|  |  |  |  |
| --- | --- | --- | --- |
| **PartsRobotAgent** | | | |
| **Data** | **Messages** | **Scheduler** | **Actions** |
| enum RobotActionState{OFF, RESETTING, GOING\_IDLE, STANDBY, PICKING\_UP\_PARTS, DROPPING\_OFF\_PARTS, UNKNOWN}  enum RobotPartPossessionState{HOLDING\_PARTS, NOT\_HOLDING\_PARTS, UNKNOWN}  enum PartInHandState{HOLDING, READY\_TO\_PUT\_IN\_KIT, DROPPING, UNKNOWN}  enum KitOrderState{DONE, FULL, PARTIALLY\_FILLED, EMPTY, INCORRECT, PARTS\_NOT\_FOUND, UNKNOWN}  enum KitPartOrderState{NEEDED, ACCEPTING, FILLED, MISSING, INCORRECT, UNKNOWN}  enum NestPositionState{HOLDING\_NO\_PART, HOLDING\_GOOD\_PART, HOLDING\_BAD\_PART, PICKUP\_PENDING, UNKNOWN}  private class PartsRobotPartInHand  {  PartType partType  PartInHandState state  public PartsRobotPartInHand(PartType type)  {  this.partType = type  this.state = PartInHandState.HOLDING  }  }  private class PartsRobotLaneModel  {  int laneNum  PartType partType  int nestCapacity  List<NestPositionState> nestContents  boolean noGoodPartsInNest  boolean onlyGoodPartsInNest  public PartsRobotLaneModel(int num type, int nestCap)  {  this.laneNum = num  this.partType = UNKNOWN  this.nestCapacity = nestCap  for(int i = 0; i < this.nestCapacity; i++)  this.nestContents.add(NestPositionState.UNKNOWN)  checkForGoodPartsInNest()  checkIfOnlyBadPartsInNest()  }  private setNestContents(List<NestPositionState> contents)  {  List<NestPositionState> nestContents  = new ArrayList<NestPositionState>();    ∀ NestPositionState q in contents  {  switch(q)  {  GOOD: nestContents.add(HOLDING\_GOOD\_PART)  BAD: nestContents.add(HOLDING\_BAD\_PART)  MISSING: nestContents.add(HOLDING\_NO\_PART)  UNKNOWN: nestContents.add(UNKNOWN)  default: nestContents.add(NestPositionState.UNKNOWN)  }  }    this.nestContents = nestContents;  checkForGoodPartsInNest();  checkIfOnlyBadPartsInNest();  }  private void resetNestContents()  {  ∀ NestPositionState nps in this.nestContents  {  nps = NestPositionState.UNKNOWN  }  checkForGoodPartsInNest();  checkIfOnlyBadPartsInNest();  }  private checkForGoodPartsInNest()  {  ∀ NestPositionState q in nestContents  {  if(q == PartQuality.HOLDING\_GOOD\_PART)  {  noGoodPartsInNest = false  return  }  }  noGoodPartsInNest = true  }  private void checkIfOnlyBadPartsInNest()  {  ∀ NestPositionState q in this.nestContents  {  if(q == HOLDING\_GOOD\_PART or PICKUP\_PENDING  or UNKNOWN)  {  this.onlyBadPartsInNest = false  return  }  }  this.onlyBadPartsInNest = true  }  }  private class PartsRobotKitOrder  {  private class KitPartOrder  {  PartType partType  KitPartOrderState state  public KitPartOrder(PartType type)  {  partType = type  state = KitPartOrderState.NEEDED  }  }  List<KitPartOrder> kitParts  KitOrderState state  int kitStandLocation  public PartsRobotKitOrder(KitOrder kitOrder, int standLoc)  {  ∀ partOrders p in kitOrder  {  this.kitParts.add(new KitPartOrder(p.partType)  }  this.state = KitOrderState.EMPTY  this.kitStandLocation = standLoc  }  private checkKitOrderProgress()  {  int filledPartsCount = 0  ∀ KitPartOrder p in kitParts  {  if(p == MISSING || INCORRECT)  {  this.state = INCORRECT  return  }  else if(p == FILLED)  {  filledPartsCount++  }  }  if(filledPartsCount == kitParts.size())  this.state = FILLED  else if(filledPartsCount > 0 && filledPartsCount <  kitParts.size())  this.state = PARTIALLY\_FILLED  }  }  KitRobotAgent myKitRobotAgent CameraSystemAgent myCameraSystemAgent  CellManagerAgent myManagerAgent  PartsRobotGui myPartsRobotGui  RobotActionState myActionState  RobotPartPossessionState myPartPossessionState  boolean faultyRobot  List<PartsRobotLaneModel> myLanes  List<PartsRobotKitOrder> myKitOrders  int myNumGrippers  List<PartsRobotPartInHand> myPartsInHand | msgHereAreUpdatedNestContents(int laneNum, List<PartQuality> nestContents)  {  if(Ǝ lane l in this.myLanes | l.laneNum == laneNum)  l.setNestContents(nestContents)  //stateChanged()  }  msgHereIsANewKitOrderToFill(KitOrder newKitOrder, int kitStandLoc)  {  this.myKitOrders.add(new PartsRobotKitOrder(new  KitOrder, kitStandLoc)  //stateChanged()  }  msgThisIsCurrentKitConfiguration(KitConfig newKitConfig)  {  ∀ laneInfo l in newKitConfig  {  if(Ǝ lane n in this.myLanes | l.laneNum == n.laneNum)  n.partType = l.partType  }  //stateChanged()  }  msgDonePickUpFromLanes(List<PartType> partsInHand, List<NestPosition> pickedPositions)  {  ∀ PartType t in partsInHand  {  this.myPartsInHand.clear()  this.myPartsInHand.add(new PartRobotPartInHand(t)  }  if(this.myPartsInHand is not empty)  this.myPartPossessionState = HOLDING\_PARTS  this.myActionState = RobotActionState.STANDBY    ∀ NestPosition np in pickedPositions  {  update corresponding nestContents for this lane/nest  position to be HOLDING\_NO\_PART  }  ∀ PartsRobotLaneModel l in this.myLanes  {  l.checkIfOnlyBadPartsInNest()  if(l.onlyBadPartsInNest)  {  if(lane.laneNum % 2 == 0)//Even numbered lane  {  l.resetNestContents()  this.myLanes.get((l.laneNum - 1) - 1).resetNestContents()  myCameraSystemAgent.msgDoneTakingGoodPartsFromNest(l.laneNum)  myCameraSystemAgent.msgDoneTakingGoodPartsFromNest(laneNum - 1)  }  else //odd numbered lane  {  lane.resetNestContents();  this.myLanes.get(lane.laneNum).resetNestContents();  myCameraSystemAgent.msgDoneTakingGoodPartsFromNest(l.laneNum)  myCameraSystemAgent.msgDoneTakingGoodPartsFromNest(l.laneNum + 1)  }  }  //stateChanged()  }  msgDonePutPartsInKit(int kitStandLoc, List<PartType>  depositedPartTypes, List<PartType> stillHoldingPartTypes)  {  if(Ǝ PartRobotKitOrder k in this.myKitOrders |  k.kitStandLocation = kitStandLoc)  {  ∀ PartType t in depositedPartTypes  {  if(Ǝ KitPartOrder p in k.kitParts | p.partType)  {  p.state = FILLED  }  }  k.checkKitOrderProgress()  }  if(this.myPartsInHand is empty)  this.myPartPossessionState = NOT\_HOLDING\_PARTS  this.myActionState = RobotActionState.STANDBY  //stateChanged()  }  msgDoneDroppingParts(List<PartType> partsInHand)  {  if(this.myPartsInHand is not empty)  this.myPartPossessionState = HOLDING\_PARTS  else  this.myPartPossessionState = NOT\_HOLDING\_PARTS  this.myActionState = RobotActionState.STANDBY  //stateChanged()  }  msgDoneGoingIdle()  {  this.myActionState = RobotActionState.STANDBY  //stateChanged()  }  msgDoneResetting()  {  this.myActionState = RobotActionState.STANDBY  //stateChanged()  }  msgGoOnline()  {  this.myActionState = RobotActionState.GOING\_IDLE  //stateChanged()  }  msgGoOnStandby()  {  this.myActionState = RobotActionState.STANDBY  //stateChanged()  }  msgDisable()  {  this.myActionState = RobotActionState.OFF  //stateChanged()  } | if(this.myActionState == RobotActionState.RESETTING)  {  reset()  }  if(this.myActionState == RobotActionState.GOING\_IDLE)  {  goIdle()  }  if(Ǝ KitOrder k in this.myKitOrders | k.state = FULL)  {  informKitRobotOfCompletedKit(k.kitStandLocation)  }  if(this.myActionState == RobotActionState.STANDBY)  {  if(this.myPartPossessionState == HOLDING\_PARTS)  {  distributePartsInHand()  }  else if(Ǝ KitOrder k in this.myKitOrders | k.state == EMPTY)  {  gatherParts()  }  } | gatherParts()  {  If there are parts needed for the first PartsRobotKitOrder in  the kit order queue, then select up up to myNumGrippers  parts to pick up  - Determine which lane in this.myLanes has the needed  part type  - Determine which nest positions in this lane have good  parts. Pick one of the good positions.  - If there are no good positions in this nest, look for  another lane with this part type.  - If there is another lane with this type, repeat step  above to find good nest positions  - If there is no other lane with this type, change  your selected parts in the very first step in this  method and start over  If fewer than myNumGrippers are needed and able to be  picked up for the first PartsRobotKitOrder, fill up the  remaining grippers with as many needed parts for the  second kit order as possible using the same method as above  myPartsRobotGui.doPickUpPartsFromLanes(new  NestPosition(laneNum, positionNum))  }  distributePartsInHand()  {  If the first PartsRobotKitOrder in queue needs some or all of  the parts in hand, put the needed parts in this kit.  myPartsRobotGui.doPutPartsInKit(kitStandLoc,  List<PartType>)  If the first PartsRobotKitOrder in queue did not need any of  the parts in hand, but the second PartsRobotKitOrder in  queue does, put the needed parts in this kit.  myPartsRobotGui.doPutPartsInKit(kitStandLoc,  List<PartType>)  **Non normative:** PartsRobot can be faulty/broken such that it doesn't quite get parts into the get when it attempts to do so (GUI side). In this case, the agent thinks it filled these parts in the kit and will report a completed kit even though one or more parts is physically missing. The camera and kit robot handle this and put the kit back on the kitting stand and give PartsRobot the order again so that the parts robot can fill the kit completely. Implementation for this here:  if(faultyRobot)  myPartsRobotGui.doFaultyDistributeParts(partsToDist)  else  myPartsRobotGui.doDistributeParts(partsToDist)  If no PartsRobotKitOrder needed one or more of the parts in  hand, drop these un-needed parts  myPartsRobotGui.dropTheseParts(List<PartType>)  }  informKitRobotOfCompletedKit(int kitStandLoc)  {  if(Ǝ PartRobotKitOrder k in this.myKitOrders |  k.kitStandLocation = kitStandLoc)  k.state = KitOrderState.DONE  this.myKitOrderRobot.msgThisKitIsFull(kitStandLoc)  //stateChanged()  }  reset()  {  myPartsRobotGui.doDropAllParts()  myPartsRobotGui.doGoIdle()  }  goIdle()  {  myPartsRobotGui.doGoIdle()  }  PartsAgentGui API Methods (define in Gui class)  =====================================  doPickUpPartsFromLanes(LIst<>)  doPutPartsInKit(kitStandLoc, List<PartType>)  doDropTheseParts(List<PartType>)  doDropAllParts()  doGoIdle()  doReset() |
|  |  |  |  |

msgYouCanPutThesePartsInKit(int kitStandLoc, List<PartType>

partTypes)

{

if(Ǝ PartRobotKitOrder k in this.myKitOrders |

k.kitStandLocation = kitStandLoc)  
 {

∀ PartType t in partTypes

{

if(Ǝ KitPartOrder p in k.kitParts | p.partType = t &&

p.state = NEEDED)

{

p.state = ACCEPTING

}

}

}

}