Data:**

int id; // the identifier for this piece of glass

Map<MachineType, Boolean> recipe; // Contains the recipe for this piece of glass

MachineType[] machineIDs = {MachineType.Cutter, MachineType.Cross\_Seamer, MachineType.Grinder, MachineType.Drill, MachineType.Washer, MachineType.Paint, MachineType.UV\_Lamp, MachineType.Oven} ; // Will hold references to anything that could be a part of the recipe

**Methods:**

N/A

---------------------------------------------------------------------------------------------------------------------

\*Note: For the next classes, there will be interfaces created that contain abstract implementations of the messaging methods. This will be used for unit testing and the conveyorFamily, specifically. This description will suffice for the rest of the interfaces relative to each agent/class.

---------------------------------------------------------------------------------------------------------------------

**Name:** Sensor (Interface)

**Messages**:

msgHereIsGlass(Glass glass);

// The following messages will be special to transducer events, and will be called after parsing arguments in the EventFired(args[]) function.

msgGlassOffSensor(Glass glass);

---------------------------------------------------------------------------------------------------------------------

**Name:** SensorAgent

**Description:** Will detect if a piece of glass has entered, exited, or on a popup for any given set of conveyors. Even though all of the sensor functionality currently melded into one agent, I may split this agent up into a base agent and three inheritance agents – EntrySensorAgent, ExitSensorAgent, and PopUpSensorAgent – during implementation

**Data:**

Class MyGlass {

Glass glass; // Holds a reference to the glass

enum onSensor {justEntered, yes, no}; // Is the glass on an given sensor?

enum location {entry, popup, exit}; // Which sensor the glass is currently on – this will not be needed if using the multiple inheritance design paradigm

}

List<String> type; // Will hold the type of sensor this is, and it may be of more than one type

List<MyGlass> glassSheets; // Will hold all glass references

ConveyorFamily cf; // Reference to the current conveyor family

**Messages**:

msgHereIsGlass(Glass glass) {

glassSheets.add(new MyGlass(glass, location.entry, onSensor.justEntered);

stateChanged();

}

// The following messages will be special to transducer events, and will be called after parsing arguments in the EventFired(args[]) function.

msgGlassOffSensor(Glass glass) {

if ( g in glassSheets s.t. g.glass.id == glass.id) then

g.onSensor = onSensor.no;

stateChanged();

}

msgHereIsGlass(Glass glass) {

glassSheets.add(new MyGlass(glass, getLocationFromSensor(), onSensor.justEntered);

stateChanged();

}

**Scheduler:**

if ( g in glassSheets) then

if (g.onSensor == onSensor.justEntered) then

actPassGlassToConveyor(g); return true;

if (g.onSensor == onSensor.no) then

actRemoveGlass(g); return true;

return false;

**Actions:**

actPassGlassToConveyor(MyGlass g) {

if (g.location == location.entry) then

cf.conveyor.msgGiveGlassToConveyor(g.glass);

g.onSensor = onSensor.yes;

if (g.location == location.popup) then

cf.conveyor.msgGiveGlassToPopUp(g.glass);

g.onSensor = onSensor.yes;

if (g.location == location.exit) then

cf.conveyor.msgPassOffGlass(g.glass);

g.onSensor = onSensor.yes;

}

actRemoveGlass(MyGlass g) {

glassSheets.remove(g);

if (g.location == location.entry &&  cf.prevCF) then // Tell the previous conveyor family that the sensor currently has nothing on it

cf.prevCF.msgPositionFree();

}

**Other Methods:**

N/A

---------------------------------------------------------------------------------------------------------------------

**Name:** PopUp (Interface)

**Messages**:

msgGiveGlassToPopUp(Glass g);

msgDoneProcessingGlass(Glass g);

---------------------------------------------------------------------------------------------------------------------

**Name:** PopUpAgent

**Description:** Will act as a mediator between the conveyor agent and the robot agents for getting glass to the processing machines, if necessary.

**Data:**

Class MyGlass {

Glass glass;

enum processState {unprocessed, doneProcessing};

}

// The reason why there is not a middle stage is because this glass is removed from the pop-up

// during processing – there should be no reference to a glass sheet that is being processed in the

// pop-up agent when it is not with the pop-up agent and with the robot or machine agents

Class RobotCom { // Will hold a communication channel to a robot, allowing for the possibility to communicate to multiple robots at once

RobotAgent robot; // Robot reference

boolean inUse; // Is this channel currently occupied by a piece of glass

MachineType processType; // What process does this robot do? Does the glass need to undergo this process?

MyGlass glassBeingProcessed; // This reference needs to be held so PopUpAgents know which piece of glass is being processed by the robot. This name will be abbreviated to gBP.

} // Note, in the version 2 design, the robots are replaced by machines

List<MyGlass> glassToBeProcessed; // This name will be abbreviated as gTBP in many functions to save on space and complexity

List<RobotCom> robotComs; // Note, in the version 2 design, robots are replaced by machines

// Positional variable for whether the Pop-Up in the GUI is up or down, and it will be changed through the transducer and checked within one of the scheduler rules

boolean popUpDown; // Is this value is true, then the associated popUp is down (will be changed through the appropriate transducer eventFired(args[]) function.

conveyorFamily cf; // Conveyor Family reference

**Messages**:

msgGiveGlassToPopUp(Glass g) { // Get Glass from conveyor to PopUp

gTBP.add(new MyGlass(g, processState.unprocessed));

stateChanged();

}

msgDoneProcessingGlass(Glass g){

gTBP.add(new MyGlass(g, processState.doneProcessing));

if ( com in robotComs s.t. com.gBP.glass.id == g.id) then

com.inUse = false;

com.gBP = null;

else // There is a bug – this should never happen

stateChanged();

}

**Scheduler:**

if ( g in gTBP s.t. g.processState == processState.unprocessed) then

if ( com in robotComs s.t. com.inUse == false && popUpDown == true) then

actPassGlassToRobot(g, com); return true;

if ( g in gTBP s.t. g.processState == processState.doneProcessing) then

actPassGlassToConveyor(g); return true;

return false;

**Actions:**

actPassGlassToRobot(MyGlass g, RobotCom com) {

if (g.glass.recipe.contains(com.processType) then

com.robot.msgProcessGlass(g.glass);

com. glassBeingProcessed = g;

com.inUse = true;

gTBP.remove(g);

else

g.processState == processState.doneProcessing;

actPassGlassToConveyor(g);

// Remove statement isn’t needed – it is done within the actPassGlassToConveyor

}

actPassGlassToConveyor(MyGlass g) {

cf.conveyor.msgUpdateGlass(g.glass);

gTBP.remove(g);

}

**Other Methods:**

N/A

---------------------------------------------------------------------------------------------------------------------

**Name:** Conveyor (Interface)

**Messages**:

msgGiveGlassToConveyor(Glass g);

msgGiveGlassToPopUp(Glass g);

msgPassOffGlass(Glass g);

msgPositionFree();

msgUpdateGlass(Glass g);

---------------------------------------------------------------------------------------------------------------------

**Name:** ConveyorAgent

**Description:** Will hold the glass until it needs to go into the next conveyor for a different set of processes, or to leave the factory entirely.

**Data:**

Class MyGlass {

Glass glass;

enum conveyorState {onConveyor, passPopUp, passCF};

}

List<MyGlass> glassSheets; // List to hold all of the glass sheets

boolean positionFreeNextCF; // Will determine if a piece of glass should be passed to the next conveyor family. This will initially be set to true.

ConveyorFamily cf;

**Messages**:

msgGiveGlassToConveyor(Glass g) {

glassSheets.add(new MyGlass(g)); // conveyorState will always initializes to onConveyor

stateChanged();

}

msgGiveGlassToPopUp(Glass g) {

if ( glass in glassSheets s.t. glass.glass.id == g.id) then

glass.conveyorState = conveyorState.passPopUp;

stateChanged();

}

msgPassOffGlass(Glass g) {

if ( glass in glassSheets s.t. glass.glass.id == g.id) then

glass.conveyorState = conveyorState.passCF;

stateChanged();

}

msgPositionFree() {

positionFreeNextCF = true;

stateChanged();

}

msgUpdateGlass(Glass g) { // This message is akin to a stub, but I wanted to match up to my current interaction diagram – I could just call msgGiveGlassToConveyor directly, but the semantics do not look as good that way

msgGiveGlassToConveyor(g);

stateChanged();

}

**Scheduler:**

if ( g in glassSheets s.t. g.conveyorState == conveyorState.passPopUp && cf.popUp.gTBP.empty() == true &&  com in cf.popUp.robotComs s.t. com.inUse == false) then

// This rule will only work when the glassSheet is supposed to go to the PopUp, when there is nothing on the pop-up, and when there is a available robot to process the glass

actPassGlassToPopUp(g); return true;

if ( g in glassSheets s.t. g.conveyorState == conveyorState.passCF && positionFreeCF == true) then

actPassGlassToNextCF(g); return true;

**Actions:**

actPassGlassToPopUp(MyGlass g) {

cf.popUp.msgGiveGlassToPopUp(g.glass);

glassSheets.remove(g);

}

actPassGlassToNextCF(MyGlass g) {

cf.nextCF.msgHereIsGlass(g.glass);

glassSheets.remove(g);

positionFreeNextCF = false;

}

**Other Methods:**

N/A

---------------------------------------------------------------------------------------------------------------------

**Name:** Robot (Interface)

**Messages**:

msgProcessGlass(Glass g);

msgDoneProcessingGlass(Glass g);

\*Note that this interface is currently NOT IN USE with the version 2 design

---------------------------------------------------------------------------------------------------------------------

**Name:** RobotAgent

**Description:** Will transfer a piece of glass from the popUpAgent to an associated machine to complete a process.

\*Note that this class is currently NOT IN USE with the version 2 design

**Data:**

Class MyGlass {

Glass glass;

enum processState {unprocessed, doneProcessing};

}

Class MachineCom { // Will hold a communication channel to a machine, allowing for the possibility to communicate to multiple machines at once

MachineAgent machine; // Machine reference

boolean inUse; // Is this channel currently occupied by a piece of glass

MachineType processType; // What process does this machine do? Does the glass need to undergo this process?

MyGlass glassBeingProcessed; // This reference needs to be held so RobotAgents know which piece of glass is being processed by the Machine. This name will be abbreviated to gBP.

}

List<MyGlass> glassToBeProcessed; // This name will be abbreviated as gTBP in many functions to save on space and complexity

List<MachineCom> machineComs;

ConveyorFamily cf;

**Messages**:

msgProcessGlass(Glass g) { // Get Glass from popUp to robot

gTBP.add(new MyGlass(g, processState.unprocessed));

stateChanged();

}

msgDoneProcessingGlass(Glass g){

gTBP.add(new MyGlass(g, processState.doneProcessing));

if ( com in machineComs s.t. com.gBP.glass.id == g.id) then

com.inUse = false;

com.gBP = null;

else // There is a bug – this should never happen

stateChanged();

}

**Scheduler:**

if ( g in gTBP s.t. g.processState == processState.unprocessed) then

if ( com in machineComs s.t. com.inUse == false) then

actPassGlassToMachine (g, com); return true;

if ( g in gTBP s.t. g.processState == processState.doneProcessing) then

actPassGlassToCF (g); return true;

return false;

**Actions:**

actPassGlassToMachine(MyGlass g, MachineCom com) {

com.machine.msgProcessGlass(g.glass);

com. glassBeingProcessed = g;

com.inUse = true;

gTBP.remove(g);

}

actPassGlassToCF(MyGlass g) {

cf.conveyor.msgDoneProcessingGlass(g.glass);

gTBP.remove(g);

}

**Other Methods:**

N/A

---------------------------------------------------------------------------------------------------------------------

**Name:** Machine (Interface)

**Messages**:

msgProcessGlass(Glass g);

msgDoneProcessingGlass(Glass g);

---------------------------------------------------------------------------------------------------------------------

**Name:** MachineAgent

**Description:** Will compete a process upon a piece of glass.

**Data:**

Class MyGlass {

Glass glass;

enum processState {unprocessed, processing, donePorcessing};

}

List<MyGlass> glassToBeProcessed;

RobotAgent robot; // Need a reference to the attached robot \*Note that this is currently NOT IN USE with the version 2 design, a reference to the conveyorFamily (and subsequently the popUp) are used instead.

MachineType processType; // Designates what process this machine performs

**Messages**:

msgProcessGlass(Glass g) {

gTBP.add(new MyGlass(g, unprocessed));

stateChanged();

}

// Transducer specific message

msgDoneProcessingGlass(Glass g) {

if ( glass in gTBP s.t. glass.glass.id == g.id) then

glass.processState = processState.doneProcessing;

}

**Scheduler:**

if ( g in gTBP s.t. g.processState == processState.unProcessed) then

actProcessGlass(g); return true;

if ( g in gTBP s.t. g.processState == processState.doneProcessing) then

actPassGlassToRobot(g); return true;

return false;

**Actions:**

actProcessGlass(MyGlass g) {

t.sendProcessGlassMessage(); // Stub for when the transducer is set up to send a processing message to the animation

g.processState = processState.processing;

}

actPassGlassToRobot(MyGlass g) {

g.glass.recipe.remove(this.processType); // Done with process, does not need to be in recipe anymore

robot.msgDoneProcessingGlass(g.glass);

gTBP.remove(g);

}

**Other Methods:**

N/A

---------------------------------------------------------------------------------------------------------------------

**Name:** ConveyorFamily (Interface)

**Messages**:

msgHereIsGlass(Glass g);

msgPositionFree();

msgDoneProcessingGlass(Glass g, int machineIndex);

---------------------------------------------------------------------------------------------------------------------

**Name:** ConveyorFamilyImp

**Description:** Will act as a wrapper class for a set of conveyors, sensors, and pop-ups. It will also contain a reference to robots and machines through its components

**Data:**

ConveyorFamily nextCF; // reference to the next ConveyorFamily – this could even be the final truck at the end of the line

ConveyorFamily prevCF; // reference to the previous conveyor family, will be NULL if it does not exist

ConveyorAgent conveyor;

List<SensorAgent> sensors; // Will hold all of the sensors of different types in one place – adds to the modularity of the system

PopUpAgent popUp;

**Messages**:

msgHereIsGlass(Glass g) {

if ( s in sensors s.t. s.type == “entry”) then

s.msgHereIsGlass(g);

}

msgPositionFree() {

conveyor.msgPositionFree();

}

msgDoneProcessingGlass(Glass g, int machineIndex) {

popUp.msgDoneProcessingGlass(g);

}

---------------------------------------------------------------------------------------------------------------------