Tim Righettini

3/31/13

Professor Wilczynski

CSCI-201

**Factory Design Document: Version 1**

**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:**

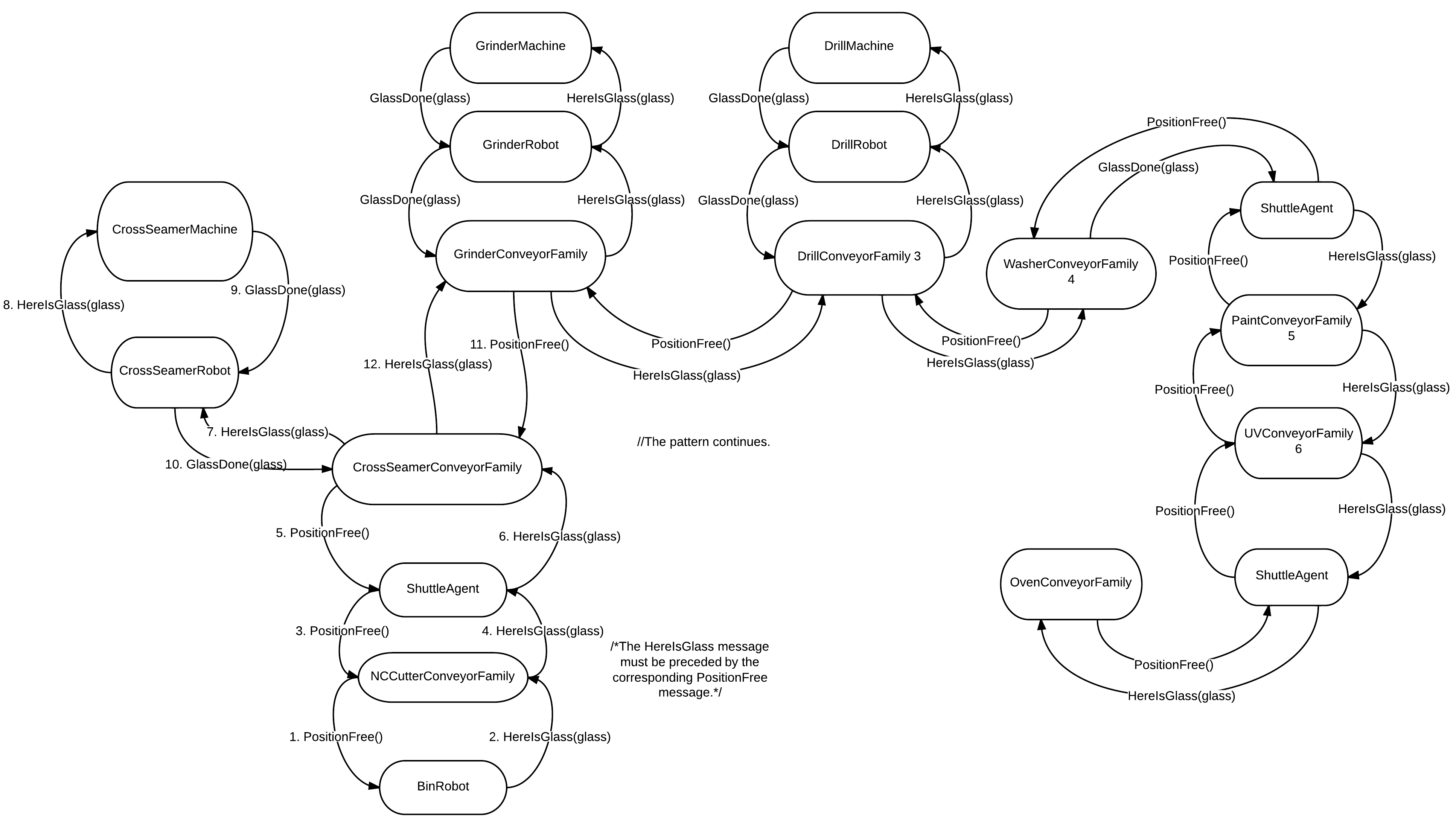
**\*****or t.e. = There Exists**

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

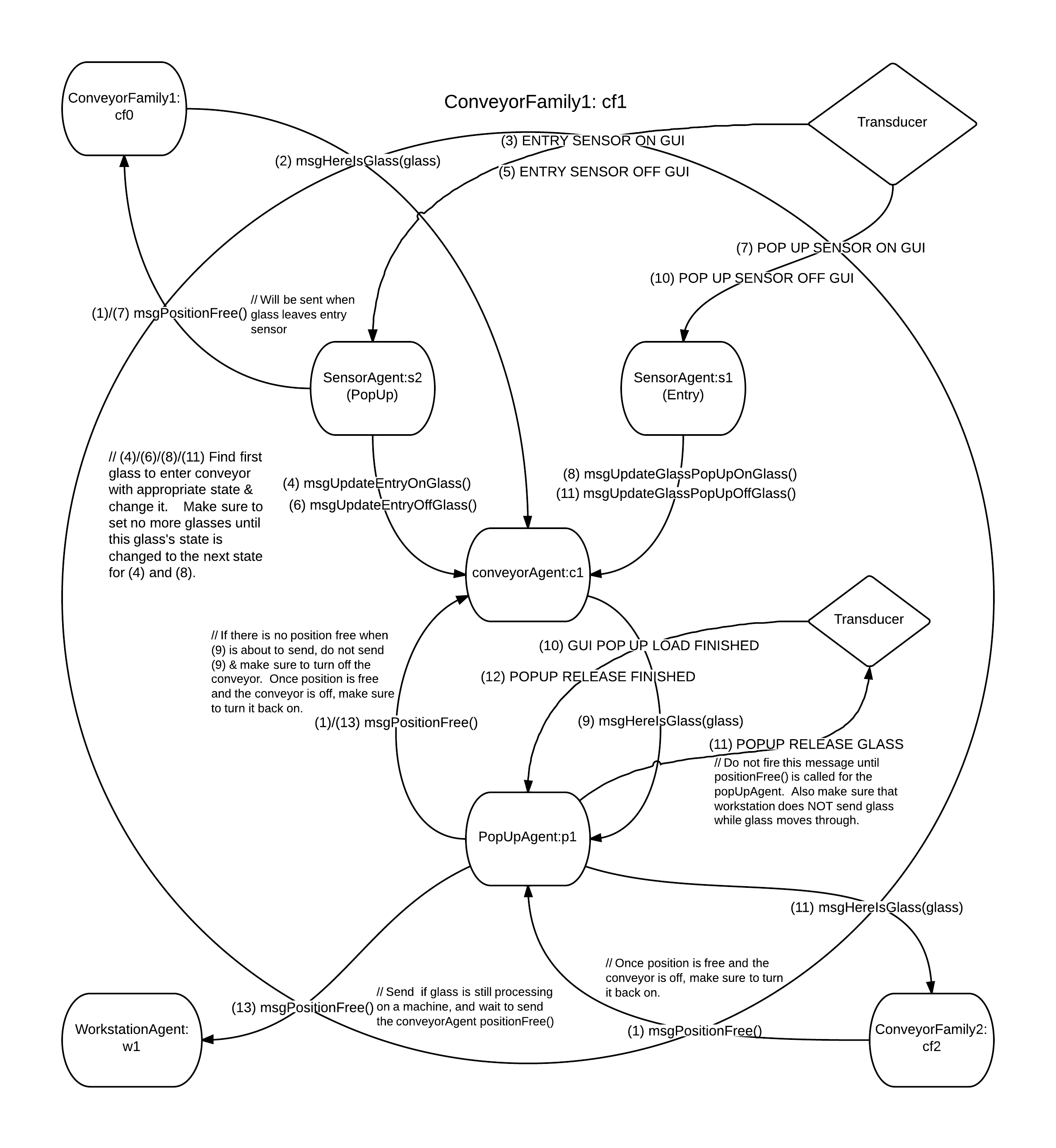
**\*****or f.a. = For All**

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

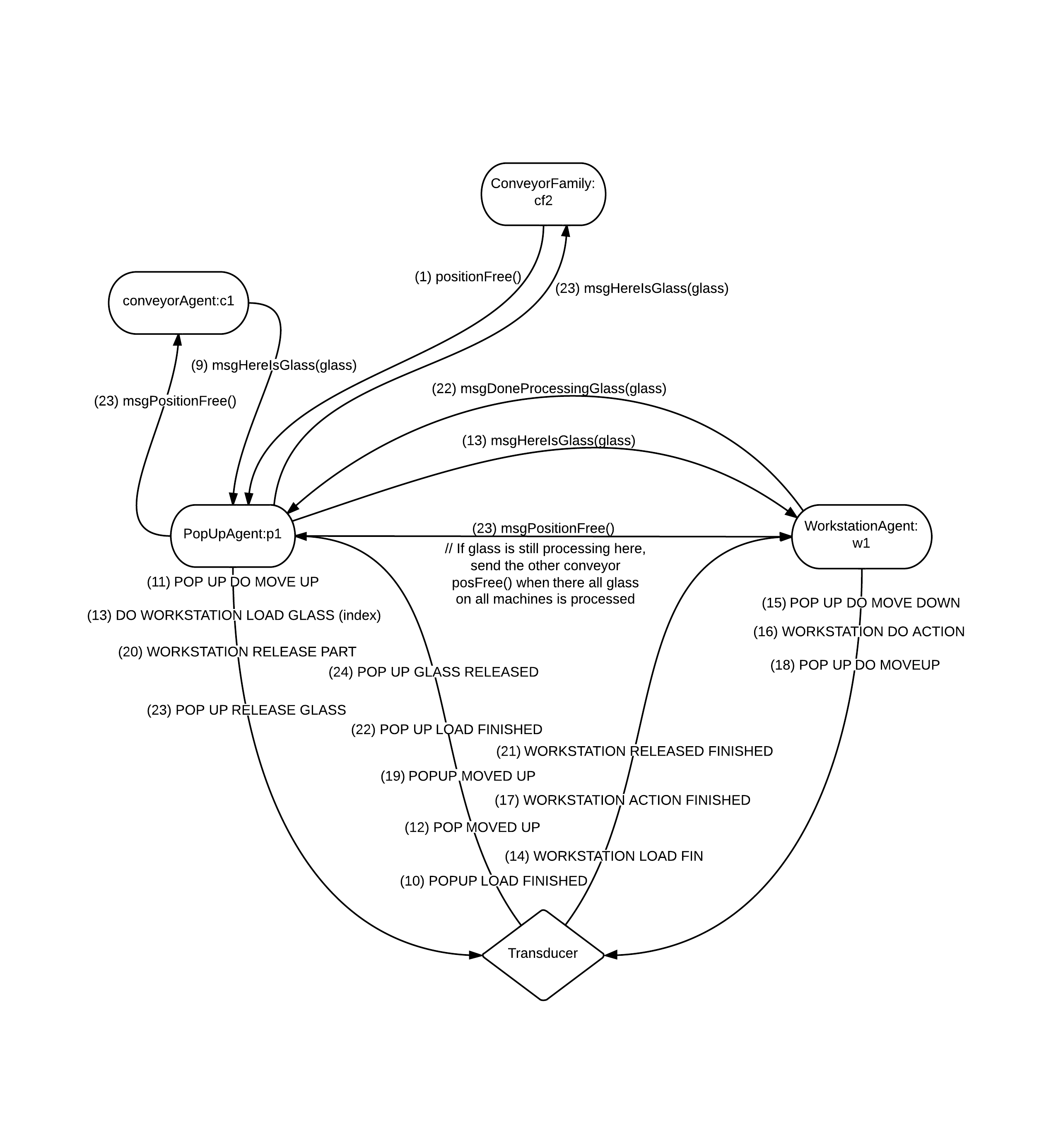
**Normative Interaction Diagram: Team**



**Normative Interaction Diagram: Team** includes the simple normative scenario under one hood. This scenario will be addressed in v1 and v2.

**Normative Interaction Diagram 1: Individual (Tim) – Version 2**

**Normative Interaction Diagram 1: Individual (Tim) Version 2** includes the normative scenario for the agent messaging system to be used for the conveyor family when a piece of glass does NOT need processing

**Normative Interaction Diagram 2: 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 includes everything from the previous normative interaction diagram, but this included the glass processing transducer messaging and other things related to it.

**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)

// Theoretically, the sensor agent interface should not have anything within it

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

**Name:** SensorAgent

**Description:** Will detect if a piece of glass has entered, exited, or on a popup for any given set of conveyors. Only one instance of this agent will be needed to check for which GUI sensor was hit, because this agent will watch for transducer messages from both GUI sensors

**Data:**

ConveyorFamily cf; // Reference to the current conveyor family

int guiIndexEnterSensor; // Needed to communicate with the transducer

int guiIndexPopUpSensor; // Needed to communicate with the transducer

**Messages**:

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

msgUpdateGlassEntrySensorEnter() {

cf.getConveyor().msgUpdateGlass(ConveyorEvent.onEntrySensor);

}

msgUpdateGlassEntrySensorExit() {

cf.getConveyor().msgUpdateGlass(ConveyorEvent.offEntrySensor);

}

msgUpdateGlassPopUpSensorEnter() {

cf.getConveyor().msgUpdateGlass(ConveyorEvent.onPopUpSensor);

}

msgUpdateGlassPopUpSensorExit() {

cf.getConveyor().msgUpdateGlass(ConveyorEvent.offPopUpSensor);

}

// All four for these agents will listen for sensor events and forward the appropriate sensor events to the conveyor agents

**Other Methods:**

eventFired(...); // Will catch all of the animation completions and process them accordingly

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

**Name:** Conveyor (Interface)

**Messages**:

msgGiveGlassToConveyor(Glass g);

msgGiveGlassToPopUp(Glass g);

msgPositionFree();

msgUpdateGlass(ConveyorEvent event);

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

**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 {beforeEntrySensor, onEntrySensor, beforePopUpSensor, onPopUpSensor};

}

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

ConveyorFamily cf;

enum conveyorEvent {onEntrySensor, offEntrySensor, onPopUpSensor, offPopUpSensor, popUpFree} // Will be used for states of glass on the conveyor

List<ConveyorEvent> events; // Used to hold all of the sensor events

boolean conveyorOn; // Is the conveyor moving or not?

int guiIndex; // Needed to communicate with the transducer

**Messages**:

msgGiveGlassToConveyor(Glass g) {

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

stateChanged();

}

msgPositionFree() {

events.add(ConveyorEvent.popUpFree);

stateChanged();

}

msgUpdateGlass(ConveryorEvent event) { // This will receive all of the events forwarded from the sensor agents

events.add(event);

stateChanged();

}

**Scheduler:**

if (sensorEvents.empty()) return false:

ConveyorEvent e = events.remove(0);

if (e == onEntrySensor && $ g in glassSheets s.t. g.conveyorState == beforeEntrySensor) then

actSetGlassOnEntrySensor(g); return true;

if (e == offEntrySensor && $ g in glassSheets s.t. g.conveyorState == onEntrySensor) then

actSetGlassOffEntrySensor(g); return true;

if (e == onPopUpSensor && $ g in glassSheets s.t. g.conveyorState == beforePopUpSensor) then

actSetGlassOnPopUpSensor(g); return true;

if (e == popUpFree && $ g in glassSheets s.t. g.conveyorState == onPopUpSensor) then

actTurnOnConveyorAndSendGlass(); return true;

if (e == offPopUpSensor && $ g in glassSheets s.t. g.conveyorState == onPopUpSensor) then

actSetGlassOffPopUpSensor(g); return true;

// No event should return false

**Actions:**

actSetGlassOnEntrySensor(MyGlass g) {

g.conveyorState = onEntrySensor;

turnOnConveyorGUI();

}

actSetGlassOffEntrySensor (MyGlass g) {

g.conveyorState = beforePopUpSensor;

cf.prevCF.msgPositionFree();

}

actSetGlassOnPopUpSensor (MyGlass g) {

g.conveyorState = onPopUpSensor;

turnOffConveyorGUI(); // Hack method, will actually be a transducer call in code

cf.getPopUp().msgGiveGlassToPopUp(g.glass);

}

actTurnOnConveyorAndSendGlass () {

turnOnConveyorGUI(); // Hack method, will actually be a transducer call in code

}

actSetGlassOffPopUpSensor (MyGlass g) {

glassSheets.remove(g);

if (glassSheets.size() == 0)

turnOffConveyorGUI();

}

**Other Methods:**

turnOnConveyorGUI() // Will turn on the GUI conveyor if (!conveyorOn)

turnOffConveyorGUI() // Will turn off the GUI conveyor if (conveyorOn)

eventFired(...); // Will catch all of the animation completions and process them accordingly

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

**Name:** PopUp (Interface)

**Messages**:

msgGiveGlassToPopUp(Glass g);

msgGlassDone(Glass g);

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

**Name:** PopUpAgent

**Description:** Will act as a mediator between the conveyor agent and workstation agents to get glass from one to the other

**Data:**

Class MyGlass {

Glass glass;

enum processState {awaitingArrival, unprocessed, processing, doneProcessing, awaitingRemoval};

}

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

WorkstationAgent machine; // Robot reference

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

int machineIndex; // Where the machine is located within the animation

}

MachineType processType; // Will hold what the concurrent workstation agents can process for any given popUp – it is safe to assume that the workstations process the same thing

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

List< MachineCom> machineComs;

conveyorFamily cf; // Conveyor Family reference

boolean passNextCF; // Is it possible to pass to the next conveyor family yet?

int guiIndex; // Needed to communicate with the transducer

**Messages**:

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

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

stateChanged();

}

msgGlassDone(Glass g, int machineIndex){

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

glass.processState = doneProcessing;

else // this is a bug, it should never get here

machineComs(machineIndex).inUse = false;

stateChanged();

}

msgPositionFree() {

passNextCF = true;

stateChanged();

}

**Scheduler:**

if ( g in gTBP s.t. g.processState == processState.awaitingRemoval && passNextCF == true) then

actPassGlassToNextCF(g); return true;

else

return false; // Do not want another piece of glass to collide, so shut the agent down until positionFree() is called

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

actRemoveGlassFromMachine(g); return true;

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

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

actPassGlassToMachine(g, com); return true;

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

if ( com in machineComs s.t. !com.inUse || !g.glass.getNeedsProcessing(processType)) then

actSendForGlass(g); return true;

return false;

**Actions:**

actSendForGlass(MyGlass g) {

// Fire transducer event to move the popUp down here index – make sure to stall the agent until the right time to prevent any weird synchronization issues cf.getConveyor().msgPositionFree();

// Fire transducer event to send glass from conveyor to popUp – wait until event is done

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

g.processState = unprocessed;

else

g.processState = awaitingRemoval;

}

actPassGlassToMachine (MyGlass g, RobotCom com) {

com.inUse = true;

g.processState = processing;

// Fire the PopUp up transducer event index – make sure to stall the agent until the glass arrives to prevent any weird synchronization issues

com.machine.msgHereIsGlass(g.glass);

// Machine Load glass transducer events w/right index (can be attained from the machineCom machineIndex – make sure to stall the agent until the glass arrives to prevent any weird synchronization issues

}

actRemoveGlassFromMachine(MyGlass g) {

// Make sure to call Transducer events: Move PopUp up, Machine Release Glass, Move PopUp Down all with the correct timing so nothing is funky

g.processState = awaitingRemoval;

}

actPassGlassToNextCF(MyGlass g) {

cf.nextCF.msgHereIsGlass(g.glass);

// Fire transducer event to release glass index – make sure to stall the agent until the glass arrives to prevent any weird synchronization issues

passNextCF = false;

gTBP.remove(g);

}

**Other Methods:**

eventFired(...); // Will catch all of the animation completions and process them accordingly

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

**Name:** ConveyorFamily (Interface)

**Messages**:

msgHereIsGlass(Glass g);

msgPositionFree();

msgGlassDone (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;

SensorAgent sensors; // Will communicate with all of the GUI sensors

PopUpAgent popUp;

**Messages**:

msgHereIsGlass(Glass g) {

conveyor.msgGiveGlassToConveyor (g);

}

msgPositionFree() {

conveyor.msgPositionFree();

}

msgGlassDone (Glass g, int machineIndex) {

popUp.msgGlassDone(g, machineIndex);

}

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