APPENDIX A

The Simulation Program Generator designed provides objects which represent entities within a hypothetical flexible assembly system. This appendix provides a description of the geometry of the robot and the conveyor used in the design of the SPG. The links of the robot and the elements of the conveyor are identified in Section A.1. The geometric data of these elements are described in the Section A.2.

A.1 Geometric Drawings

This section provides the drawings of the links of the robot and the conveyor bed and legs.
LD
5655
1855
1990
G 847
Appendix
C.2
Figure 1. Robot Link0

UNITS = feet
Figure 2. Robot Link1 and Link2
Figure 5. Conveyor Bed

UNIT $= \text{feet}$
UNIT S= feet

Figure 6. Conveyor Leg
A.2 Geometric Data

This section describes the geometric data of the links and the conveyor bed and legs. A description of the data is given in an earlier section.

A.2.1 Link0 Data

1 0 0 0
2 1 0 0 0
3 1 0 1 0
4 0 1 0
5 0 1 1
6 1 0 1 1
7 0 0 1
8 1 0 0 1
1 1 2
10 0 0
2 2 3
0 1 0
3 4 3
10 0 0
4 1 4
0 1 0
5 5 4
0 0 -1
6 6 5
-10 0 0
7 6 3
0 0 -1
8 7 5
0 1 0
9 1 7
0 0 1
10 8 7
-10 0 0
11 6 8
0 -1 0
12 8 2
0 0 -1
1 0 0 -1
4 3 -2 -1
2 0 1 0
-5 -6 7 -3
3 0 0 1
-10 -11 6 -8
4 0 -1 0
1 -12 10 -9
5 -1 0 0
9 8 5 -4
6 1 0 0
2 -7 11 12
0 0 0 1 0 0 0 1 0
10 1 1 -10 0 0 0 0 -1
10 1 1 -10 0 0 0 -1 0
0 0 0 1 0 0 0 0 1

APPENDIX A
A.2.2 Link1 Data

1000
2100
3190
4090
5091
6191
7001
8101
112
100
223
090
343
100
414
090
554
00-1
665
-100
763
00-1
875
090
917
001
A.2.3 Link2 Data

1000
2100
3110
4010
501-9
611-9
700-9
810-9
112
100
223
010
343
100
414
010
554
009
665
-100
763
009
875
010
917
00-9
A.2.4 Link3 Data

1000
2200
3210
4010
5010.1
6210.1
7000.1
8200.1
112
200
223
010
343
200
414
010
554
00-0.1
665
-200
763
00-0.1
875
010
917
000.1
| 10 8 7 |
| -2 0 0 |
| 11 6 8 |
| 0 -1 0 |
| 12 8 2 |
| 0 0 -0.1 |
| 1 0 0 -1 |
| 4 3 -2 -1 |
| 2 0 1 0 |
| -5 -6 7 -3 |
| 3 0 0 1 |
| -10 -11 6 -8 |
| 4 0 -1 0 |
| 1 -12 10 -9 |
| 5 -1 0 0 |
| 9 8 5 -4 |
| 6 1 0 0 |
| 2 -7 11 12 |
| 0 0 0 2 0 0 0 1 0 |
| 2 1 0.1 -2 0 0 0 0 -0.1 |
| 2 1 0.1 -2 0 0 0 -1 0 |
| 0 0 0 2 0 0 0 0 0.1 |
| 0 0 0 0 1 0 0 0 0.1 |
| 2 1 0.1 0 -1 0 0 0 -0.1 |
A.2.5 Link4 and Link5 Data

1 0 0 0
2 0 1 0 0
3 0 1 0 5 0
4 0 0 5 0
5 0 0 5 1
6 0 1 0 5 1
7 0 0 1
8 0 1 0 1
1 1 2
0 1 0 0
2 2 3
0 0 5 0
3 4 3
0 1 0 0
4 1 4
0 0 5 0
5 5 4
0 0 -1
6 6 5
-0.1 0 0
7 6 3
0 0 -1
8 7 5
0 0 5 0
9 1 7
0 0 1
10 8 7
-0.1 0 0
11 6 8
0 -0.5 0
12 8 2
0 0 -1
1 0 0 -1
4 3 -2 -1
2 0 1 0
-5 -6 7 -3
3 0 0 1
-10 -11 6 -8
4 0 -1 0
1 -12 10 -9
5 -1 0 0
9 8 5 -4
6 1 0 0
2 -7 11 12
0 0 0 0.1 0 0 0 0.5 0
0.1 0.5 1 -0.1 0 0 0 -1
0.1 0.5 1 -0.1 0 0 -0.5 0
0 0 0 0.1 0 0 0 1
0 0 0 0.5 0 0 0 1
0.1 0.5 1 0 -0.5 0 0 0 -1
A.2.6 Conveyor Bed Data

1 0 0 0
2 4 0 0 0
3 4 0 1 0
4 0 1 0
5 0 1 4
6 4 0 1 4
7 0 0 4
8 4 0 0 4
1 1 2
4 0 0 0
2 2 3
0 1 0
3 4 3
4 0 0 0
4 1 4
0 1 0
5 5 4
0 0 -4
6 6 5
-4 0 0 0
7 6 3
0 0 -4
8 7 5
0 1 0
9 1 7
0 0 4
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10 8 7</td>
<td>-40 0 0</td>
<td>11 6 8</td>
<td>0 -1 0</td>
<td>12 8 2</td>
<td>0 0 -4</td>
</tr>
<tr>
<td>1 0 0 -1</td>
<td>4 3 -2 -1</td>
<td>2 0 1 0</td>
<td>-5 -6 7 -3</td>
<td>3 0 0 1</td>
<td>-10 -11 6 -8</td>
</tr>
<tr>
<td>4 0 -1 0</td>
<td>1 -12 10 -9</td>
<td>5 -1 0 0</td>
<td>9 8 5 -4</td>
<td>6 1 0 0</td>
<td>2 -7 11 12</td>
</tr>
<tr>
<td>0 0 0 40 0 0 0 1 0</td>
<td>40 1 4 -40 0 0 0 0 -4</td>
<td>40 1 4 -40 0 0 0 0 -1 0</td>
<td>0 0 0 40 0 0 0 4</td>
<td>0 0 0 0 1 0 0 4</td>
<td>40 1 4 0 -1 0 0 0 -4</td>
</tr>
</tbody>
</table>
A.2.7 Conveyor Leg Data

1000
2100
3130
4030
5031
6131
7001
8101
112
100
223
030
343
100
414
030
554
00-1
665
-100
783
00-1
875
030
917
001
<table>
<thead>
<tr>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 8 7</td>
<td>-1 0 0</td>
<td>11 6 8</td>
<td>0 -3 0</td>
<td>12 8 2</td>
<td>0 0 -1</td>
<td>1 0 0 -1</td>
<td>4 3 -2 -1</td>
</tr>
<tr>
<td>2 0 1 0</td>
<td>-5 6 7 -3</td>
<td>3 0 0 1</td>
<td>-10 -11 6 -8</td>
<td>4 0 -1 0</td>
<td>1 -12 10 -9</td>
<td>5 -1 0 0</td>
<td>9 8 5 -4</td>
</tr>
<tr>
<td>6 1 0 0</td>
<td>2 -7 11 12</td>
<td>0 0 0 1 0 0 0 3 0</td>
<td>1 3 1 -1 0 0 0 0 -1</td>
<td>1 3 1 -1 0 0 0 -3 0</td>
<td>0 0 0 1 0 0 0 0 1</td>
<td>0 0 0 0 3 0 0 0 1</td>
<td>1 3 1 0 -3 0 0 0 -1</td>
</tr>
</tbody>
</table>

**APPENDIX A**
APPENDIX B

The description of the icons and the construction of the demonstration model is depicted graphically in this appendix.

B.1 Icons

Some of the icons that were designed for the user interface module of the SPG are presented in this section. There were too many icons designed and created to show them in this document.
Figure 7. Idle Conveyor Icon
Figure 8. 25% Full Conveyor Icon
Figure 9. Idle Robot Icon
Figure 11. Part Icon
Figure 12.  Part Enter Icon
Figure 13. Part Exit Icon
Figure 15. Assemble Icon
Figure 16. Collision Check Icon
B.2 Construction and Execution of Demonstration Model

This section graphically describes the construction and execution of the demonstration model. Again, only the major steps involved in the construction or execution of the model is depicted.
Figure 17. Initial SPG Screen
Figure 18. Retrieving Objects From Model Base
| Figure 19. Placing Retrieved Objects On Screen |
Figure 20. Creating and Placing Other Objects
Figure 21. Connecting Icons
APPENDIX C

C.1 Master Simulation Class

Object subclass: #Simulation
  instanceVariableNames: 'partDict palletNumber subSystems
    stats geoData statisticsPane inputString partTypes
    realTime eventChain movesSpeed icon aArray topPane
    iconPane simulationPane iconDict cursorIn timeDisplay
    simTime timePosition timeDisplay iconSelected
    iconsSelectedTillNow`
classVariableNames: 'SubSystemList`
poolDictionaries: '"

‘Simulation class methods!

addSS: aSubSystem
  "adds model or object instance to the model base"

  (SubSystemList = nil) ifTrue: [ self initializeSS ].
  (SubSystemList includes: aSubSystem) ifFalse: [ 
    SubSystemList add: aSubSystem ];

initializeSS
  "initializes and creates model base"

  SubSystemList := Set new.;

instanceSS: aSimulator
  | dSystems key allDone dorm yn m |
"retrieves an instance of a model or object from the model base. Returns nil if model base is empty otherwise prompts the names of the objects in the form of a menu for further selection"

(SubSystemList = nil) ifTrue: [ self initializeSS ]
(SubSystemList isEmpty) ifTrue: [
    Menu notify: 'The subsystem library is empty.'
    .
].

SubSystemList do: [: each |
    (each = '') ifFalse: [
        dSystems at: (each label) put: each .
    ].
].

key := dSystems pick: 'Model Base'.
(key = nil) ifTrue: [ self .
allDone := false.
[ allDone ] whileFalse: [
    m := Menu labels: 'Create an instance
        Remove this subsystem' lines: #() selectors: #(1 2).
    dorm := m popupAt: (Cursor offset).
    (dorm = nil) ifTrue: [ self ]
    (dorm = 1) ifTrue: [
        key assignSimulator: aSimulator.
        key copyInteract.
        allDone := true
    ].
    (dorm = 2) ifTrue: [
        yn := Menu yesNo:
        'Are you sure? This can only be restored by rebuilding it.'
        (yn = nil) ifFalse: [
            (yn = 1) ifTrue: [
                SubSystemList remove: key
            ]
        ]
        allDone := true
    ]
].

new
"initializes the Simulation Class"

self (super new) initialize!

'Simulation methods!'

activatePane
"Changes the cursor shape to cross hair"
when in the IconPane or Graphpane"

cursorln := true.
CursorManager crossHair change!

addIcon: anObject
| key aString |

"Adds an icon anObject to the iconsSelectedTillNow
dictionary"

aString := anObject name.
-iconsSelectedTillNow includes: anObject) ifTrue: [
    key := iconsSelectedTillNow keyAtValue: anObject.
    iconsSelectedTillNow removeKey: key
]
iconsSelectedTillNow at: aString put: anObject!

addSubSystem: aSubSystem
"Adds a aSubSystem object or model to the subSystems set
if not already existing"

[subSystems = nil] ifTrue: [ subSystems := Set new ].
(subSystems includes: aSubSystem) ifFalse: [
    subSystems add: aSubSystem
]

buildup
| entity p |

"Finds the icon entity at the position p selected by the
user and if found returns the object menu of that icon"
p := self wantPick.
entity := self findObjectAt: p.
(entity isNil) ifFalse: [
    self startUp: entity
]

changeToMeters: aPoint
"Changes the position in pixels of icon at aPoint to
meters"

| screenCenter x y nx ny pixelEquals |
screenCenter := 320@240.
x := aPoint x.
y := aPoint y.
x := ((x - (screenCenter x)) * 0.1.
y := ((screenCenter y) - y) * 0.1.
("Point new x: nx; y: ny")

changeToPixels: aPoint
"Changes the position in meters of icon at aPoint
to pixels"

| screenCenter x y nx ny pixelEquals|
screenCenter := 320@240.
x := aPoint x.
y := aPoint y.
xn := (screenCenter x) + (x/0.1).
y := (screenCenter y) + (y/0.1).
((Point new) x: xn; y: ny)!

clipRect
"Returns a rectangle the size and position
of the Graph Pane"
((130@22 extent: 510@455)!)!

clipTimeRect
"Returns a rectangle the size and position
of the time slot"

((5@22 extent: 125@20)!)!

convDataMenu
"Menu for viewing the database of the conveyor"

cPromptMenu
  prompt: 'Conveyor Statistics'
  labels: 'conveyor bed\'conveyor leg' withCrs
  lines: #(1)
  selectors: #(7 8)!

convMenu
| which |

"Pops up the conveyor database menu and determines the conveyor
data to be viewed"

which := self convDataMenu popUpAt: (Cursor offset).
(which isNil) ifFalse: [self whichLinkData: (geoData at: which)
of: which]!

createIns
| stopButton buttonLoc rect subSys sBback subSysC lab |

"Creates a button below the time slot and then requests the
user to select the objects to be stored in the model base.
After the selection is completed the button can be clicked and
the objects are stored."

stopButton := 'Definition complete'.

APPENDIX C 43
buttonLoc := 5@20 + (0@(SysFont charSize y)).
rect := Rectangle origin: buttonLoc extent: ((
    Point new) x: (stopButton size) * (SysFont charSize x));
y: (SysFont charSize y)).
sBback := ColorForm fromDisplay: rect.
stopButton displayAt: buttonLoc.
" subSys := SubSystem newOb: self. "
subSys := SimulationObjects new.
sys editSubsystem: rect sim: self.
sBback display.
subSys graphOn.
lab := Prompter prompt: 'What is the name of this subsystem?'
default: 'a Model'.
(lab = nil) ifTrue: [lab := 'a Model'].
sys label: lab.
sys showIcon.
lab := Menu yesNo: 'Do you want to add this to the subsystem library?'.
(lab = nil) ifFalse: [
    (lab = 1) ifTrue: [
        subSysC := subSys copy.
        Simulation addSS: subSysC.
        subSysC rmNoDetach
    ]
].

deactivatePane

"Changes the shape of the cursor back to its normal form"

cursorIn := false.
CursorManager normal change!

deSchedule: anObject
| tempChain times |

"Removes the all events of the anObject stored in the eventChain"
tempChain := eventChain select: [ : event | ((event actor) ≠ = anObject)].
times := Bag new.
eventChain do: [ : each |
    ((each actor) = anObject) ifTrue: [times add: each]
].
eventChain := tempChain asSortedCollection.
times!

displayTime

"Displays the time slot at the left upper corner of the screen"

Display
    fill: self clipTimeRect
rule: Form over
  mask: Form white.
timeDisplay := 'Time: ', (simTime asFloat printString), '
  timeDisplay displayAt: 5@20!

drawIconsIn: aRectangle
  | aRect |

  "Draws the icons in the Icon Pane surrounded by a rectangle"

  Display white: aRectangle.
anArray := self iconArray.
  1 to: 8 do: [: index |
    aRect := (anArray at: index) copy
    moveBy: aRectangle origin.
    icon := self prepareIconDictionary.
    (icon at: index) displayAt: aRect origin.
    Display border: aRect!"

eventChain
  "Returns the eventChain"

  \eventChain!"

eventChain: aSortedCollection
  "Sets the value of the eventChain"

  eventChain := aSortedCollection!"

findIcon: index
  "Depending upon the icon selected in the Icon Pane, a corresponding instance is created of that object."

  (index = 1) ifTrue: [self selectRobot].
  (index = 2) ifTrue: [self selectConveyor].
  (index = 3) ifTrue: [self newOb: Part].
  (index = 4) ifTrue: [self newOb: Grasp].
  (index = 5) ifTrue: [self newOb: Enter].
  (index = 6) ifTrue: [self newOb: Exit].
  (index = 7) ifTrue: [self newOb: Transfer].
  (index = 8) ifTrue: [self newOb: Assemble]."

findObjectAt: aPoint
  "Scan the iconsSelectedTillNow dictionary to find the icon at aPoint. The position aPoint was selected by the user"

  | key founds list arr which m |
  (iconsSelectedTillNow = nil) ifTrue: [\nil].
  founds := Set new.
  iconsSelectedTillNow do: [: each |
    (each isMemberOf: SimulationObjects) ifFalse: [

APPENDIX C 45
key := each iconPosition.
(key = nil) ifFalse: [
  ((key <= aPoint) & (aPoint <= (key + 30))) ifTrue: [
    founds add: each
  ]
]
]

((founds size) = 1) ifTrue: [
  founds do: [ : each | %each ]
].
((founds size) = 0) ifTrue: [%nil]

graph: aRectangle
| aForm |

"Creates the Graph Pane of the size equal to aRectangle"

aForm := ColorForm
  width: aRectangle width
  height: aRectangle height.
aForm displayAt: aRectangle origin.
iconSelected := Simicons new.
%aForm!

iconArray
| aRect curRect |

"Sets the icons on the Icon Pane in rectangles"

aRect := 30@90 extent: 30@30.
anArray := Array new: 8.
0 to: 3 do: [: row |
  curRect := aRect copy moveBy: 0 @ (35*row).
  0 to: 1 do: [: column |
    curRect := curRect copy moveBy: (35*column) @ 0.
    anArray at: (row*2 + column + 1)
      put: curRect].
%
anArray!

icons: aRectangle
| aForm |

"Creates the Icon Pane the size of aRectangle and draws the icons in it"

aForm := ColorForm
  width: aRectangle width
  height: aRectangle height.
aForm displayAt: aRectangle origin.
self drawiconsIn:aRectangle.
self displayTime.
initialize

"Initializes the instance variables of the Simulation Class
and creates some data structures"

simTime := 0.
iconsSelectedTillNow := Dictionary new.
realTime := 0.
eventChain := SortedCollection new: 1000.
self initializePartTypes.
self manipStart: 'manip2'.
self getPart!

initializePartTypes

"Creates the partTypes dictionary and sets the values of the
part types in it"

(partTypes = nil) ifTrue: [ partTypes := Dictionary new ].
(partTypes includesKey: 'Part') ifFalse: [ 
  partTypes at: 'Part'
  put: (PartType newPartType: 'Part'
  icons: PartForm runBy: self)
].
(partTypes includesKey: 'Part2') ifFalse: [ 
  partTypes at: 'Part2'
  put: (PartType newPartType: 'Part2'
  icons2: PartForm2 runBy: self)
].
(partTypes includesKey: 'Part3') ifFalse: [ 
  partTypes at: 'Part3'
  put: (PartType newPartType: 'Part3'
  icons3: PartForm3 runBy: self)
] !

initWindowSize

"Returns the size of the screen"

(Display extent!

input

"Returns the statistics of the objects"

(stats contents!

manipStart: fileName

| array1 array2 array3 array4 |

"Reads the data of the robot and the conveyor and stores it
into the geoData database"
array1 := Array new: 12.
array2 := Array new.
array3 := Array new.
array4 := Array new.
geoData := Dictionary new.
array1 := self readAsChar: fileName.
array3 := self readAsChar: 'convstat'.

7 to: 12 do: [:i |
    array2 := self readDataOf: (array1 at: i).
    geoData at: (i - 6) put: array2].

1 to: 2 do: [:i |
    array4 := self readDataOf: (array3 at: i).
    geoData at: (i + 6) put: array4].

menu
   "Returns the Simulation Menu"

   ☞PromptMenu
   prompt: 'Simulation Menu'
   labels: "Execute|Build Up|Reset Time|View database|Save Instance|Retrieve Instance|Redraw|Print Screen" withCrs lines: #( )
   selectors: #( simulate buildup resetTime viewData createIns retrievalns reDraw prtScreen )

movesSpeed
   "Returns the animation speed of the part type"

   ☞movesSpeed!

movesSpeed: anInteger
   "Sets the animation speed of the part to anInteger"

   movesSpeed := anInteger!

newOb: aClass
   "Creates an instance of the icon of the object"

   (aClass newIcon: self) graphicsOn!

newPart: aString
   "Initializes the part database and adds a part type to it if it does not exist"

   self initializePartTypes.
   (partTypes includesKey: aString) ifTrue: [
     ☞((partTypes at: aString) newPart)
   ]
ifFalse: [  
    Menu notify: ('Part ', aString, ' is unknown. Creating a  
    "Part")).  
    c((partTypes at: 'Part') newPart)  
]!

objectCount
"Returns the number of objects in the iconsSelectedTillNow  
database"

(iconsSelectedTillNow = nil) ifTrue: [c0].  
c((iconsSelectedTillNow size))

open
"Opens the Top Pane, the Icon Pane and the Simulation Pane.  
Sets the icons in the Icon Pane and Top Pane is made the  
controller."

(topPane := TopPane new)  
    label: 'HETEROGENEOUS SIMULATION MODELING';  
    model: self;  
    yourself.  
topPane addSubpane:  
    (iconPane := SimIconPane new  
     model: self;  
     name: #icons;;  
     change: #findIcon;;  
     iconArray: self iconArray;  
     framingRatio: (0@0 extent: 1/5 @ 1)).  
topPane addSubpane:  
    (simulationPane := GraphPane new  
     menu: #menu;  
     model: self;  
     name: #graph;;  
     framingRatio: (1/5 @ 0 extent: 4/5 @ 1)).  
topPane dispatcher open scheduleWindow!

openStatsWin
| topPane |

"Opens the statistics pane"

statisticsPane := TopPane new label: 'Statistics'.  
statisticsPane addSubpane:  
    (TextPane new  
     model: self;  
     name: #input).  

statisticsPane dispatcher open scheduleWindow!

palletNumber
palletNumber
palletNumber: anInteger

palletNumber := anInteger!

partDict
"Returns the partDict"

partDict!

partDict: aDictionary
"Sets the value of partDict to aDictionary"

partDict := aDictionary!

partTypes

partTypes!

placeIcon: aPosition
"Replaces the mouse cursor by the icon. After the click icon is displayed at aPosition"

iconSelected placeAtMouse.
iconSelected displayAt: aPosition!

prepareIconDictionary
"Sets the icons in the Icon Pane into a iconDict dictionary"

iconDict := Dictionary new.
iconDict at: 1 put: (self newObject: RobotMT).
iconDict at: 2 put: (self newObject: ConvMT).
iconDict at: 3 put: (self newObject: PartRaw).
iconDict at: 4 put: (self newObject: GraspForm).
iconDict at: 5 put: (self newObject: EnterForm).
iconDict at: 6 put: (self newObject: ExitForm).
iconDict at: 7 put: (self newObject: TransferForm).
iconDict at: 8 put: (self newObject: AssembleForm).
iconDict do: [: each | each offset: 0@0].
FreeDrawing pictureDictionary do: [: each |
 each offset: 0@0].
iconSelected := Simicons new.
iconSelected forms: iconDict.
iconSelected inOffset: (0.1@0.1).
iconSelected outOffset: (1@1).

prntScreen
HPLaserjet printScreen!
pullOutOf: anObject
    | key |
    "Removes the object anObject from the iconsSelectedTillNow database"
    (iconsSelectedTillNow includes: anObject) ifTrue: [
        key := iconsSelectedTillNow keyAtValue: anObject.
        iconsSelectedTillNow removeKey: key
    ]!

redraw
    "Erase the drawing form to white.
     pane form fill: Form white.
     pane showWindow"
    Display
        fill: self clipRect
        rule: Form over
        mask: Form white.
    FreeDrawing pictureDictionary do: [: each |
        each offset: 0@0].
    ScreenDispatcher simulate!

removeSubSystem: aSubSystem
    "Removes the model aSubSystem from the model base"
    (subSystems = nil) ifTrue: [#nil].
    (subSystems includes: aSubSystem) ifTrue: [
        subSystems remove: aSubSystem
    ]!

resetTime
    "Resets the time and each object in the model being executed.
     Also the eventChain times are updated"
    (iconsSelectedTillNow = nil) ifFalse: [
        iconsSelectedTillNow do: [: each |
            each resetStats ]
    ].
    eventChain do: [: each |
        each time: ((each time) - simTime)
    ].
    simTime := 0.
    self displayTime!

retrieveIns
    "Retrieves an instance of an object from the model base"
    Simulation instanceSS: self!
robotDataMenu
   "Menu for viewing the statistics of the robot links"

@PromptMenu
   prompt: 'Robot Statistics'
   labels: 'link0 \ link1 \ link2 \ link3 \ link4 \ link5'
   withCr
   lines: #()
   selectors: #(1 2 3 4 5 6)!}

robotmenu
   | which |
   "Pops up the Robot Statistics Menu and the determines which
link was selected"
   which := self robotDataMenu popUpAt: (Cursor offset).
   (which isNil) ifFalse: [self whichLinkData: (geoData at: which)
of: which]!

scheduleEvent: anObject at: aNumber for: aName
   "Schedules the event by placing it on the eventChain at time
aNumber"
   eventChain add: (Event newFor: anObject at: aNumber partName:
aName)!

selectConveyor
   | which |
   "Pops up the Abstraction Level Menu and determines which level
was selected. Then creates an instance of a conveyor at that
abstraction level"
   which := (self symKinGeoMenu) popUpAt: 180@90.
   (which notNil) ifTrue: [
      (which = 'symbolic') ifTrue: [self newOb: Conveyor]
      ifFalse: [
         (which = 'kin') ifTrue: [
            self newOb: Kin Conveyor]
         ifFalse: [self newOb:
                     GeoConveyor].
   ].

selectRobot
   | which |
   "Pops up the Abstraction Level Menu and determines which level
was selected. Then creates an instance of a robot at that
abstraction level

which := (self symKinGeoMenu) popUpAt: 180@60.
(which notNil) ifTrue: [
  (which = 'symbolic') ifTrue: [self newOb: Robot]
  ifFalse: [
    (which = 'kin') ifTrue: [
      self newOb: KinRobot]
    ifFalse: [self newOb: GeoRobot].
  ].
].

simTime

\$!simTime!

simulate

| t p lastActualTime |

'This method controls the simulation of the model. The events are removed from the eventChain and then the executed according to their execution time'

(eventChain isEmpty) ifTrue: [\$nil].
[((Menu thereWasAClick) not) &
((eventChain isEmpty) not)] whileTrue:

  t := eventChain removeFirst.
  simTime := (t time).
  self displayTime.
  " (simTime = 0) ifFalse: [" (t actor isMemberOf: Enter) ifTrue: [eventChain do: [: each |
    (each actor isMemberOf: Enter) ifTrue: [((each time) = (t time)) ifFalse: [(t actor) doit: (t partName)]]
    ifTrue: [(t actor) schedule: ((t actor) arrivalProbabilityDict at: (t partName)). FreeDrawing l].
  ]].
  ifFalse: [FreeDrawing l.
    (t actor) doit: (t partName)
  ]."
  " (t actor) doit: (t partName)"
  " (simTime = 0) ifFalse: [(t actor) doit: (t partName)]
].
p := Cursor offset.
\$p!
startup: anObject
    | which |

    "Controls the Simulation Menu"

    (anObject = nil) ifFalse: [
        which := (anObject menu) popUpAt: Cursor offset.
        "(which = 'finish') ifFalse: ['anObject menuResponse:
        which']."
    ]!

symKinGeoMenu

    "The Abstraction Level Menu"

¢PromptMenu
    prompt: 'Which Type?'
    labels: 'Symbolic\Kinematic\Geometric' withCrs
    lines: #()
    selectors: #('symbolic' 'kin' 'geo')!

topPane
    ¤topPane!

viewData
    | which |

    "Pops up the DataBase Menu and then determines which option
    was selected (whether the robot or conveyor)"

    which := self viewDataMenu popUpAt: (Cursor offset).
    (which isNil) ifFalse: [
        (which = 'robotmenu') ifTrue: [self robotmenu]
        ifFalse: [self convmenu].
    ]!

viewDataMenu
    "The DataBase Menu"

    ¢PromptMenu
        prompt: 'Select one'
        labels: 'Robot\Conveyor' withCrs
        lines: #()
        selectors: #('robotmenu' 'convmenu')!

waitPick
    |release c pos|
"Stops execution until user clicks left mouse button. Returns location of mouse cursor at time button is released. MouseEvent is set to false."

release := false,
MouseEvent := false,
[release] whileFalse: [
  c := Terminal read.
  (MouseEvent) ifTrue: [
    c := Terminal read.
    ((c asciiValue) = 1) ifTrue: [
      release := true.
      pos := Cursor offset
    ].
    MouseEvent := false
  ].
] pos!

whichLinkData: anArray of: aNumber
  | x pt1 pt2 pt3 pt4 pt5 pt6 pt7 pt8 ptMatrix |

  "Depending upon the link selected from the DataBase Menu the statistics are calculated and displayed on a DataBase Pane"

  pt1 := Array new.
  pt2 := Array new.
  pt3 := Array new.
  pt4 := Array new.
  pt5 := Array new.
  pt6 := Array new.
  pt7 := Array new.
  pt8 := Array new.
  ptMatrix := Matrix new: 8@4.
  x := Geometric new.
  stats := WriteStream on: String new.

  ptMatrix := x matrixFromArray: anArray from: 1 size: 8@4.
  pt1 := ptMatrix rowAt: 1.
  pt3 := ptMatrix rowAt: 3.
  pt5 := ptMatrix rowAt: 5.
  pt8 := ptMatrix rowAt: 8.

  stats
    nextPutAll: 'CONSULT THESIS DOCUMENT FOR DETAILS';
    cr;
    cr;
    nextPutAll: 'Coordinates of Link: ',(aNumber printString);
    cr;
    cr;
nextPutAll: ('Point: ', (pt1 printString));
nextPutAll: ('Point: ', (pt2 printString));
nextPutAll: ('Point: ', (pt3 printString));
nextPutAll: ('Point: ', (pt4 printString));
nextPutAll: ('Point: ', (pt5 printString));
nextPutAll: ('Point: ', (pt6 printString));
nextPutAll: ('Point: ', (pt7 printString));
nextPutAll: ('Point: ', (pt8 printString));
self openStatsWin! ;
C.2 SimulationObject Class

Object subclass: #SimulationObjects
  instanceVariableNames: 'numberReceivedDict partSet probabDict
  scheduleListDict formDict dimensionDict tempEventChain
  positionInMeters refIcon refPoint subSystems components
  detailedIcon Showing previousObjDict stats statisticsPane
  waitingToShowAccept label name itsSimulator nextObjects
  previousObjects numberReceived numberReceived1
  numberReceived2 numberReceived3 numberReceived4
  graphicsOn graphicsOff iconSelected '
  classVariableNames: '
  poolDictionaries: ''

!'SimulationObjects class methods!

formList

  ☞FormList!

initWithList
  "Initializes and creates the PartList, FormList and
  ScheduleList databases"

  PartList := nil.
  (PartList = nil) ifTrue: [PartList := Dictionary new].
  (FormList = nil) ifTrue: [FormList := Dictionary new].
  (SchedulePlanList = nil) ifTrue: [SchedulePlanList :=
  Dictionary new].

new
  "Creates an instance of the SimulationObject Class"

  ☞super new initialize!

newIcon: aSimulation

  ☞((self new) runBy: aSimulation)!

!SimulationObjects methods!

accept: aPart
  "Each subclass of SimulationObject has an accept: aPart method
  which is executed when a message is sent to it"

  self subclassResponsibility!
assignSim: aSimulator

itsSimulator := aSimulator!

assignSimulator: aSimulator

itsSimulator := aSimulator.
components do: [: each | each assignSim: aSimulator ]!

attaches: components dict; cd copy: copyDict
  | nx eachkey keynx |
  "After the objects are retrieved from the model base they are
  attached to each other depending upon their stored version in
  components dictionary"
  nx := self nexts.
eachkey := cd keyAtValue: self.
nx do: [: eachnx |
  (components includes: eachnx) ifTrue: [
    keynx := cd keyAtValue: eachnx.
    (copyDict at: eachkey) attachTo: (copyDict at: keynx)
  ]
  ]!

attachTo: anObject

"Objects are attached to other objects according to the Rules
of Interaction. These rules are controlled or managed by this
method. If a connection is possible then, the objects are
connected and an instance of each object is stored in its
connected to object and vice versa"

(nextObjects = nil) ifFalse: [
  anObject previousObjDict remove: (self label) ifAbsent: [.
  iconSelected removePointTo: (nextObjects inPosition).
  nextObjects doesNotReceiveFrom: self ].

(previousObjects isEmpty) ifFalse: [
  (previousObjects includes: anObject) ifTrue: [
    Menu notify: 'Cannot attach back to previous object'.
    nil
  ].
].

(self isMemberOf: Part) ifFalse: [
  (anObject isMemberOf: Enter) ifTrue: [
    Menu notify: 'Only Part icon can be attached to IN'.
    nil
  ].
].

(isSelf isKindOf: Conveyor) ifTrue: [
  (anObject isKindOf: Robot) ifTrue: [}
Menu notify: 'Cannot attach Conveyor to Robot directly'.
  \n  ;nil
  ].
].

(self isKindOf: Robot) ifTrue: [
  (anObject isKindOf: Conveyor) ifTrue: [
    Menu notify: 'Cannot attach Robot to Conveyor directly'.
    ;nil
  ].
].

(self isKindOf: Conveyor) ifTrue: [
  (anObject isKindOf: Conveyor) ifTrue: [
    Menu notify: 'Cannot attach Conveyors directly'.
    ;nil
  ].
].

(self isKindOf: Robot) ifTrue: [
  (anObject isKindOf: Robot) ifTrue: [
    Menu notify: 'Cannot attach Robots directly'.
    ;nil
  ].
].

(self isMemberOf: Part) ifTrue: [
  (anObject isMemberOf: Enter) ifFalse: [
    Menu notify: 'Part icon can only be attached to IN'.
    ;nil
  ].
].

(anObject isMemberOf: Part) ifTrue: [
  Menu notify: 'Cannot attach anything to Part'.
  ;nil
].

(self isMemberOf: Assemble) ifTrue: [
  (anObject isKindOf: Robot) ifFalse: [
    Menu notify: 'Assemble can only be attached to a Robot'.
    ;nil
  ].
].

(anObject isMemberOf: Assemble) ifTrue: [
  Menu notify: 'Cannot attach anything to Assemble icon'.
  ;nil
].

(self isMemberOf: Grasp) ifTrue: [
  ((anObject isKindOf: KinRobot) or: (anObject isKindOf: KinConveyor)) ifFalse: [
    Menu notify: 'Grasp can only be attached to a Kin/Geo

APPENDIX C

59
Robot or Conveyor.

```
黔nil
].

(anObject isMemberOf: Grasp) ifTrue: [
    Menu notify: 'Cannot attach anything to Grasp icon'.
   黔nil
].

(self isMemberOf: Exit) ifTrue: [
    Menu notify: 'Cannot attach Exit icon to anything'.
   黔nil
].

(self isMemberOf: Enter) ifTrue: [
    (anObject isKindOf: Conveyor) ifFalse: [
        (anObject isKindOf: Robot) ifFalse: [
            Menu notify: 'Interaction icons cannot be attached to each other'.
           黔nil
        ].
    ].
].

(self isMemberOf: Transfer) ifTrue: [
    (anObject isKindOf: Conveyor) ifFalse: [
        (anObject isKindOf: Robot) ifFalse: [
            Menu notify: 'Interaction icons cannot be attached to each other'.
           黔nil
        ].
    ].
].

iconSelected addPointTo: (anObject inPosition).
"iconSelected show."
nextObjects := anObject.
```

```
(self isMemberOf: Part) ifFalse: [
    anObject receivesFrom: self.
    self moveit
] ifTrue: [anObject receivesFrom: self]!
```

```
attachToInteract
    | ob nextPoint |

"Requests the user for a response on the connecting objects. Once the request is given then the Rules of Interactions are checked"
(itsSimulator = nil) ifTrue: [黔self].
((itsSimulator objectCount) = 1) ifTrue: [黔self].
nexPoint := self promptClick:
    'Click on object to send parts to'.
oob := itsSimulator indexOfObjectAt: nextPoint.
oob := (oob = nil) ifFalse: [
```
(ob == self) ?False: [self attachTo: ob]
},
itsSimulator startUp: self!

border

iconSelected border!

changeFromPoint: newPoint from: oldPoint
  iconSelected removePointFrom: oldPoint; addPointFrom: newPoint!

changeToPoint: newPoint from: oldPoint
  iconSelected removePointTo: oldPoint; addPointTo: newPoint!

copy
  | ob ec componentDict copyDict counter nx keyn eachkey array |
  "Copies all the instance variables of the objects to be stored
into the model base. Their interactions with other objects is
also stored"

  positionInMeters := 1 @ 1.
  ob := self copySim.
  " ob positionInMeters: positionInMeters. "
  componentDict := Dictionary new.
  copyDict := Dictionary new.
  counter := 1.
  (self isMemberOf: SimulationObjects) ifTrue: [itsSimulator
    eventChain: tempEventChain].
  array := Array new.
  array := components asArray.
  array do: [ : each |
    componentDict at: counter put: each.
    copyDict at: counter put: ec.
    ec positionInMeters: (each positionInMeters).
    counter := counter + 1.
    ec couple: ob.
    ob include: ec
  ].

  componentDict do: [ : each |
    each attaches: components dict: componentDict copy:
      copyDict
  ].
  (detailed) ifTrue: [ob detail] ifFalse: [ob noDetail].
  ob!

copyInteract
  | c |

APPENDIX C
"Starts the copying operation of the objects into the model base"

iconSelected whichOne: (self whichIcon).
iconSelected show.
c := self copy.
c graphicsForCopy.
c placeComponents.
c showIcon!

copyInteractSim

iconSelected show.
(sell copySim) graphicsOn!

copySim

| ob canHandles |

"Copies the instance variables of the SimulationObject Class to the model base"

ob := (self class) newIcon: itsSimulator.
ob label: label.
ob tempEventChain: (tempEventChain copy).
ob fromFileIcon.
\ob!

decouple

subSystems := nil!

detachFrom: anObject

"Detaches objects from one another. All objects except the Transfer object uses this method. It is controlled from the Object Menu"

(nextObjects = nil) ifTrue: [\nil].
(anObject ≠ nextObjects) ifTrue: [\nil ].
anObject previousObject remove: (self label) ifAbsent: [].
iconSelected removePointTo: (nextObjects inPosition).
anObject doesNotReceiveFrom: self.
nextObjects := nil!

detachOutsideSubSystemFrom: anObject

| oss |

oss := anObject subSystems.
(oss = nil) ifTrue: [self detachFrom: anObject].
(oss = subSystems) ifFalse: [self detachFrom: anObject]!
"Starts the detaching operation of the objects"

(itsSimulator = nil) ifTrue: [cself. 
(|itsSimulator objectCount| = 1) ifTrue: [cself. 
nextPoint := self promptClick: 
'Click on object to detach from'. 
ob := itsSimulator findObjectAt: nextPoint. 
(ob = nil) ifFalse: [ 
 (ob = = self) ifFalse: [self detachFrom: ob] 
]."

(nextObjects = nil) ifFalse: [self detachFrom: nextObjects]. 
itsSimulator startup: self!

detail

detailed := true!

detailed

c'detailed!

display

iconSelected show!

displayBordered

iconSelected border!

doesNotReceiveFrom: anObject

"if objects are detached from each other then this method is 
called. The identity of its attached to object is removed 
from itself"

(previousObjects includes: anObject) ifFalse: [cnil]. 
iconSelected removePointFrom: (anObject outPoint). 
previousObjects remove: anObject!

doit

self subclassResponsibility!

editSubsystem: aRectangle sim: aSimulation 
| ob nextPoint |

"Prompts the user for the objects to be included into the 
model base. The objects selected by the user are then 
bordered and their instance variables stored"
itsSimulator := aSimulation.
tempEventChain := SortedCollection new: 1000.
nextPoint := (aRectangle origin) - (10@10).
components do: [ : each | each border ].
[aRectangle containsPoint: nextPoint] whileFalse: [.
   nextPoint := Menu promptClick:
      'Click on object to include'.
   ob := itsSimulator findObjectAt: nextPoint.
   (ob = nil) ifFalse: [.
      (components includes: ob) ifTrue: [.
         ob decouple.
         ob unBorder.
         components remove: ob ifAbsent: [nil]
      ]
      ifFalse: [.
         (ob couple: self) ifTrue: [.
            ob displayBordered.
            self include: ob.
            self scanEventChain: ob.
            FreeDrawing !
         ]
      ]
   ]
].

finish

¢nil!

fromFile

self subclassResponsibility!

graphicsForCopy

graphicsOn := true!

graphicsOff

"Removes the graphics from the screen"

graphicsOn := false.
iconSelected turnOff!

graphicsOn

| s |

"Places the icons at the mouse and then further on the Simulation pane only if the graphics are on"
graphicsOn := true.
(!iconSelected = nil) ifFalse: [self fromFile].
"((iconSelected position) = (0.1@0.1)) ifFalse: [iconSelected
placeAtMouse].
"((iconSelected position) = (0.1@0.1)) ifTrue: [self place].
(iconSelected isNil) ifFalse: [
  iconSelected turnOn].

graphOff

self graphicsOff.
components do: [: each | each graphicsOff].

graphOn

graphicsOn := true.
(iconShowing) ifTrue: [self showIcon].

hideIcon

"The identity of the object is removed from the eventChain and
the iconsSelectedTillNow database. Further, it is removed from
the screen"

>((iconSelected position) = (0.1@0.1)) ifTrue: [self].
(iconShowing = nil) ifFalse: [itsSimulator pullOutOf: self].
iconShowing := false.
iconSelected turnOff.
(detailed) ifTrue: [components do: [: each | each unBorder]].

iconPosition

c((iconSelected position)!

iconPosition: aPoint
| p oldIn outOut |
"Sets the position of the icon to aPoint and removes it from
its previous position"

oldIn := self inPosition.
oldOut := self outPosition.
p := (Point new) x: ((aPoint x) rounded); y: ((aPoint y)
rounded).
iconSelected position: p.
positionInMeters := itsSimulator changeToMeters: aPoint.
(previousObjects = nil) ifFalse: [
  previousObjects do: [: each |
    each changeToPoint: (self inPosition) from: oldIn
  ].
  (nextObjects = nil) ifFalse: [
    nextObjects changeFromPoint: (self outPosition) from:
oldOut
]

iconShowing

\textcopyright{} iconShowing!

\textbf{include: anObject}

components add: anObject!

\textbf{initialize}

"Initializes the instance variables of the SimulationObject
Class and creates its data structures"

label := 'a', ((self class) printString).
name := ('a', (Date today printString).
         (Time millisecondClockValue printString)).
iconSelected := SimIcons new.
previousObjects := Set new.
previousObjDict := Bag new.
components := Set new.
detailed := true.
iconShowing := false.
graphicsOn := false.
self resetStats!

\textbf{inPosition}

\textcopyright{}(iconSelected inPosition)

\textbf{input}

\textcopyright{}stats contents!

\textbf{label}

\textcopyright{}label!

\textbf{label: aString}

| iconWasShowing |

"Puts an identifying label on each object created"

iconWasShowing := iconShowing.
(iconWasShowing) ifTrue: [self hideIcon].
self labelSim: aString.
(iconWasShowing) ifTrue: [self showIcon]!
labelSim: aString
        | iconWasShowing |

        label := aString.
        (iconSelected = nil) ifFalse: [ 
            (subSystems = nil) ifFalse: [ 
                iconWasShowing := subSystems iconShowing.
                (iconWasShowing) ifTrue: [subSystems hideIcon] 
            ].
            (subSystems = nil) ifFalse: [ 
                (iconWasShowing) ifTrue: [subSystems showIcon] 
            ]
        ]
    ]

menu
    "The Object Menu"

cPromptMenu
    prompt: 'Object Menu'
    labels: 'Attach to\Detach from\Remove\View statistics\Probability\finish' withCRs
    lines: #(3)
    selectors: #('attachToInteract' 'detachFrom' 'remove'
                   'viewStatistics' 'probab' 'finish')

menuResponse: aString
    "Depending upon the function selected by the user from the Object Menu the corresponding method is executed"

    (aString = 'TypeOfPart') ifTrue: [self typeOfPart].
    (aString = 'attachToInteract') ifTrue: [self attachToInteract].
    (aString = 'detachFrom') ifTrue: [self detachFrom].
    (aString = 'remove') ifTrue: [self remove].
    (aString = 'viewStatistics') ifTrue: [self viewStatistics].
    (aString = 'probab') ifTrue: [self probab].
    (aString = 'robotprg') ifTrue: [self rob].
    (aString = 'robotno') ifTrue: [self robotno].
    (aString = 'changeVel') ifTrue: [self changeVel].
    (aString = 'finish') ifTrue: [ 
        (self isKindOf: Robot) ifTrue: [ 
            (self robotNo isNil) ifTrue: [ 
                Menu notify: 'Select robot number before finishing'.
                itsSimulator startUp: self
            ]
            iFalse: [self finish].
        ]
        ]
    self finish.
]!
moveIt

    self subclassResponsibility!

moveIt: anInteger

    self subclassResponsibility!

name

    ⦿ name!

nexts

    | s |

    s := OrderedCollection new.
    (nextObjects = nil) ifFalse: [s add: nextObjects].
    ⦿ s!

noDetail

    detailed := false!

numberReceived

    ⦿ numberReceived!

numberReceived: anInteger

    numberReceived := anInteger!

openStatsWin

    | topPane |

    "Opens the Statistics Pane and passes control to it"

    statisticsPane := TopPane new label: 'Statistics'.
    statisticsPane addSubpane:
      (TextPane new
        model: self;
        name: #input).

    statisticsPane dispatcher open scheduleWindow!

outPosition

    ⦿ (isOnSelected outPosition)!
partList
   ?PartList!

place
   | oldIn oldOut iconWasShowing |

   "Places the icon in the Simulation Pane at a certain position.

   If that icon is present at another position then it is erased from there"

   oldIn := self.inPosition.
   oldOut := self.outPosition.
   (subSystems = nil) ifFalse: [ 
      iconWasShowing := subSystems iconShowing.
      (iconWasShowing) ifTrue: [subSystems hideIcon] 
   ].
   iconSelected placeAtMouse.
   (subSystems = nil) ifFalse: [ 
      (iconWasShowing) ifTrue: [subSystems showIcon] 
   ].
   positionInMeters := (itsSimulator changeToMeters: (iconSelected position)).
   (previousObjects = nil) ifFalse: [ 
      previousObjects do: [: each | 
         each changeToPoint: (self.inPosition) from: oldIn 
      ]
   ].
   (nextObjects = nil) ifFalse: [ 
      nextObjects changeFromPoint: (self.outPosition) from: oldOut
   ]!

placeComponents
   | oldPlace |

   "Objects from the model base are placed on the screen one after the other"

   (detailed) ifTrue: [ 
      components do: [: each | 
         each graphicsOff
      ]
   ].
   oldPlace := refPoint.
   (detailed) ifTrue: [ 
      components do: [: each | 
         each graphicsOn
      ]
   ]!
positionInMeters

\textquote{positionInMeters!}

positionInMeters: aPoint
| p q |

positionInMeters := aPoint.
p := ItsSimulator changeToPixels: positionInMeters.
q := (Point new) x: (p x rounded); y: (p y rounded).
IconSelected position: q!

previousObjDict

\textquote{previousObjDict!}

previousObjDict aBag

previousObjDict := aBag!

probab

self subclassResponsibility!

promptClick: aString

"Give message as a prompt attached to the cursor, and go away."

| height width rectExtent background mess |
width := ((aString size) \times (SysFont charSize x)) + 14.
height := 14 max: (SysFont charSize y).
rectExtent := (Point new) x: width; y: height.
background := ColorForm fromDisplay:
            (Rectangle origin: (0@0) extent: rectExtent).
ArrowForm displayAt: (0@0).
aString displayAt: (14@0).
mess := Form fromDisplay: (Rectangle origin: (0@0) extent: rectExtent).
background display.
mess offset: (0@0).
mess reverse.
\textquote{mess placeAtMouse!}

receivesFrom: anObject

"Add anObject to the previousObjects database of the controlling object. Further, connect the two object."

(previousObjects includes: anObject) ifTrue: [ \textquote{false} ].
previousObjects add: anObject.
previousObjDict add: (anObject label).
iconSelected addPointFrom: (anObject outPosition).
@true!

refPoint

@refPoint!

remove

"Detaches the object from its previous object and next object
and then removes it from the simulation. Further, all
scheduled events of object are removed from the eventChain."

i := Menu yesNo: 'Are you sure you want to remove this
object?'.
(i = nil) ifFalse: [
    (i = 0) ifTrue: [@nil]
    ifFalse: [
        itsSimulator deSchedule: self.
        itsSimulator pullOutOf: self.
        (nextObjects = nil) ifFalse: [
            nextObjects previousObjDict remove: (self label)
            ifAbsent: [].
            self detachFrom: nextObjects
        ].
        (previousObjects notNil) ifTrue: [
            previousObjects do: [ : each | each detachFrom: self
        ]
    ]
    (graphicsOn) ifTrue: [ self graphicsOff ].
    itsSimulator := nil
].
]

resetStats

numberReceived := 0.
numberReceivedDict := Dictionary new!

rmNoDetach

"Objects are removed from the screen after they are stored
into the model base. Further, all their interconnections are
removed"

{detailed} ifFalse: [itsSimulator pullOutOf: self].
(detailed & iconShowing) ifTrue: [itsSimulator pullOutOf:
    self].
itsSimulator removeSubSystem: self.
(iconShowing) ifTrue: [self hideIcon].

APPENDIX C 71
components do: [: each | each rmNoDetachSim ].
components do: [: each | each decouple ].
(graphicsOn & (detailed = false)) ifTrue: [
    self graphOff
].
itsSimulator := nil!

rmNoDetachSim

itsSimulator deSchedule: self.
itsSimulator pullOutOf: self.
(nextObjects = nil) ifFalse: [self detachOutsideSubSystemFrom:
    nextObjects].
(previousObjects notNil) ifTrue: [
    previousObjects do: [: each | each
detachOutsideSubSystemFrom: self ]
].
(graphicsOn) ifTrue: [ self graphicsOff ].
itsSimulator := nil!

runBy: aSimulation

itsSimulator := aSimulation.
itsSimulator addIcon: self.

scanEventChain: anObject:
    "The eventChain is scanned and copied into the tempEventChain.
This event chain is stored into the model base"

itsSimulator eventChain do: [: each |
    (each actor) = anObject ifTrue: [
        tempEventChain add: (Event newFor: anObject at: (each
time) palletNumber: (each palletNumber)).
        itsSimulator deSchedule: anObject
    ].
]!

showIcon

" (iconSelected position) = (0.1@0.1) ifTrue: [
    self place
] "
(itsSimulator = nil) ifFalse: [itsSimulator addIcon: self].
(detailed) ifTrue: [components do: [: each | each border ]].
iconShowing := true.
" iconSelected turnOn "!

subSystems

?subSystems!
tempEventChain

c.tempEventChain!

tempEventChain: aSortedCollection

tempEventChain := aSortedCollection!
C.2.1 Active Entity Classes

C.2.1.1 Symbolic Conveyor Class

\textbf{SimulationObjects \texttt{subclass: \#Conveyor}}
\begin{verbatim}
instanceVariableNames: `partName iconBroken notAccepting
partPosOnConv movingQueue travelTime partQueue capacity
waitingToDrawNewIcon numberSentOn avgNumberIn
largestNumberIn timeLastStateChange`
classVariableNames: `
poolDictionaries: `''
\end{verbatim}

!Conveyor class methods ! !

!Conveyor methods !

\texttt{accept: aPart}
\begin{verbatim}
| time addTime n firstPart nextArrival name |
``
``

"Accepts the part from its previous object if its current
capacity is less than 10. Statistics on the number of parts
received are updated and the future departure time from the
object is scheduled"

((self numberIn) < capacity) ifTrue: [
  time := itsSimulator simTime.
  addTime := time - timeLastStateChange.
  timeLastStateChange := time.
  avgNumberIn := avgNumberIn + (addTime * (self numberIn)).
  movingQueue add: aPart.
  (travelTime notNil) ifFalse: [n := 0]
    ifTrue:[n := travelTime next].
  (n < 0) ifTrue: [nextArrival := time] ifFalse: [
    nextArrival := time + n
  ].
  name := aPart partType partName.
  ((numberReceivedDict includesKey: name)) ifFalse: [
    numberReceivedDict at: name put: 0
  ].
  numberReceived := ((numberReceivedDict at: name) + 1).
  numberReceivedDict at: name put: numberReceived.
  itsSimulator scheduleEvent: self at: nextArrival for: name.
  waitingToDrawNewIcon := true.
  n := self numberIn.
  (n > largestNumberIn) ifTrue: [ largestNumberIn := n ].
  ttrue
] ifFalse: [false]
\end{verbatim}
avgNumberIn
  \#avgNumberIn!

avgNumberIn: aNumber
  avgNumberIn := aNumber!

capacity: aNumber
  capacity := aNumber!

copy
  | ob |

  "Copies all the instance variables and stores them with the
  object into the model base"

  ob := super copySim,
  ob travelTime: (travelTime copy),
  ob capacity: (capacity copy),
  ob notAccepting: (notAccepting copy),
  ob movingQueue: (movingQueue copy),
  ob numberReceivedDict: (numberReceivedDict copy),
  ob numberReceived: (numberReceived copy),
  ob largestNumberIn: (largestNumberIn copy),
  ob timeLastStateChange: (timeLastStateChange copy),
  ob waitingToDrawNewItem: (waitingToDrawNewItem copy),
  ob partPosOnConv: (partPosOnConv copy),
  ob iconBreaks: (iconBreaks copy),
  ob iconSelected: (iconSelected copy),
  ob partName: (partName copy),
  ob partQueue: (partQueue copy),
  ob avgNumberIn: (avgNumberIn copy).
  \#ob!

doit: aName
  "Executes the conveyor but checks before whether it has the
  right interactions attached to it"

  (previousObjDict includes: 'aEnter') ifFalse: [
    (previousObjDict includes: 'aGrasp') ifFalse: [
      (previousObjDict includes: 'aTransfer') ifFalse: [
        Menu notify: 'To work, attach interaction icon
        before Conveyor'.
      ]
    ]
  ].

  self getsThere: aName!

fromFile
"Creates a dictionary to store the icon set of the conveyor. The icon set contains five icons and are displayed on the screen with the change in capacity of the conveyor"

iconDict := Dictionary new.
iconDict at: '0' put: ConvMT.
iconDict at: '25' put: Conv25.
iconDict at: '50' put: Conv50.
iconDict at: '75' put: Conv75.
iconDict at: '100' put: Conv100.
iconSelected := SimIcons new.
iconSelected forms: iconDict.
iconSelected inOffset: (0.5@1).
iconSelected outOffset: (2@1).
iconSelected whichOne: '0'!

getThere: aName
"Removes the part from the conveyor and orders the part to be sent to its next object"
partName := aName.
(movingQueue = nil) ifTrue: [movingQueue := OrderedCollection new].
((movingQueue size) = 0) ifTrue: [nil].
(partQueue = nil) ifTrue: [partQueue := OrderedCollection new].
partQueue add: (movingQueue removeFirst).
(nextObjects = nil) ifFalse: [self moveIt].

iconBreaks

^
iconBreaks!

iconBreaks: aDictionary

iconBreaks := aDictionary!

iconSelected

^
iconSelected!

iconSelected: aForm

iconSelected := aForm!

InitIcons
"Controls the changing of the icons depending upon the current capacity of the conveyor"

iconBreaks := Dictionary new.
iconBreaks at: '25' put: ((capacity * 0.25) rounded).
iconBreaks at: '50' put: ((capacity * 0.5) rounded).
iconBreaks at: '75' put: ((capacity * 0.75) rounded).
iconBreaks at: '100' put: ((capacity * 1.0) rounded).

initialize

capacity := 10.
super initialize.
self initCaplcons!

largestNumberln

c!largestNumberln!

largestNumberln: aNumber

largestNumberln := aNumber!

moveit
| somethingMoved time addTime firstPart |

"The next object is checked whether it can accept the part. If it cannot then, the part is stored back into a queue. If it can then, the part is shown to pass between the two objects animatedly. After the part has passed then, the icon changes and the conveyor requests its previous object for another part."

(nextObjects isMemberOf: Robot) ifTrue: [ Menu notify: 'For parts to transfer there should be interaction with the robot'.
cnil ].
((partQueue size) > 0) ifTrue: [ time := itsSimulator simTime.
somethingMoved := true.
[somethingMoved & (partQueue size > 0)] whileTrue: [ somethingMoved := false.
firstPart := partQueue removeFirst.
(nextObjects accept: firstPart) ifTrue: [ numberSentOn := numberSentOn + 1.
addTime := time - timeLastStateChange.
timeLastStateChange := time.
avgNumberln := avgNumberln + (addTime * (partQueue size)).
iconSelected whichOne: (self whichIcon).
"iconSelected offset: 0@0."
iconSelected flash.

(graphicsOn) ifTrue: [ firstPart quickFrom: (iconSelected outPosition) ]]

APPENDIX C 77
to: (nextObjects inPosition) for:
(firstPart partType partName)

].
nextObjects showAccept.
somethingMoved := true.
notAccepting := false
]
ifFalse: [partQueue addFirst: firstPart.
    notAccepting := true]
]

(partQueue size) < capacity) ifTrue: [
    previousObjects do: [ : each | each moveit]
    ]

movingQueue
  movingQueue!

movingQueue: aCollection
    movingQueue := aCollection!

notAccepting
  notAccepting!

notAccepting: aBoolean
    notAccepting := aBoolean!

numberIn
  numberIn!

((partQueue size) + (movingQueue size))!

numberSentOn
  numberSentOn!

numberSentOn: aNumber
    numberSentOn := aNumber!

partName
  partName!

partName: aString
partName := aString!

partPosOnConv := partPosOnConv!

partPosOnConv: aDictionary
  partPosOnConv := aDictionary!

partQueue := partQueue!

partQueue: aCollection
  partQueue := aCollection!

probab
  | s |

  "Computes the travel time on the conveyor according to some probability distribution"
  s := ProbabilityDist fromUser.
  if s = nil then: [return: [nil]].
  (s isKindOf: Integer) ifTrue:
    (if self travelTime: (Constant returns: 0))
      ifFalse:
        (if self travelTime: s).
    itsSimulator startUp: self!

resetStats
  | time |

  "Resets the instance variables and waits for the execution to start again"
  super resetStats.
  movingQueue := OrderedCollection new.
  timeLastStateChange := 0.
  avgNumberOf := 0.
  largestNumberOf := 0.
  numberOfSentOn := 0.

  (itsSimulator = nil) ifFalse:
    (time := itsSimulator simTime.
      movingQueue := nil)
    ifFalse:
      movingQueue do: [ : each |
        (each notNil) ifTrue:
          each enteredAt: ((each enteredAt) - time)
showAccept
  "The conveyor shows the acceptance of the part by changing its
  icon depending upon the current capacity and flashing it"

(waitingToDoNewIcon) ifTrue: [
  iconSelected whichOne: (self whichIcon).
  iconSelected flash.
  waitingToDoNewIcon := false
]!

timeLastStateChange

timeLastStateChange!

timeLastStateChange: aNumber

  timeLastStateChange := aNumber!

travelTime

  travelTime!

travelTime: aProbabilityDistribution

  travelTime := aProbabilityDistribution!

viewStatistics
  "The stats stream contains the statistics of the conveyor. It
  is displayed on the Statistic Pane"

  stats := WriteStream on: String new.
  stats
    nextPutAll: 'STATISTICS OF CONVEYOR';
    cr;
    cr;
    nextPutAll: ('Capacity: ', (capacity printString));
    cr;
    nextPutAll: ('Travel Time: ', (travelTime printString));
    cr;
    nextPutAll: ('No. of pallets on conveyor: ', (movingQueue
      size printString));
    cr.
  numberReceivedDict keys do: [: each |
    stats
      nextPutAll: ('Number Received of ', (each

printString)', ' : ', ((numberReceivedDict at: each)printString));
  }
].

(self isMemberOf: Conveyor) ifTrue: [
  self openStatsWin;
  itsSimulator startUp: self!]

waitingToDrawNewIcon
  \waitingToDrawNewIcon!

waitingToDrawNewIcon: aBoolean

waitingToDrawNewIcon := aBoolean!

whichIcon
  | s i j |

"Determines which icon is to be displayed depending upon the current capacity of the conveyor"

s := self numberOf;
(capacity = 0) ifTrue: [s='0']
ifFalse: [
  (s = 0) ifTrue: [s='0'].
  i := iconBreaks at: '25'.
  ((s > 0) & (s <= i)) ifTrue: [s='25'].
  j := iconBreaks at: '50'.
  ((s > i) & (s <= j)) ifTrue: [s='50'].
  i := iconBreaks at: '75'.
  ((s > j) & (s <= i)) ifTrue: [s='75'].
  (s > i) & (s <= capacity) ifTrue: [s='100'].
]!!
C.2.1.2 Kin/Geo Conveyor Class

Conveyor subclass: #KinConveyor

instanceVariableNames: ‘part convbed pallet travelTime vel
initialPos bufferQueue nextTime justEntered pointers’
classVariableNames: ‘’
poolDictionaries: ‘’!

!KinConveyor class methods !

!KinConveyor methods !

accept: aPart
| time addTime n firstPart nextArrival p name |

"Accepts the part if the current capacity of the conveyor is
less than 10. If Collision Check object is attached to
conveyor then, a geometric intersection check is done between
the conveyor bed and the pallet. If there is interaction then
only the part will be accepted. The position of the part is
then added to the partPosOnConv dictionary and the pointers
structure is updated. Further, the statistics on the number
of parts received is also updated"

(notAccepting) ifFalse: [
((self numberIn) < capacity) ifTrue: [
  initialPos := Array new: 3.
  initialPos at: 1 put: 40.
  initialPos at: 2 put: 4.
  initialPos at: 3 put: 7.1.
  (previousObjDict includes: ‘aGrasp’) ifTrue: [
    (part checkIntersection: convbed with: pallet)
    ifFalse: [
      Menu notify: ‘Part not interacting with the
      conveyor’.
      false
    ].
  ].
  (numberReceived > 0) ifTrue: [
    (partPosOnConv isEmpty) ifFalse: [
      ((partPosOnConv at: numberReceived) =
        initialPos) ifTrue: [false].
    ].
  ].
  name := aPart partType partName.
  ((numberReceivedDict includesKey: name)) ifFalse: [
    numberReceivedDict at: name put: 0
  ].
  numberReceived := ((numberReceivedDict at: name) + 1).
  numberReceivedDict at: name put: numberReceived.
]
time := itsSimulator.simTime.
addTime := time - timeLastStateChange.
timeLastStateChange := time.

avgNumberIn := avgNumberIn + (addTime * (self
numberIn)).
(time <= nextTime) ifTrue: [itsSimulator.deSchedule:
self].

movingQueue add: aPart.
partPosOnConv at: numberReceived put: initialPos.
(numberReceived > 1) ifTrue: [
(partPosOnConv size > 1) ifTrue: [
((partPosOnConv at: numberReceived) =
(partPosOnConv at: (numberReceived - 1)))
ifTrue: [
itsSimulator.deSchedule: self].
].

pointers add: numberReceived.
justEntered := true.

itsSimulator.scheduleEvent: self at: time for: name.
waitingToDrawNewIcon := true.
p := self.numberIn.
(p > largestNumberIn) ifTrue [ largestNumberIn := p ].
true]
ifFalse: [cfalse].
]
ifTrue: [cfalse]!

changeVel
| x |

"Prompts the user for a change in the velocity of the
conveyor"

x := Prompter
  prompt: 'What is the velocity of the conveyor'
  default: '10'.
vel := x asFloat.
[vel between: 5 and: 20] whileFalse: [
  x := Prompter
  prompt: 'Velocity should be between 5 and 20'
  default: '10'.
  (x = nil) ifFalse: [vel := x asFloat]
  ifTrue: [vel := 1].
]!

checkPartPresent
"Checks whether there are any parts on the conveyor"

(movingQueue size isNil) ifFalse: ['no']!
convbed

@convbed!

convbed: anArray

convbed := anArray!

copy
  | ob |

  "Copies the instance variables of the Kin/Geo conveyor to the
  model base along with the object"

  ob := super copy.
  ob part: (part copy).
  ob convbed: (convbed copy).
  ob pallet: (pallet copy).
  ob travelTime: (travelTime copy).
  ob vel: (vel copy).
  ob initialPos: (initialPos copy).
  ob nextTime: (nextTime copy).
  ob justEntered: (justEntered copy).
  ob pointers: (pointers copy).
  ob!

doit: aName
  | t x y z part array partPosition |

  "Updates the positions of the parts on the conveyor after a
certain time interval. After a part reaches the end of the
conveyor then it is removed from the partPosOnConv and the
pointers data structure."

  (notAccepting) ifFalse: [
    t := itsSimulator simTime.
    x := (t - timeLastStateChange) * vel.
    timeLastStateChange := t.
    pointers do: [: each |
      array := partPosOnConv at: each.
      y := array at: 1.
      partPosition := (y - x).
      (partPosition < = 0) ifTrue: [
        self getsThere: aName.
        part := pointers removeFirst.
        partPosOnConv removeKey: part
      ]
      ifFalse: [
        array at: 1 put: partPosition.
        partPosOnConv at: each put: array
      ].
    ].

  (justEntered) ifTrue: ["
partPosOnConv at: numberReceived put: initialPos.
justEntered := false.
z := self checkPartPresent.
(z = 1) ifTrue: [
isSimulator scheduleEvent: self at: (t + 0.4) for:
aName.
nextTime := (t + 0.4).
]
ifTrue: [itsSimulator deSchedule: self]!

crateFile

| iconDict |

"Creates an icon dictionary which contains a set of the five icons of the Kin/Geo conveyor. The icons are initialized and the ConvKinMT icon is brought up"

iconDict := Dictionary new.
iconDict at: '0' put: ConvKinMT.
iconDict at: '50' put: ConvKin50.
iconDict at: '75' put: ConvKin75.
iconDict at: '100' put: ConvKin100.
iconSelected := SimlIcons new.
iconSelected forms; iconDict.
iconSelected inOffset; (0.5@1).
iconSelected outOffset; (2@1).
iconSelected whichOne; '0'!

getpalconv

| output1 output2 output3 |

"Geometric data of the conveyor and pallet is read from the hard disk for intersection analysis"

part := Geometric new.
output1 := self readDataOf: 'convbed'.
output2 := self readDataOf: 'pallet'.
output3 := self readDataOf: 'trconv'.
convbed := part translate; output1 by: output3.
pallet := part translate; output2 by: initialPos!

initialize

"Instance variables of the Kin/Geo Conveyor Class are initialized and some data structures created"

notAccepting := false.
initialPos := Array new: 3.
initialPos at: 1 put: 40.
initialPos at: 2 put: 4.
initialPos at: 3 put: 7.1.
vel := 10.
self getpalconv.
nextTime := 0.
pointers := OrderedCollection new.
partPosOnConv := Dictionary new.
super initialize.
travelTime := 4.0!

initialPos
initialPos!

initialPos: anArray
initialPos := anArray!

justEntered
justEntered!

justEntered: aBoolean
justEntered := aBoolean!

menu
"The Conveyor Object Menu"

promptMenu
prompt: 'Conveyor Menu'
labels: 'Attach to\Detach from\Remove\Change velocity\View statistics\finish' withCrs
lines: #3
selectors: #('attachTo' 'interact' 'detachFrom' 'remove'
'changeVel' 'viewStatistics' 'finish')!

nextTime
nextTime!

nextTime: aNumber
nextTime := aNumber!

pallet
pallet!

pallet: anArray
pallet := anArray!
part
  ☻part!

part: anObject
  part := anObject!

pointers
  ☻pointers!

pointers: anOrderedCollection
  pointers := anOrderedCollection!

resetStats
  super resetStats!

travelTime
  ☻travelTime!

travelTime: aNumber
  travelTime := aNumber!

vel
  ☻vel!

vel: aNumber
  vel := aNumber!

viewStatistics
  "Writes the statistics of the Kin/Geo Conveyor onto the Statistics Pane. Also, the statistics of the Symbolic Conveyor are first written to the pane."

  super viewStatistics.
  stats
    cr;
    nextPutAll: 'Velocity of the Conveyor: ', (vel printString);
cr:
nextPutAll: 'Positions of the pallets: '
    cr; space; space; space.

pointers do: [: each |
    array := partPosOnConv at: each.
    stats
    nextPutAll: ('Pallet: ', each printString, ' at: ',
        array printString);
    cr; space; space; space].
(self isMemberOf: KinConveyor) ifTrue: [
    self openStatsWin.
    itsSimulator startUp: self]!
C.2.1.3 Symbolic Robot Class

`SimulationObjects` subclass: `#Robot`

instanceVariableNames: 'aPane aTopPane paletteType robotNo
partName numRows rows robotProg robotProgNo whenDone
whenChangeDone workpieces timeLastStateChange state
oldState processingProbability partsNeededToCycle
idleTime workingTime blockedTime '

classVariableNames: `RobotProgDict`'
poolDictionaries: `' '

!Robot class methods!

`initProgDict`

"Creates and initializes the RobotProgDict and stores some robot programs in it"

{RobotProgDict isNil} ifTrue: [
RobotProgDict := Set new.
RobotProgDict add: 'robprog1'.
RobotProgDict add: 'robprog2'.
RobotProgDict remove: 'robprog'
]

!Robot methods!

`accept: aPart` | time n |

"Accepts the part from the previous object if its status is idle. The status of the robot is then changed to working and there is a corresponding change in the icon of the robot. The departure time of the part is then scheduled according to the execution time of the robot program"

{itsSimulator = nil} ifTrue: [nil].
time := itsSimulator simTime.
{state = 'idle'} ifTrue: [
workpieces add: aPart.
sel changeStateTo: 'working'.

{workpieces size = partsNeededToCycle} ifTrue: [
waitingToListAccept := true.
{previousObjDict includes: 'aAssembly'} ifFalse:
{processingProbability := nil}.
{processingProbability = nil} ifTrue: [whenDone := time] ifFalse: [
next := processingProbability next.
(n < 0) ifTrue: [whenDone := time] ifFalse: [}
whenDone := (time + n)
].
].
name := aPart partType partName.
((numberReceivedDict includesKey: name)) ifFalse: [
    numberReceivedDict at: name put: 0
].

numberReceived := ((numberReceivedDict at: name) + 1).
numberReceivedDict at: name put: numberReceived.

(self isKindOf: KinRobot) ifTrue: [self setPart:
aPart]
itsSimulator scheduleEvent: self at: whenDone for: name
].
'true
]
ifFalse: [false]!

aTopPane

| aTopPane |

aTopPane := aPane!

blockedTime

| blockedTime |

blockedTime := aNumber!

changeStateTo: aString
| time |

"Changes the status of the robot and calculates the statistics on the time"

(aString = state) ifTrue: [nil].
time := itsSimulator simTime.
(state = 'blocked') ifTrue: [
builtinTime := builtinTime + (time - timeLastStateChange)
].
(state = 'idle') ifTrue: [
idleTime := idleTime + (time - timeLastStateChange)
].
(state = 'working') ifTrue: [
workingTime := workingTime + (time - timeLastStateChange)
].
timeLastStateChange := time.
oldState := state.
state := aString.
iconSelected whichOne: state!

closeIt
"Closes the Robot Program pane and passes control to the Simulation Pane"

aTopPane dispatcher closeItSim.
aTopPane dispatcher deactivateWindow.
Scheduler displayAll.
itsSimulator startUp: self!

copy
| ob |
"Copies all the instance variables of the robot and stores
them with the object in the model base"

ob := super copySim.
ob robotNo: (robotNo copy).
ob partName: (partName copy).
ob iconSelected: (iconSelected copy).
ob whenDone: (whenDone copy).
ob robProg: (robotProg copyFrom: 1@1 to: (self numOfRows)@8).
ob workpieces: (workpieces copy).
ob timeLastStateChange: (timeLastStateChange copy).
ob state: (state copy).
ob oldState: (oldState copy).
ob partsNeededToCycle: (partsNeededToCycle copy).
ob idleTime: (idleTime copy).
ob workingTime: (workingTime copy).
ob blockedTime: (blockedTime copy).
ob processingTime: (processingProbability copy).
ob numOfRows: (numOfRows copy).
ob robotProgNo: (robotProgNo copy).
ob numberReceived: (numberReceived copy).
ob numberReceivedDict: (numberReceivedDict copy).
}ob!

createProg
"The Robot Program pane is created and the control passed to
it. The Pane is provided with its own menu set"

Scheduler topDispatcher deactivateWindow.
aPane := TextPane new.
aPane
  dispatcher: aPane dispatcher.

aTopPane := TopPane new.
aTopPane
  label: 'Robot Program';
model: self;
menu: #topPaneProgMenu;
minimumSize: 24@48;
right Icons: #(resize collapse zoom);
foreColor: 10;
yourself.

aTopPane addSubPane:
  (aPane :=
   TextPane new
   change: #fileit with: :
   menu: #robotProgPaneMenu;
   name: #inputBlankString;
   model: self).

aTopPane dispatcher
open;
activate;
display.
Scheduler add: aTopPane dispatcher.
aTopPane dispatcher searchForActivePane!

doIt: aName
  | time |

  partName := aName.
  (state = 'working') ifTrue: [self done: aName]!

done: aName
  | time n |

  "If the status of the robot is working then finish execution
  and send the part to the next object"

  time := itsSimulator simTime.
  self changeStateTo: 'blocked'.
  self moveIt.
  iconSelected flash.
  (state = 'blocked') ifTrue: [
    (graphicsOn) ifTrue: [(workpieces at: 1) displayAt: (self
     outPosition)]
  ].
  (state = 'working') ifTrue: [
    (graphicsOn) ifTrue: [(workpieces at: 1) displayAt: (self
     inPosition)
  ]!

fileIt: aString with: aDispatcher
  !fileInput aFile |

  "Stores the robot program created on the Robot Program Pane
  into a file on the hard disk"

  aFile := Prompter
    prompt: 'Enter file name (max. 8 letters)'

APPENDIX C
default: 'robProg'.
aDispatcher pane fileOutOn: fileInput.
RobotProgDict add: aFile.
fileInput close!

fromFile
| d peq |

"Creates an icon dictionary containing a set of robot icons"

d := Dictionary new.
d at: 'idle' put: RobotMT.
d at: 'blocked' put: RobotBloc.
d at: 'working' put: RobotWork.
iconSelected := SimIcons new.
iconSelected forms: d.
iconSelected inOffset: (1@1).
iconSelected outOffset: (1@1).
iconSelected whichOne: 'idle'!

iconSelected

g!iconSelected!

iconSelected: aForm

iconSelected := aForm!

idleTime

g!idleTime!

idleTime: aNumber

idleTime := aNumber!

initialize

"Initializes the instance variables of the robot"

super initialize.
state := 'idle'.
oldState := 'idle'.
workpieces := OrderedCollection new.
partsNeededToCycle := 1.
Robot initProgDict!

inputBlankString
matrixFromArray: anArray from: start size: aPoint |
  k aMatrix |

  "Creates an matrix of size x @ y from anArray starting from
  the kth element of the array"

  k := start.
  aMatrix := Matrix new: aPoint.
  1 to: (aPoint x) do: [i | 
      1 to: (aPoint y) do: [j | 
          aMatrix at: i @ j put: (anArray at: k).
          k := k + 1].
  ].
  aMatrix!

menu

  "The Robot Object Menu"

  | PromptMenu |
  prompt: 'Robot Menu'
  labels: 'Attach to\::\:Detach from\::\:Remove\::\:Robot Program\::\:Robot Number\::\:View statistics\::\:finish' withCrs
  lines: #3
  selectors: #('attachTo\::\:interact' 'detachFrom\::\:remove'
              'robotProg\::\:robotNo\::\:viewStatistics\::\:finish')!

moveIt |
  time somethingMoved firstPart |

  "Sends the part from the robot to the next object if that
  object can accept the part. The status and icon of the robot
  are changed. Further, the robot requests its previous object
  to send it a part"

  (nextObjects = nil) ifTrue: [nil].
  (state = 'blocked') ifTrue: [
      (workpieces size) = 0] ifTrue: [self changeStateTo: 'idle'
      ]
  ifFalse: [
      time := itsSimulator simTime.
      somethingMoved := true.
      [somethingMoved & (workpieces size > 0)] whileTrue: [
          somethingMoved := false.

          "firstPart := self checkPalletType."
          firstPart := workpieces removeFirst.

          (nextObjects accept: firstPart) ifTrue: [
              ((workpieces size) = 0) ifTrue: [self changeStateTo: 'idle'].
          ]
      ].
  ]
iconSelected flash.
  (graphicsOn) ifTrue: [
    (state = 'blocked') ifTrue: [
      (workpieces at: 1) displayAt: (self outPosition)
    ].
    (previousObjDict includes: 'aAssemble')
    ifTrue: [
      firstPart quickFrom: (iconSelected outPosition)
      to: (nextObjects inPosition)
      for: partName
    ]
    ifFalse: [
      firstPart quickFrom: (iconSelected outPosition)
      to: (nextObjects inPosition) for:
      partName
    ].
  ].
nextObjects showAccept.
somethingMoved := true
] ifFalse: [ workpieces addFirst: firstPart ]
]
.

(state = 'idlw') ifTrue: [
  partsNeededToCycle timesRepeat: [
    previousObjects do: [: each | each moveit]
  ]
]!

numOfRows

¢numOfRows!

numOfRows: anInteger
  numOfRows := anInteger!

oldState

¢oldState!

oldState: aString
  oldState := aString!

palletType
\$\text{palletType!}$

\text{palletType: aInteger}
\text{palletType} := \text{aInteger!}

\text{partName}
\text{\$partName!}

\text{partName: aString}
\text{partName} := \text{aString!}

\text{partsNeededToCycle}
\text{\$partsNeededToCycle!}

\text{partsNeededToCycle: aNumber}
\text{partsNeededToCycle} := \text{aNumber!}

\text{probab}
| j \times y |

"Sets the execution time of the robot according to the time taken to execute the whole robot program"

\text{x := 0.}
\text{y := 0.}
\text{2 to: numORows do: [: i |}
\text{\hspace{1cm} x := robotProg at: i@7.}
\text{\hspace{1cm} y := x + y}.\text{j := Constant returns: y.}
\text{(j = nil) ifTrue: [\text{\$nil}].}
\text{(previousObjDict includes: 'aAssemble') ifFalse: [}
\text{\hspace{1cm} self processingTime: nil]}
\text{\hspace{1cm} ifTrue: [self processingTime: j]}
\text{\hspace{1cm} self processingTime: j]}

\text{processingTime: aProbabilityDistribution}
| \text{time n} |

"The completion of the execution is scheduled according to the probability distribution"

\text{(itsSimulator = nil) ifTrue: [\text{\$nil}].}
\text{time := itsSimulator simTime.}
\text{processingProbability := aProbabilityDistribution.
(state = 'working') ifTrue: [
  itsSimulator deSchedule: self.
  (processingProbability = nil) ifFalse: [
    n := processingProbability next.
    (n < 0) ifTrue: [whenDone := time] ifFalse: [
      whenDone := time + n
    ].
  ]
  itsSimulator scheduleEvent: self at: whenDone for:
    partName
]

resetStats
| time d |

"The instance variables of the robot are reset when the
execution of the simulation is restarted"

super resetStats.
timeLastStateChange := 0.
idleTime := 0.
workingTime := 0.
blokedTime := 0.
(itsSimulator = nil) ifFalse: [
  time := itsSimulator simTime.
  (whenDone notNil) ifTrue: [whenDone := whenDone - time].
  (workpieces notNil) ifTrue: [workpieces do: [ :each |
    each enteredAt: ((each enteredAt) - time)
  ]].
]

retrieveIt
| afile |

"Retrieves a robot program file from the hard disk"
afile := Prompter
  prompt: 'Enter file name (max. 8 letters)'
  default: 'aFile'.
afile := Disk file: afile.
aPane fileInFrom: afile!

rob
| names |

"Pops up the names of the robot programs created till now in
the form of a menu for selection by the user. The user can
select an existing program or create a new one"

names := RobotProgDict asSortedCollection asArray,
  (Array with: 'create new').
robotProgNo := (Menu
  labelArray: names
  lines: Array new
selectors: names) popUpAt: Cursor offset.

\( \text{robotProgNo notNil ifTrue: [} \)
\( \text{robotProgNo = 'create new' ifTrue: [} \)
\( \text{self createProg} \)
\( \text{]} \)
\( \text{ifFalse: [} \)
\( \text{self robotProg: robotProgNo.} \)
\( \text{(self isMemberOF: Robot) ifTrue: [self probab].} \)
\( \text{itsSimulator startUp: self} \)
\( \text{]} \)

\( \text{robotNo} \)

\( \text{\&robotNo!} \)

\( \text{robotNo} \)

"The identification number of the robot is requested from the user"

\( \text{robotNo := Prompter} \)
\( \text{prompt: 'What is the robot number?'} \)
\( \text{default: '1'.} \)
\( \text{itsSimulator startUp: self!} \)

\( \text{robotNo: anInteger} \)

\( \text{robotNo := anInteger!} \)

\( \text{robotProg: fileName} \)
\( \mid \text{array ordColl]} \)

"The robot program is read from the file fileName. It is then
stored into a N X 7 matrix, where N is the number of
instructions"

\( \text{array := Array new.} \)
\( \text{array := self readDataOf: fileName.} \)
\( \text{ordColl := array asOrderedCollection.} \)
\( \text{ordColl addFirst: 1000000.} \)
\( \text{ordColl addFirst: 0.} \)
\( \text{ordColl addFirst: 0.} \)
\( \text{ordColl addFirst: 0.} \)
\( \text{ordColl addFirst: 0.} \)
\( \text{ordColl addFirst: 3.} \)
\( \text{ordColl addFirst: 0.} \)
\( \text{array := ordColl asArray.} \)
\( \text{numOfRows := (array size)/7.} \)
\( \text{robotProg := self matrixFromArray: array from: 1 size:} \)
\( \text{(numOfRows @ 7)!} \)
robotProgPanelMenu

"The Robot Program Pane Menu"

<Menu
  labels: 'copy\cut\paste\save\retrieve' withCrs
  lines: #13
  selectors: #((copySelection cutSelection pasteSelection
            accept retrieveIt))

robotProg: aMatrix

  robotProg := aMatrix!

showAccept

"Shows the acceptance of the part by the robot by changing its icon"

(waitingToShowAccept) ifTrue: |
  (state = 'working') ifTrue: [|
    iconSelected whichOne: state.
    iconSelected flash
  ].
  waitingToShowAccept := false
|

state

  $state!

state: aString

  state := aString!

timeLastStateChange

  $timeLastStateChange!

timeLastStateChange: aNumber

  timeLastStateChange := aNumber!

topPaneProgMenu

"The top pane menu of the Robot Program Pane"

<Menu
  labels: 'color\label\collapse\cycle\frame\move\close'
     withCrs
     lines: #1
  selectors: #((color newLabel collapse cycle resize move}
viewStatistics
    \{ x y \}

    "The statistics of the robot are stored in the variable stats
    and then displayed on the Statistics Pane"

    x := 0.
y := 0.
2 to: numOfRows do: [ i |
x := robotProg at: i@7.
y := x + y].

stats := WriteStream on: String new.
stats
    nextPutAll: 'STATISTICS OF ROBOT';
cr;
cr;
    nextPutAll: ('Idle Time: ', (idleTime printString));
cr;
    nextPutAll: ('Working Time: ', (workingTime printString));
cr;
    nextPutAll: ('Program Execution Time: ',(y printString));
cr;
    nextPutAll: ('Status: ', (state printString));
cr.

(self isMemberOf: Robot) ifTrue: [
    self openStatsWin.
    itsSimulator startUp: self].

whenDone

\#whenDone!

whenDone: aNumber

    whenDone := aNumber!

workingTime

\#workingTime!

workingTime: aNumber

    workingTime := aNumber!

workpieces
\texttt{workpieces!}

\texttt{workpieces::\textit{anOrderedCollection}}

\texttt{workpieces := anOrderedCollection!}

C.2.1.4 Kinematic Robot Class

Robot subclass: #KinRobot

    instanceVariableNames: 'partPositionDict partEnteringName
partStream partArray partArriving transferA grasped
geoData link4 link5 part partA partB partC partD partE
pallet stime times posDict sensors count index clock
ptransJoint numJoint numLinks velDict jointPos jointVel
aDict jointLim kinData '

    classVariableNames: ''

    poolDictionaries: '' !

!KinRobot class methods !

    new

    @ (super new) initialize !

!KinRobot methods !

    copy

    | ob |

    "Copies the instance variables of the robot and stores them in
the model base along with the object"

    ob := super copy.
    ob transferA: (transferA copy).
    ob grasped: (grasped copy).
    ob geoData: (geoData copy).
    ob link4: (link4 copy).
    ob partA: (partA copy).
    ob posDict: (posDict copy).
    ob count: (count copy).
    ob index: (index copy).
    ob numOfRows: (numOfRows copy).
    ob ptransJoint: (ptransJoint copyFrom: 1@1 to: 3@1).
    ob robProg: (robotProg copyFrom: 1@1 to: (self numOfRows)@8).
    ob numJoint: (numJoint copy).
    ob numLinks: (numLinks copy).
    ob velDict: (velDict copy).
    ob jointPos: (jointPos copyFrom: 1@1 to: 4@1).
    ob jointVel: (jointVel copyFrom: 1@1 to: 4@1).
    ob jointLim: (jointLim copyFrom: 1@1 to: 2@4).
    ob kinData: (kinData copy).
    "ob!"

    count

    @ count!
count := aNumber!

createPartDataFrom: anArray
   | x y z |

   "Creates the data of the part. The dimensions of length, width and height of the part are got from the robot program and then the point coordinates, edge vectors, face normals and surface vectors are calculated"

x := anArray at: 1.
y := anArray at: 2.
z := anArray at: 3.
partStream
   nextPutAll: ('1 0 0 0'); cr.
   nextPutAll: ('2' , ',(x printString),', ',0 0'); cr.
   nextPutAll: ('3' , ', (x printString),', ',0'); cr.
   nextPutAll: ('4 0', ', (y printString),', ',0'); cr.
   nextPutAll: ('5 0', ', (y printString),', ', z printString)); cr.
   nextPutAll: ('6', ', (x printString),', ',y printString)); cr.
   nextPutAll: ('7 0 0', ', (z printString)); cr.
   nextPutAll: ('8', ', (x printString),', ',0', ', (z printString)); cr.
   nextPutAll: ('1 1 2'); cr.
   nextPutAll: ((x printString),', ',0 0'); cr.
   nextPutAll: ('2 2 3'); cr.
   nextPutAll: ('0', ', (y printString),', ',0'); cr.
   nextPutAll: ('3 4 3'); cr.
   nextPutAll: ((x printString),', ',0 0'); cr.
   nextPutAll: ('4 1 4'); cr.
   nextPutAll: ('0', ', (y printString),', ',0'); cr.
   nextPutAll: ('5 5 4');
nextPutAll: ('0 0',' ','0 0',' ','(z printString));
cr;
nextPutAll: ('6 6 5');
cr;
nextPutAll: ('-','(x printString),' ','0 0');
cr;
nextPutAll: ('7 6 3');
cr;
nextPutAll: ('0 0',' ','0 0',' ','(z printString));
cr;
nextPutAll: ('8 7 5');
cr;
nextPutAll: ('0',' ','(y printString),' ','0');
cr;
nextPutAll: ('9 1 7');
cr;
nextPutAll: ('0 0',' ','(z printString));
cr;
nextPutAll: ('10 8 7');
cr;
nextPutAll: ('-','(x printString),' ','0 0');
cr;
nextPutAll: ('11 6 8');
cr;
nextPutAll: ('0',' ','(y printString),' ','0');
cr;
nextPutAll: ('12 8 2');
cr;
nextPutAll: ('0 0',' ','(z printString));
cr;
nextPutAll: ('1 0 0 -1');
cr;
nextPutAll: ('4 3 -2 -1');
cr;
nextPutAll: ('2 0 1 0');
cr;
nextPutAll: ('-5 -6 7 -3');
cr;
nextPutAll: ('3 0 0 1');
cr;
nextPutAll: ('-10 -11 6 -8');
cr;
nextPutAll: ('4 0 -1 0');
cr;
nextPutAll: ('1 -12 10 -9');
cr;
nextPutAll: ('5 -1 0 0');
cr;
nextPutAll: ('9 8 5 -4');
cr;
nextPutAll: ('6 1 0 0');
cr;
nextPutAll: ('2 -7 11 12');
cr;
nextPutAll: ('0 0 0',' ','(x printString),' ','0 0',' ','(y
printString), ' ', '0');
cr;
nextPutAll: (x printString), ' ', (y printString), ' ', (z
printString), ' ', ' ', (x printString), ' ', '0 0 0 0',
' ', ' ', (z printString));
cr;
nextPutAll: (x printString), ' ', (y printString), ' ', (z
printString), ' ', ' ', (x printString), ' ', '0 0 0',
' ', ' ', (y printString), ' ', '0'),
cr;
nextPutAll: '0 0 0', ' ', (x printString), ' ', '0 0 0 0',
' ', (z printString));
cr;
nextPutAll: '0 0 0', ' ', (y printString), ' ', '0 0 0',
' ', (z printString));
cr;
nextPutAll: (x printString), ' ', (y printString), ' ', (z
printString), ' ', '0', ' ', ' ', (y printString), ' ', '0 0
0', ' ', ' ', (z printString));

delay: anArray

"The DELAY command sets the velocities of the links and the
joints equal to zero"

jointVel atAllPut: 0.
self dynManipFrame!

doit: aName

"Execute the robot program if the Assemble object is attached
to the robot object. If it is not, then a message is passed
to moveIt method to send the part to the next object. No
operation is performed on the part. After the execution of
each command of the robot program the clock is updated"

partName := aName.

(previousObjDict includes: 'aAssemble') ifTrue: [
  self manipCtrl.
  self updateClock.
  (index = numOfRows) ifTrue: [
    partPositionDict at: partEnterName put: partE.
    self changeStateTo: 'blocked'.
    self moveIt].
  ]
ifFalse: [
  Menu notify: 'Assemble icon needs to be attached to
Robot'.
  self changeStateTo: 'blocked'.
  self moveIt
].
partArriving := false!

dynManipFrame
| offsetVector array |

"Calculates the position and the velocities of the links and stores in posDict and velDict respectively"

array := Array new: 15.
offsetVector := Array new: 3.
1 to: 3 do: [:i |
   ptransJoint := self matVectMultPos: (aDict at: i).
   array := aDict at: i.
   offsetVector at: 1 put: (array at: 1).
   offsetVector at: 2 put: (array at: 2).
   offsetVector at: 3 put: (array at: 3).
   posDict at: i put: (self vectorAdd: ptransJoint and:
   offsetVector).
   velDict at: i put: (self matVectMultVel: (aDict at: i))].

fromFile

| d peq |

"Creates an icon dictionary with the robot icon set"

d := Dictionary new.
d at: 'idle' put: RobotKinMT.
d at: 'blocked' put: RobotKinBloc.
d at: 'working' put: RobotKinWork.
d iconSelected := Simicons new.
d iconSelected forms: d.
d iconSelected inOffset: (1@1).
d iconSelected outOffset: (1@1).
d iconSelected whichOne: 'idle'.

geoData

@geoData!

geoData: aDictionary

geoData := aDictionary!

geoStats

(self geokinStats) popUpAt: (Cursor offset)! 

grapsed

@grapsed!

grapsed: aBoolean

grapsed := aBoolean!
incrCounter: anArray

    count := count + 1!

index

    nil!

index: aNumber

    index := aNumber!

initialize

    "Initializes the instance variables of the robot and reads the
geometry of the links and their positions in space."

    posDict := Dictionary new.
    velDict := Dictionary new.
    jointPos := Matrix new: 4@1.
    jointVel := Matrix new: 4@1.
    jointPos atAllPut: 0.
    jointVel atAllPut: 0.
    count := 0.
    index := 1.
    part := Geometric new.
    self kinematicStart.
    partArriving := false.
    super initialize!

jointLim

    nil!

jointLim: aMatrix

    jointLim := aMatrix!

jointLimCheck

    "Checks whether the links of the robot are within the
workspace of the robot or not"

    1 to: 4 do: [: i |
        ((jointPos at: i@1) > (jointLim at: 2@1)) ifTrue: [c1].
        ((jointPos at: i@1) < (jointLim at: 1@1)) ifTrue: [c1].
    ].

c0!
jointPos

g jointPos!

jointPos: aMatrix
jointPos := aMatrix!

jointVel

g jointVel!

jointVel: aMatrix
jointVel := aMatrix!

jointVelIntegrate: firstArray and: secondArray time: dt |
array |

"Creates an array for calculating the joint positions"
array := Array new: 3.
1 to: 4 do: [: i |
array at: 1 put: (firstArray at: i@1). 
array at: 2 put: (secondArray at: i@1). 
array at: 3 put: dt. 
jointPos at: i@1 put: (self linearIntegrate: array)!!

kinData

g kinData!

kinData: anArray

kinData := anArray!

kinematicStart

"Reads the geometric data of the links of the robot and
offsets these links in space"

self manipinit: 'manip1'. 
aDict := self manipStart: 'manip2'. 
self dynManipFrame. 
transferA := self readDataOf: 'trparte'!

linearIntegrate: anArray

"Calculates and returns the position of the links"
link4

c!link4!

link4: anArray

link4 := anArray!

mainStatisticsMenu

cMenu
labels: 'Geo/Kin Statistics\General Statistics' withCr
lines: #()
selectors: #('geoStats' 'genStats')!

manipCtrl

| command array |

"Reads the robot program and decipheres the command to be executed. The robot program is stored in a N X 7 matrix, where N is the number of instructions. The pMove command is depicted by '1' in the second column of the program, the delay command by a '2' and the writeO command by a '3'."

array := Array new: 7.
array := robotProg rowAt: index.
command := robotProg at: (index @ 2).

(command <= 3) ifTrue: [
    (command <= 2) ifTrue: [
        (command = 1) ifTrue: [self pMove; array]
        ifFalse: [self delay: array].
    ]
    ifFalse: [self writeO: array].
].

index := self nextLine: (array at: 1)

manipInit: fileName

| array |

"Reads the limits of the workspace and stores it into a jointLim matrix"

array := self readDataOf: fileName.
jointLim := Matrix new: 2@4.
jointLim := self matrixFromArray: array from: 1 size: 2@4!
manipStart: fileName
    | array1 array2 array3 |

"Reads the geometric and kinematic data of the links and stores into a kinData and geoData dictionary"

array2 := Array new: 15.
array3 := Array new.
kInData := Dictionary new.
geoData := Dictionary new.
array1 := self readAsChar: fileName.
1 to: 6 do: [: i | 
    array2 := self readDataOf: (array1 at: i).
    kinData at: i put: array2].
7 to: 12 do: [: i | 
    array3 := self readDataOf: (array1 at: i).
    geoData at: (i - 6) put: array3].
link4 := geoData at: 5.
$kinData!

matrixFromArray: anArray from: start size: aPoint
    | k aMatrix |

"Creates a matrix of size x @ y from an anArray, starting from the kth element of the array"

k := start.
aMatrix := Matrix new: aPoint.
1 to: (aPoint x) do: [: i | 
    1 to: (aPoint y) do: [: j | 
        aMatrix at: i@j put: (anArray at: k).
        k := k + 1].
]!

$matrixFromArray!

matVectMultPos: anArray
    | ptrans |

"Calculates the joint positions"

ptrans := Matrix new: 3@4.
ptrans := self matrixFromArray: anArray from: 4 size: 3@4.
$ (ptrans innerProduct: jointPos)!

matVectMultVel: anArray
    | vtrans |

"Calculates the joint velocities"

vtrans := Matrix new: 3@4.
vtrans := self matrixFromArray: anArray from: 4 size: 3@4.
$vtrans innerProduct: jointVel)!
moveJoint: firstArray and: secondArray time: dt
   "The movement of the links and the joints are simulated from
   this method. The velocities and the positions of the links
   and the joints are continuously updated"
selJointVelIntegrate: firstArray and: secondArray time: dt.
sel dynManipFrame!

nextLine: aNumber
   "Returns the next instruction to be executed. If the last
   line of the program has been executed then the first line
   instruction pointer is returned"
   (aNumber = numOfRows) ifTrue: [print]
      ifFalse: [(aNumber + 1)]!

numJoint
   print!

numJoint: aNumber
   numJoint := aNumber!

numLinks
   print!

numLinks: aNumber
   numLinks := aNumber!

numOfRows
   print!

numOfRows: aNumber
   numOfRows := aNumber!

pallet
   print!

pallet: anArray
   pallet := anArray!
partA

cpartA!

partA: anArray

partA := anArray!

pMove: anArray

| remTime x y array anotherArray z a b length |

"The entire simulation of the links and joints is controlled by this method. The joint velocity of the joints is read from the 3rd to 6th columns of the program and the time of command execution from the 7th column. If the gripper grasps the part then, the position of the part is updated with the movement of the robot.
If a Collision Check object is attached to the robot then a geometric intersection check is done between the gripper and the part. Only if there is intersection will the part be moved. A geometric check is done again when the gripper releases the part. The updating of the position of the part is then stopped."

b := geoData at: 5.
array := Array new: 3.
remTime := anArray at: 7.
anotherArray := Array new: 3.
1 to: 4 do: [: i |
jointVel at: i@1 put: (anArray at: (i + 2))].

((anArray at: 5) = 0) and: [partArriving] ifFalse: [
length := (((self partList at: partName)dimensions) at: 2).
remTime := (1.8 - length)
].
((anArray at: 6) = 0) and: [partArriving := false] ifFalse: [
remTime := anArray at: 7
].


((anArray at: 8) = 0) and: [partArriving] ifFalse: [
link4 := part translate: b by: (posDict at: 5).

(previousObjDict includes: 'aGrasp') ifTrue: [
(part checkIntersection: partA with: link4) ifTrue:
[grasped := true]
ifFalse:
[grasped := false].
]
iFFalse; [grasped := true].
]
{(anArray at: 6) = 0) and: [partArriving := false]) iIFalse:

  iNk4 := part translate: b by: (posDict at: 5).

(PreviousObjDict includes: 'aGrasp') iIFalse. [
  (part checkIntersection: partE with: link4) iIFalse:
    grasped := true]

  iFFalse:
    [grasped := false].
]
ifFalse; [grasped := true].
]

((anArray at: 6) < 0) iFTrue; [grasped := false].

z := 3.
y := index.
1 to: 3 do: [: i |
    array at: i put: (robotProg at: y@z).
    z := z + 1].

1 to: 3 do: [: i |
    a := ((array at: i) * (robotProg at: y@7)).
    anotherArray at: i put: a].

(grasped isNil) iIFalse: [
  (grasped) iFTrue; [partE := part translate: partE by:
    anotherArray].
].

x := self jointLimCheck.
(x = '1') iFTrue: [
  Menu notify: 'The robot has gone out of its workcell'!]

posDict

cposDict!

posDict: aDictionary

posDict := aDictionary!

ptransJoint

cptransJoint!

ptransJoint: aMatrix

ptransJoint := aMatrix!
resetStats

    super resetStats!

robotProg

    ⌜robotProg!

robProg: aMatrix

    robotProg := aMatrix!

setPart: aPart

    | name |

    "The dimensions and position of the part arriving at the robot
    are read from the partPositionDict dictionary"

    name := aPart partType partName.
    partA := aPart partPositionDict at: name!

transferA

    ⌜transferA!

transferA: anArray

    transferA := anArray!

updateClock

    | xtime whenDone time |

    "The clock is updated after the completion of each command of
    the program. The completion time of the next command is then
    scheduled"

    xtime := robotProg at: (index@7).
    time := itsSimulator simTime.
    whenDone := time + xtime.
    itsSimulator scheduleEvent: self at: whenDone for: partName!

vectorAdd: aMatrix and: aVector

    | array |

    "Method for adding vectors together"

    array := Array new: 3.
    1 to: 3 do: [i |
        array at: i put: ((aMatrix at: i@1) + (aVector at: i))].
array!

velDict

cvelDict!

velDict: aDictionary

velDict := aDictionary!

viewStatistics
"Statistics of the Kin Robot are updated and displayed on the Statistics Pane"

super viewStatistics.
stats
cr;
nextPutAll: (‘Joint Velocity: ’, ((jointVel columnAt: 1)
 printString));
cr;
nextPutAll: (‘Joint Position: ’, ((jointPos columnAt: 1)
 printString));
cr.

(self isMemberOf: KinRobot) ifTrue: [self openStatsWin]!

wait: anArray

self updateClock.
{(sensors at: (anArray at: 6)@2) = (anArray at: 5)}
ifTrue: {
    index := self nextLine: (anArray at: 1)]
ifFalse: [
    jointVel atAllPut: 0.
    self dynManipFrame]!

writeO: anArray

| array |

"If there is another part entering the system at the robot then the writeO command can be used. The name of the part is stated in the schedule plan and the dimensions of the part are given in the program columns 3 to 5."

(partArriving) ifFalse: [
    partEnterName := Prompter
    prompt: 'Enter the part name'
    default: 'part'
    ].
partArriving := true.
array := Array new: 3.
array at: 1 put: (anArray at: 3).
array at: 2 put: (anArray at: 4).
array at: 3 put: (anArray at: 5).
self createPartDataFrom: array.
partArray := partStream contents asArray.
partE := part translate: partArray by: transferA!
C.2.1.5 Geometric Robot Class

KinRobot subclass: #GeoRobot
    instanceVariableNames: ""
    classVariableNames: ""
    poolDictionaries: "" !

!GeoRobot class methods ! !

!GeoRobot methods !

copy
    $super copy!

fromFile
    | d peq |

"Creates an icon dictionary and stores the icon set of the Geometric robot in it"

d := Dictionary new.
d at: 'idle' put: RobotGeoMT.
d at: 'blocked' put: RobotGeoBloc.
d at: 'working' put: RobotGeoWork.
iconSelected := SimIcons new.
iconSelected forms: d.
iconSelected inOffset: (1@1).
iconSelected outOffset: (1@1).
iconSelected whichOne: 'idle' !

viewStatistics

super viewStatistics.
stats
cr;
nextPutAll: ('Link 0 Position: ', ((posDict at: 1) printString));
cr;
nextPutAll: ('Link 1 Position: ', ((posDict at: 2) printString));
cr;
nextPutAll: ('Link 2 Position: ', ((posDict at: 3) printString));
cr;
nextPutAll: ('Link 3 Position: ', ((posDict at: 4) printString));
cr;
nextPutAll: ('Link 4 Position: ', ((posDict at: 5) printString));
cr,
nextPutAll: ('Link 5 Position:', (posDict at: 6)
  printString));
cr.

(state = 'idle') ifTrue: [
  stats
    nextPutAll: ('No Part at Robot');
cr
]
ifFalse: [
  partPositionDict keys do: [:each |
    stats
      nextPutAll: ('Position of', (each printString), ':
        ', ((partPositionDict at: each)
          printString));
cr
  ];
].
self openStatsWin!
C.2.2 Passive Entity Classes

C.2.2.1 Part Class

SimulationObjects subclass: #Part
instanceVariableNames:
   'partPositionDict partType enteredAt label roboiNoColl '
classVariableNames:
   'PlaceGenerator '
poolDictionaries: ''

!Part class methods !

generatePlace
| xpt ypt |

"Determines the position of the part according to a probability distribution"

{PlaceGenerator = nil} ifTrue: [  
   PlaceGenerator := Normal mean: 8 deviation: 16
].
xpt := PlaceGenerator next.
ypt := PlaceGenerator next.
\(xpt \oplus ypt)!

icons

\(\{FreeDrawing pictureDictionary at: \{PartRaw\}\)!

new

\(\superclass new)!

nilPlaceGenerator

PlaceGenerator := nil !

!Part methods !

accept: aPart

Menu notify: \{Part object cannot accept a part\}!

appearAt: aPoint for: aName
"Displays the part at the position aPoint"
self displayAt: aPoint for: aName!

| ob |

ob := super copy.
ob isSelected: (isSelected copy).
ob enteredAt: (enteredAt copy).
ob partType: (partType copy).
\ob!

disappearAt: aPoint for: aName
"Displays the part at the position generated by the probability distribution. It is used when the part is dropped from the system"
self displayAt: (aPoint + (Part generatePlace)) for: aName!

| p partKind |

partKind := (PartList at: aName).
p := ((Point new) x: (aPoint x rounded); y: (aPoint y rounded)).
(partKind icons) displayAt: p!

enteredAt
\enteredAt!

enteredAt: aNumber

enteredAt := aNumber!

fromFile
| iconDict |

iconDict := Dictionary new.
iconDict at: '2' put: (FreeDrawing pictureDictionary at:
'PartRaw').
isSelected := SmallIcon new.
isSelected forms: iconDict.
isSelected inOffset: (1@1).
isSelected outOffset: (1@1).
isSelected whichOne: '2'!

iconSelected
\iconSelected!
iconSelected: aForm

iconSelected := aForm!

menu

⌜PromptMenu
prompt: 'Part Menu'
lables: 'Attach to\Detach from\Remove\finish' withCrs
lines: #(3)
selectors: #('attachToInteract' 'detachFrom' 'remove'
'finish')

moveit

⌜nil!

partPositionDict

⌜partPositionDict!

partPositionDict: aDictionary

partPositionDict := aDictionary!

partType

⌜partType!

partType: aPartType

partType := aPartType!

quickFrom: firstPoint to: lastPoint for: aName
| sp |

"Sends the part from firstPoint to lastPoint. The speed of
the part is determined by movesSpeed. The part is shown to
flow animatedly from one icon to another."

sp := (partType runBy) movesSpeed.
(sp := nil) ifTrue: [sp := 50].
(partType icons) quickFrom: firstPoint to: lastPoint speed: sp.
(partType icons) offset: (0@0)!

robotNoColl
robotNoColl

robotNoColl: anOrderedCollection

  robotNoColl := anOrderedCollection! :
C.2.2.2 Part Type Class

SimulationObjects subclass: #PartType
    instanceVariableNames:
        'partPositionDict dimensions prob schList partName
        itsSimulator icons'
    classVariableNames: ''
    poolDictionaries: ''

!PartType class methods!

newPartType: aString icons: aForm runBy: aSimulator dimensions: anArray prob: aProbabilityDist schList: anOrderedCollection
partPositionDict: aDictionary

"Creates an instance of a part type with name aString, controlled by aSimulator; with icon type aForm; dimensions anArray; arrival probability given by aProbabilityDist; schedule plan given by anOrderedCollection and part positions by partPositionDict"

((super new) partName: aString; runBy: aSimulator; icons:
    aForm; dimensions: anArray; prob: aProbabilityDist;
    schList: anOrderedCollection; partPositionDict:
    aDictionary)!!

!PartType methods!

dimensions

!dimensions!

dimensions: anArray

dimensions := anArray!

icons

!icons!

icons: aForm

icons := aForm.
icons offset: (0@0)!

newPart
    | part |
"Creates an instance of a part and provides it with the schedule plan and the part position dictionary"

part := Part new.
part partType: self.
part robotNoColl: (schList copy).
part partPositionDict: (partPositionDict copy).
(itsSimulator = nil) ifFalse: [part enteredAt: (itsSimulator simTime)].

part!

partName

(partName!)

partName: aString

partName := aString!

partPositionDict

(partPositionDict!)

partPositionDict: aDictionary

partPositionDict := aDictionary!

prob

(prob!)

prob: aProbabilityDist

prob := aProbabilityDist!

runBy

(itsSimulator!)

runBy: aSimulator

itsSimulator := aSimulator!

schList

(schList!)

APPENDIX C
schList: anOrderedCollection

schList := anOrderedCollection! !
C.2.3 Interaction Object Classes

C.2.3.1 Assemble Class

SimulationObjects subclass: #Assemble
  instanceVariableNames: '
  classVariableNames: '
  poolDictionaries: '

Assemble class methods !

Assemble methods!

accept: aPart
  Menu notify: 'Assemble object cannot accept a part.'!

copy
  | ob |
  ob := super copySim.
  ob iconSelected: (iconSelected copy).
  @ob!

fromFile
  | iconDict |
  iconDict := Dictionary new.
  iconDict at: '10' put: AssembleForm.
  iconSelected := SimIcons new.
  iconSelected forms: iconDict.
  iconSelected inOffset: (1@1).
  iconSelected outOffset: (1@1).
  iconSelected whichOne: '10'!

iconSelected
  @iconSelected:

iconSelected: aForm
  iconSelected := aForm!

menu
<p>1. Prompt Menu
   - Prompt: 'Assemble Menu'
   - Labels: 'Attach to\Detach from\Remove\finish' with Crs
   - Lines: #1(3)
   - Selectors: #('attachToInteract' 'detachFrom' 'remove'
     'finish')

2. MoveIt
   - nil!

APPENDIX C
C.2.3.2 Part Enter Class

SimulationObjects subclass: #Enter

instanceVariableNames: 'partPositionDict aPane dimArray robotNoColl pane iconName dataFile dimChanged probChanged iconChanged valuesChanged partName fileInput aTopPane schedulePane notAccepted1 notAccepted2 palletType numberSentOn1 numberSentOn2 nextArrival1 nextArrival2 arrivalProbability1 arrivalProbability2 partType2 numberSentOn nextArrival partType arrivalProbability arrivalProbabilityDict notAcceptedDict nextArrivalDict numberSentOnDict notAccepted part '

classVariableNames: 'Spane'

poolDictionaries: '"

!Enter class methods!

pane

*Spane*!!

!Enter methods!

arrivalProbability

arrivalProbability!!

arrivalProbability: aProbability

arrivalProbability := aProbability!!

arrivalProbabilityDict

arrivalProbabilityDict!!

arrivalProbabilityDict: aDictionary

arrivalProbabilityDict := aDictionary!!

arrives: aName

| p n time |

*The part arrives at the Enter Object with a database containing its dimensions, icon type, schedule plan, controller, part position dictionary, and arrival probability.*
The next object is requested for the acceptance of the part and if it does then, it is scheduled for departure. If the part cannot be accepted then, it is thrown out of the system. The arrival of the next part is also scheduled.

(previouObjects isEmpty) ifFalse: [
    numberReceived := numberReceived + 1.
p := itsSimulator newPart: aName.

    arrivalProbability := arrivalProbabilityDict at: aName.
nextArrival := nextArrivalDict at: aName.

    ((numberReceivedDict includesKey: aName)) ifFalse: [
        numberReceivedDict at: aName put: 0
    ].
numberReceived := ((numberReceivedDict at: aName) + 1).
numberReceivedDict at: aName put: numberReceived.

    (graphicsOn) ifTrue: [p appearAt: (self inPosition) for: aName].
    (arrivalProbability = nil) ifTrue: [nextArrival := nil]
ifFalse: [
    n := arrivalProbability next.
time := itsSimulator simTime.
(n < 0) ifTrue: [nextArrival := time] ifFalse: [
    nextArrival := time + n
    ].
nextArrivalDict at: aName put: nextArrival.
itsSimulator scheduleEvent: self at: nextArrival for:
aName
    ].

(nextObjects = nil) ifFalse: [
    (nextObjects accept: p) ifTrue: [

        (numberSentOnDict includesKey: aName) ifFalse: [
            numberSentOnDict at: aName put: 0
        ].
numberSentOn := ((numberSentOnDict at: aName) + 1).
numberSentOnDict at: aName put: numberSentOn.

        iconSelected flash.
        (graphicsOn) ifTrue: [
            p quickFrom: (iconSelected outPosition) to:
            (nextObjects inPosition) for: aName
        ].
nextObjects showAccept.
    (arrivalProbability = nil) ifTrue: [self arrives:
aName].
  ]]

(arrivalProbability = nil) ifTrue: [numberReceived := numberReceived - 1].
(arrivalProbability = nil) ifFalse: [
((notAcceptedDict includesKey: aName)) ifFalse: [
  notAcceptedDict at: aName put: 0
].
notAccepted := ((notAcceptedDict at: aName) + 1).
notAcceptedDict at: aName put: notAccepted.

(graphicsOn) ifTrue: [p disappearAt: (iconSelected in:Position) for: aName].
iconSelected show
].
ifTrue: [Menu notify: 'Part icon should be attached to IN '.
  nil]]]

aTopPane

:aTopPane!

aTopPane: aPane

aTopPane := aPane!

checkSize: aString
  "Checks whether the size of the part is within the limits of
  0.5 to 1.5"

  x := aString as: Float.
  [x between: 0.5 and: 1.5] whileFalse: [
    x := Prompter
    prompt: 'Size should be between 0.5 and 1.5'
    default: '1'.
    (x = nil) ifFalse: [x := x asFloat]
    ifTrue: [x := 2].
  ].
  x]

closeIt

"Closes the Schedule Plan Pane and returns the control to the Simulation Pane"

aTopPane dispatcher closeItSim.
aTopPane dispatcher deactivateWindow.
Scheduler displayAll.
self createPartType!

closeWindow

Pane windowClip: (aTopPane windowFrame).
aTopPane close.
Scheduler remove: aTopPane dispatcher!

```python
| ob |

"Copies the instance variables and data structures of Enter
Object to store in the model base along with the object"

ob := super copySim.
ob arrivalProbability: (arrivalProbability copy).
ob schedule: (arrivalProbability1 copy).
ob partType: partType.
ob iconSelected: (iconSelected copy).
ob numberSentOn: (numberSentOn copy).
ob nextArrival: (nextArrival copy).
ob notAccepted: (notAccepted copy).
ob paletteType: (paletteType copy).
ob arrivalProbability: (arrivalProbability copy).
ob numberReceived: (numberReceived copy).
ob robotNoColl: (robotNoColl copy).
ob pane: (pane copy).
ob (iconName: (iconName copy).
ob dataFile: (dataFile copy).
ob dimChanged: (dimChanged copy).
ob probChanged: (probChanged copy).
ob iconChanged: (iconChanged copy).
ob valuesChanged: (valuesChanged copy).
ob partName: (partName copy).
ob fileInput: (fileInput copy).
ob aTopPane: (aTopPane copy).
ob schedulePane: (schedulePane copy).
ob arrivalProbabilityDict: (arrivalProbabilityDict copy).
ob nextArrivalDict: (nextArrivalDict copy).
ob numberSentOnDict: (numberSentOnDict copy).
ob notAcceptedDict: (notAcceptedDict copy).

`ob`
```

createDataFile: aFile with: anArray
| x y z |

"Creates a data file containing the points coordinates, edge
vectors, face normals and surface vectors of a part"

x := anArray at: 1.
y := anArray at: 2.
z := anArray at: 3.

aFile
nextPutAll: ("1 0 0 0");
cr;
nextPutAll: ("2", ",", (x printString), ",", "0 0");
cr;
nextPutAll: ("3", ",", (x printString), ",", (y
printString), ',', '0');
cr;
nextPutAll: ('4 0', ',', (y printString), ',', '0');
cr;
nextPutAll: ('5 0', ',', (y printString), ',', (z printString));
cr;
nextPutAll: ('6', ',', (x printString), ',', (y printString), ',', '0', ',', (z printString));
cr;
nextPutAll: ('7 0 0', ',', (z printString));
cr;
nextPutAll: ('8', ',', (x printString), ',', '0', ',', '0', ',', (z printString));
cr;
nextPutAll: ('1 1 2');
cr;
nextPutAll: ((x printString), ',', '0 0');
cr;
nextPutAll: ('2 2 3');
cr;
nextPutAll: ('0', ',', (y printString), ',', '0');
cr;
nextPutAll: ('3 4 3');
cr;
nextPutAll: ((x printString), ',', '0 0');
cr;
nextPutAll: ('4 1 4');
cr;
nextPutAll: ('0', ',', (y printString), ',', '0');
cr;
nextPutAll: ('5 5 4');
cr;
nextPutAll: ('0 0', ',', '-', (z printString));
cr;
nextPutAll: ('6 6 5');
cr;
nextPutAll: ('-', (x printString), ',', '0 0');
cr;
nextPutAll: ('7 6 3');
cr;
nextPutAll: ('0 0', ',', '-', (z printString));
cr;
nextPutAll: ('8 7 5');
cr;
nextPutAll: ('0', ',', (y printString), ',', '0');
cr;
nextPutAll: ('9 1 7');
cr;
nextPutAll: ('0 0', ',', (z printString));
cr;
nextPutAll: ('10 8 7');
cr;
nextPutAll: ('-', (x printString), ',', '0 0');
cr;
nextPutAll: ('11 6 8');
createPartType
| which |

APPENDIX C
"Pops up the Part Definition menu and decides which option was selected"

which := self partEnterMenu popUpAt: (Cursor offset).
(which = nil) ifFalse: [
  self partMenuResponse: which
]

createPartType: aName
| array index x y z p partKind names finishChosen |

"The user is presented with the Part Definition Menu which has options to select dimensions, select arrival probability, create schedule plan and select or create icon of the part. The definition of the part can then be saved into the PartList database"

(PartList includesKey: aName) ifTrue: [
  partKind := (PartList at: aName) partType.
array := Array new: 3.
finishChosen := false.

[finishChosen] whileFalse: [
index := self partEnterMenu popUpAt: (Cursor offset).
(index notNil) ifFalse: [
  index := self partEnterMenu popUpAt: (Cursor offset)
]
ifTrue: [
  (index = 1) ifTrue: [
    dataFile := Prompter
    prompt: 'Enter file name (max. 8 letters)'
    default: 'dataFile'.
x := Prompter
    prompt: 'Length of part?'
    default: '1'.
y := Prompter
    prompt: 'Width of part?'
    default: '1'.
z := Prompter
    prompt: 'Height of part?'
    default: '1'.
array at: 1 put: x asFloat.
array at: 2 put: y asFloat.
array at: 3 put: z asFloat.
self createDataFile: dataFile with: array.
dimensionDict at: aName put: dataFile.
dimChanged := true.
].
(index = 2) ifTrue: [
  self probab: aName.
  probChanged := true.
].

APPENDIX C
(index = 3).isTrue: [  
schedulePane := self openScheduleWin.  
schedulePane pane  
  change: #fileIt with: ;  
  menu: #schedulePaneMenu;  
  name: #inputBlankString;  
  model: self.  

schedulePane scheduleWindow  
].
(index = 4).isTrue: [  
  "FormList removeKey: 'PartForm';"  
  names := FormList keys asSortedCollection  
  asArray,  
  (Array with: 'create new').  
  iconName := (Menu  
    labelArray: names  
    lines: Array new  
    selectors: names) popupAt:  
      Cursor offset.  

(iconName notNil).isTrue: [  
  iconName = 'create new' ifTrue: [  
    iconName := Prompter  
      prompt: 'Enter name of icon to create'  
      default: 'anIcon'.  
    "FreeDrawing 1."  
    iconName isNil ifTrue: [sself].  
    "formDict at: aName put: iconName.  
    FormList at: iconName put: iconName."  
    iconChanged := true.  
    self openDrawingWin.  
  ].  

formDict at: aName put: iconName.  
  "FormList at: iconName put: (FormList at:  
    iconName).  
  "iconChanged := true.  
].
(index = 5).isTrue: [  
  ((iconChanged = false) and: [valuesChanged =  
false]) ifTrue: [  
    iconName := (FormList keyAtValue:  
PartForm).  
formDict at: aName put: iconName.  
FormList at: iconName put: PartForm  
].  

((iconChanged = false) and: [valuesChanged =  
true]) ifTrue: [  
    iconName := (FormList keyAtValue: (partKind  
icons)).  
formDict at: aName put: iconName.  
FormList at: iconName put: (partKind icons)  
].  

((dimChanged = false) and: [valuesChanged =
false) ifTrue: [
  array aAtAllPut: 1.
  dataFile := Disk newFile: 'dataFile'.
  self createDataFile: dataFile with: array.
  dimensionDict at: aName put: dataFile
].
((dimChanged = false) and: [valuesChanged = true]) ifTrue: [
  dataFile := partKind dimensions.
  dimensionDict at: aName put: dataFile
].
((probChanged = false) and: [valuesChanged = false]) ifTrue: [
  p := Normal mean: 3.6 deviation: 1.5.
  probabDict at: aName put: p
].
((probChanged = false) and: [valuesChanged = true]) ifTrue: [
  p := partKind prob.
  probabDict at: aName put: p
].
(valuesChanged = true) ifTrue: [
  fileInput := partKind schList.
  scheduleListDict at: partName put: fileInput.
  SchedulePlanList at: partName put: fileInput.
].

x := (FormList at: iconName).
y := FormList.
itsSimulator partTypes at: aName
  put: (PartType newPartType: aName icons:
    (FormList at: iconName)
    runBy: itsSimulator dimensions: dataFile
    prob: p schList: fileInput)

partSet add: {(itsSimulator partTypes at: aName) newPart}.
PartList at: aName put: {(itsSimulator
  partTypes at: aName) newPart}.
finishChosen := true
].
]

\[\text{dataFile}\]
\[\$\text{dataFile!}\]

\[\text{dataFile: aStream}\]
\[\text{dataFile := aStream!}\]
dimChanged

c!dimChanged!

dimChanged: aBoolean

dimChanged := aBoolean!

dimensionDict

c!dimensionDict!

dimensions

|x y z x1 y1 z1 array file|

"Prompts the user for the dimensions of the part and makes sure they are within limits"

array := Array new: 3.
file := Prompter
    prompt: 'Enter file name (max. 8 letters)'
    default: 'dataFile'.
x := Prompter
    prompt: 'Length of part?'
    default: '1'.
x1 := self checkSize: x.
y := Prompter
    prompt: 'Width of part?'
    default: '1'.
y1 := self checkSize: y.
z := Prompter
    prompt: 'Height of part?'
    default: '1'.
z1 := self checkSize: z.

array at: 1 put: x1 asFloat.
array at: 2 put: y1 asFloat.
array at: 3 put: z1 asFloat.

self createDataFile: dataFile with: array.
dimArray := self readDataOf: file.
dimensionDict at: partName put: dataFile.
dimChanged := true.
self createPartType!

doIt: aName

self arrives: aName!

fileInput
fileInput!: aStream

fileInput := aStream!

filelet: aString with: aDispatcher
| aFile |

"Creates a file and stores the schedule plan into it"
aFile := Prompter
  prompt: 'Enter file name (max. 8 letters)'
  default: 'aFile'.
aDispatcher pane fileOutOn: fileInput.
scheduleListDict at: partName put: fileInput.
SchedulePlanList at: partName put: fileInput.
fileInput close!

finishMenu
| partKind array x trpart |

"After the part type has been defined the finish option will store this part into the PartList database. If certain characteristics of the part have not been defined then they are given a default value"

array := Array new: 3.
(PartList includesKey: partName) ifTrue: [
  partKind := (PartList at: partName)].
((iconChanged = false) and: [valuesChanged = false]) ifTrue: [i
  iconName := (FormList keyAtValue: PartForm).
  formDict at: partName put: iconName.
  FormList at: iconName put: PartForm
].
((iconChanged = false) and: [valuesChanged = true]) ifTrue: [i
  iconName := (FormList keyAtValue: (partKind icons)).
  formDict at: partName put: iconName.
  FormList at: iconName put: (partKind icons)
].

((dimChanged = false) and: [valuesChanged = false]) ifTrue: [array atAllPut: 1.
  dataFile := Disk newFile: 'dataFile'.
  self createDataFile: dataFile with: array.
  dimArray := self readDataOf: 'dataFile'.
  dimensionDict at: partName put: dimArray
].
((dimChanged = false) and: [valuesChanged = true]) ifTrue: [dimArray := partKind dimensions.
  dimensionDict at: partName put: dimArray
].

APPENDIX C 138
{(probChanged = false) and: (valuesChanged = false)} ifTrue: [
    p := Normal mean: 3.5 deviation: 1.5.
    probabDict at: partName put: p
].

{(probChanged = false) and: (valuesChanged = true)} ifTrue: [
    p := partKind prob.
    probabDict at: partName put: p
].

(valuesChanged = true) ifTrue: [
    robotNoColl := partKind schList.
    scheduleListDict at: partName put: fileInput.
    SchedulePlanList at: partName put: fileInput
].

tpart := self readDataOf: 'trparta'.
x := part translate: dimArray by: tpart.
partPositionDict at: partName put: x.

itsSimulator partTypes at: partName
    put: (PartType newPartType: partName icons: (FormList at: iconName)
        runBy: itsSimulator dimensions: dimArray prob: p schList:
        robotNoColl

partSet add: ((itsSimulator partTypes at: partName) newPart).

PartList at: partName
    put: (PartType newPartType: partName icons: (FormList at: iconName)
        runBy: itsSimulator dimensions: dimArray prob: p schList:
        robotNoColl

freeDrawingMenu

cMenu
    labels: 'label\collapse\cycle\frame\move\close' withCrs
    lines: #()
    selectors: #(newLabel collapse cycle resize move closeIt)!
iconChanged
    $iconChanged!

iconChanged: aBoolean
    iconChanged := aBoolean!

iconName
    $iconName!

iconName: aString
    iconName := aString!

IconSelected
    $IconSelected!

IconSelected: aForm
    IconSelected := aForm!

initialize
    super initialize.
    nextArrival := 0.
    formDict := Dictionary new.
    scheduleListDict := Dictionary new.
    probabDict := Dictionary new.
    dimensionDict := Dictionary new.
    notAcceptedDict := Dictionary new.
    arrivalProbabilityDict := Dictionary new.
    numberSentOnDict := Dictionary new.
    nextArrivalDict := Dictionary new.
    partPositionDict := Dictionary new.
    valuesChanged := false.
    dimChanged := false.
    probChanged := false.
    iconChanged := false.
    partSet := Set new.
    SimulationObjects initPartList.
    part := Geometric new.
    self initializeParts!

initializeParts
(PartList isEmpty) ifFalse: [
    PartList do: [: each |
        itsSimulator partTypes at: each put: (PartList at:
            each)].
]

inputBlankString

nil!!

menu

|prompt: 'Object Menu'
|labels: 'Attach to\Detach from\Remove\View statistics\Part type\finish' withCr
|lines: #(3)
|selectors: #('attachTo' 'detachFrom' 'remove'
    'viewStatistics' 'TypeOfPart' 'finish')!

moveit

(arrivalProbability = nil) ifTrue: [nil]!

nextArrival:

|nextArrival!

nextArrival: aNumber

nextArrival := aNumber!

nextArrivalDict

|nextArrivalDict!

nextArrivalDict: aDictionary

nextArrivalDict := aDictionary!

notAccepted

|notAccepted!

notAccepted: aBoolean

notAccepted := aBoolean!
notAcceptedDict

?notAcceptedDict!

notAcceptedDict: aDictionary

notAcceptedDict := aDictionary!

numberSentOn

?numberSentOn!

numberSentOn: aNumber

numberSentOn := aNumber!

numberSentOnDict

?numberSentOnDict!

numberSentOnDict: aDictionary

numberSentOnDict := aDictionary!

openDrawingWin

"Opens a free drawing window and provides it with tools to create an icon."

Scheduler topDispatcher deactivateWindow.

topPane := TopPane new
    model: self;
    menu: #freeDrawingMenu;
    label: 'Icon Editor';
    minimumSize: Display extent // 3;
    yourself.

topPane addSubpane:
    (Pane :=
        FreeDrawPane new
        model: FreeDrawing new;
        name: #initialize;
        menu: #menuSim;
        change: #change).

topPane dispatcher
    open;
    activate;
    display.
Scheduler add: topPane dispatcher.
topPane dispatcher searchForActivePane!

openScheduleWin
| aPane |
"Opens the Schedule Plan Pane and provides it with editing
tools to create the schedule plan"

aPane := TextPane new.
aPane
dispenser: aPane dispatcher.

aTopPane := TopPane new.
aTopPane
label: 'Schedule Plan';
model: self;
menu: #topScheduleMenu;
minimumSize: 24@48;
righticons: #(resize collapse zoom);
forecolor: 10;
yourself;
addSubpane: aPane.
aTopPane dispatcher
openIn: (225 @ 150 extent: 250 @ 200);
window.
Scheduler add: aTopPane dispatcher.
cpane dispatcher!

pane

cpane!

pane: aPane

pane := aPane!

partEnterMenu

cPromptMenu
prompt: 'Part Definition Menu'
labels: 'Dimensions\Probability\Schedule Plan\Select
ic\finish' withCr
lines: #()
selectors: #('dimensions' 'probability' 'schedulePlan'
'select\icons' 'finishMenu')!

partMenuResponse: aString
"Determines the option selected by the user and then executes
the corresponding method"

(aString = 'dimensions') ifTrue: [self dimensions].
(aString = 'probability') ifTrue: [self probability].
(aString = 'schedulePlan') ifTrue: [self schedulePlan].
(aString = 'select\icons') ifTrue: [self select\icons].
(aString = 'finishMenu') ifTrue: [self finishMenu]"
partName

case partName!

partName: aString

partName := aString!

partSelection

| names index aValue which |

"Requests the user to select or create a part type. If the user wants to create a new part type then, he or she is requested for a part name and then, the Part Definition Menu pops up. If an existing part type is selected then the user can select or remove it from the database."

index := 0.
dimChanged := false.
probChanged := false.
iconChanged := false.

names := PartList keys asSortedCollection asArray.
(Array with: 'create new').
partName := (Menu
    labelArray: names
    lines: Array new
    selectors: names) popUpAt: Cursor offset.

(partName notNil) ifTrue: [
    partName = 'create new' ifTrue: [
        partName := Prompter prompt: 'Part name'
            default: 'aPart'.
        partName isNil ifTrue: [#self].
        self createPartType
    ]
    ifFalse: [
        aValue := self selectPartMenu popUpAt: (Cursor offset).
        (aValue notNil) ifTrue: [
            (aValue = 1) ifTrue: [
                self view.
                self selectPartMenu
            ].
            (aValue = 2) ifTrue: [
                partSet add: ((itsSimulator partTypes at: partName) newPart)
            ].
            (aValue = 3) ifTrue: [
                valuesChanged := true.
                self createPartType
            ].
        ]
    ].
partType: aString
  partType := aString!

probab
  self scheduleInteract.
  itsSimulator startUp: self!

probab: aName
  self scheduleInteract: aName!

probabDict
  @probabDict!

probability
  self scheduleInteract: partName.
  probChanged := true.
  self createPartType!

probChanged
  @probChanged!

probChanged: aBoolean
  probChanged := aBoolean!

readSchedule: file
  "Reads the schedule plan from the file and stores each command into the output collection. The output is then returned to the sender"
  input := File pathName: file.
  output := OrderedCollection new.
[input atEnd]
  whileFalse: [
    w := input nextWord.
    (w = "robot") ifTrue: [
      x := input nextWord.
      y := x asInteger.
      (y isKindOf: 'integer') ifTrue: [
        output add: y
      ].
    ].
  ]

resetStats

super resetStats.
notAccepted := 0.
notAccepted1 := 0.
notAccepted2 := 0.\nnumberSentOn := 0.
numberSentOn1 := 0.
numberSentOn2 := 0.
(itsSimulator = nil) ifFalse: [
  (nextArrival = nil) ifFalse: [
    nextArrival := nextArrival - (itsSimulator simTime)
  ]
]

retrieveIt
  | afile |

  afile := Prompter
    prompt: 'Enter file name (max. 8 letters)'
    default: 'aFile'.
afile := Disk file: afile.
aPane fileInFrom: afile!

robotNoColl

\$robotNoColl!

robotNoColl: anOrderedCollection

  robotNoColl := anOrderedCollection!

runBy: aSimulation
super runBy: aSimulation.
itsSimulator scheduleEvent: self at: (itsSimulator simTime)
for: partName!

sched

nil!

schedule: aProbabilityDistribution

| time n |

"Schedules the arrival of a part according to the probability
aProbabilityDistribution. The arrival probability is stored
in arrivalProbabilityDict and the next arrival time in
nextArrivalDict"

(itsSimulator = nil) ifTrue: [nil],

time := itsSimulator simTime,
arrivalProbabilityDict at: partName put: arrivalProbability.
probabDict at: partName put: arrivalProbability.

" (time = 0) ifTrue: [itsSimulator deSchedule: self]."

(arrivalProbability = nil) ifTrue: [nil],
n := arrivalProbability next.

(n < 0) ifTrue: [nextArrival := time] ifFalse: [
nextArrival := (time + n)
],
nextArrivalDict at: partName put: nextArrival,
itsSimulator scheduleEvent: self at: nextArrival for: partName!

scheduleInteract: aName

| s button buttonLoc sBback rect |

"Prompts the user to select a probability distribution from
the Probability Distribution Menu"

button := 'Select Arrival Probability for Part'.
buttonLoc := 130@6 + (0@((SysFont charSize y)));
rect := Rectangle origin: buttonLoc extent: ((
    Point new) x: ((button size) * (SysFont charSize
    y)));
y: (SysFont charSize y)),
sBback := ColorForm fromDisplay: rect.
partName := aName.
button displayAt: buttonLoc.
s := ProbabilityDist fromUser.
sBback display.

(s = nil) ifTrue: [s := (Normal mean: 3.6 deviation: 1.6)]
(s isKindOf: Integer) ifTrue: [self schedule: (Normal mean: 3.6
deviation: 1.6)]

ifFalse: [self schedule: s]!
scheduleListDict!
schedulePane
    schedulePane!
schedulePane: aPane
    schedulePane := aPane!
schedulePaneMenu
    Menu
defaultMenuLabels: 'copy|cut|paste|save|retrieve' with Crs
    lines: #(3)
defaultMenuSelectors: #(copySelection cutSelection pasteSelection
    accept retrieve())!
schedulePlan
"Opens the Schedule Plan Pane"
Scheduler topDispatcher deactivateWindow.
aPane := TextPane new.
aPane
dispatcher: aPane dispatcher.
aTopPane := TopPane new.
aTopPane
defaultMenuLabel: 'Schedule Plan';
defaultMenuModel: self;
defaultMenuMenu: #topScheduleMenu;
defaultMenuMinimumSize: 24@48;
defaultMenuRightIcons: #(resize collapse zoom);
defaultMenuForeColor: 10;
defaultMenuYourself.
aTopPane addSubPane:
    (aPane :=
        TextPane new
        change: #filet with: ;
defaultMenuMenu: #schedulePaneMenu;
defaultMenuName: #inputBlankString;
defaultMenuModel: self).
aTopPane dispatcher
open;
activate;
display.
Scheduler add: aTopPane dispatcher.
aTopPane dispatcher searchForActivePane!

selecticons
names := FormList keys asSortedCollection asArray.  
    (Array with: 'create new').
iconName := (Menu 
    labelArray: names 
    lines: Array new 
    selectors: names) popUpAt: Cursor offset.

(iconName notNil) ifTrue: [ 
    iconName = 'create new' ifTrue: [ 
        iconName := Prompter 
        prompt: 'Enter name of icon to create' 
        default: 'anIcon'.
    iconName isNil ifTrue: [self].
    formDict at: partName put: iconName.
    iconChanged := true.
    self openDrawingWin.
    ]
    ifFalse: [ 
        formDict at: partName put: iconName.
        iconChanged := true.
        self createPartType 
    ].
]

selectPartMenu
gMenu
    prompt: 'Part Selection Menu'
    labels: 'View data\Select\Change values\Remove' withCrs
    lines: #(1 2 3 4)
    selectors: #(1 2 3 4)

topScheduleMenu
gMenu
    labels: 'color\label\collapse\cycle\frame\move\close' withCrs
    lines: #(1)
    selectors: #(color newLabel collapse cycle resize move close)

typeOfPart
goOfParts count button buttonLoc sBack rect]

    'Prompts the user for the number of part types entering and then allows the user to define them'
    count := 1.
    goOfParts := Prompter
prompt: 'Number of part types entering'
defaultExpression: '1'.

[partSet size = noOpParts] whileFalse: [
    button := ('Select Part Type ', (count printString)).
    buttonLoc := 130@6 + (0@((SysFont charSize y)).
    rect := Rectangle origin: buttonLoc extent: ((
        Point new) x: ((button size) * (SysFont
        charSize x)));
    y: (SysFont charSize y)).
    sBback := ColorForm fromDisplay: rect.
    button displayA1: buttonLoc.
    self partSelection.
    sBback display.
    count := count + 1
].
itsSimulator startUp: self!

valuesChanged

@valuesChanged!

valuesChanged: aBoolean

valuesChanged := aBoolean!

viewStatistics

stats := WriteStream on: String new.
    stats
    nextPutAll: 'STATISTICS OF ENTER:'.
    cr;
    cr.
    PartList keys do: [: each |
        stats
        nextPutAll: ('Name of Part Arriving ', ': ', (each
        printString));
        cr
    ].
    probabDict do: [: each |
        stats
        nextPutAll: ('Arrival Probability of ', ((probabDict
        keyAtValue: each) printString), ': ', (each
        printString));
        cr
    ].
    nextArrivalDict keys do: [: each |
        stats
        nextPutAll: ('Next Arrival Time of ', (each
        printString), ': ', ((nextArrivalDict at:
        each) printString));
        cr
    ].
numberReceivedDict keys do: [: each |
    stats
    nextPutAll: ('Number Received of ', (each printString), ' ', ((numberReceivedDict at: each) printString));
    cr
].

numberSentOnDict keys do: [: each |
    stats
    nextPutAll: ('Number Sent of ', (each printString), ' ', ((numberSentOnDict at: each) printString));
    cr
].

self openStatsWin.
itsSimulator startup: self !
C.2.3.3 Part Exit Class

SimulationObjects subclass: #Exit
    instanceVariableNames:
        'timeInSystem'
    classVariableNames: ''
    poolDictionaries: '!' 

!Exit class methods !

!Exit methods !

accept: aPart
  | n waitingToShowAccept name |
  "Accepts all parts arriving to the object and updates statistics on itself"
  waitingToShowAccept := aPart.
  name := aPart partType partName.
  (numberReceivedDict includesKey: name) ifFalse: [ 
    numberReceivedDict at: name put: 0
  ].
  numberReceived := (numberReceivedDict at: name) + 1.
  numberReceivedDict at: name put: numberReceived.
  n := (llitsSimulator simTime) - (aPart enteredAt).
  timeInSystem := timeInSystem + n.
  !true!

attachTo: anObject

  Menu notify: 'OUT cannot be attached to any object.'!!

copy
  | ob |
  ob := super copySim.
  ob iconSelected: (iconSelected copy).
  ob timeInSystem: (timeInSystem copy).
  ob numberReceived: (numberReceived copy).
  ob numberReceivedDict: (numberReceivedDict copy).
  ob!

fromFile
  | iconDict |
  iconDict := Dictionary new.
  iconDict at: '6' put: (FreeDrawing pictureDictionary at: 'ExitForm').
iconSelected := SimIcons new.
iconSelected forms: iconDict.
iconSelected inOffset: (1@1).
iconSelected outOffset: (1@1).
iconSelected whichOne: '6'.

iconSelected.

!iconSelected!

iconSelected: aForm

iconSelected := aForm!

menu

!PromptMenu
prompt: 'Exit Menu'
labels: 'Detach from\Remove\View statistics\finish' withCrs
lines: #(3)
selectors: #("detachFrom" remove "viewStatistics"
"finish")

moveit

!nil!

resetStats

timelnSystem := 0.
numberReceived := 0.
super resetStats!

showAccept

(waitingToShowAccept = nil) ifFalse: [
iconSelected flash.
waitingToShowAccept := nil
].

timelnSystem

!timelnSystem!

timelnSystem: aNumber

timelnSystem := aNumber!
viewStatistics
    | x |
    (numberReceived = 0) ifFalse: [
        x := (time/$nSystem/numberReceived) asFloat
    ] ifTrue: [x := 0].

stats := WriteStream on: String new.
stats
    nextPutAll: 'STATISTICS OF EXIT';
    cr;
    cr;
    nextPutAll: ('Average time in the system: ', (x printString));
    cr;
    nextPutAll: ('No. of pallets received: ', (numberReceived printString));
    cr.
numberReceivedDict keys do: [:each |
    stats
        nextPutAll: ('Number Received of ', (each printString), ': ', (numberReceivedDict at: each) printString));
        cr
].

self openStatsWin.
itsSimulator startUp. self!}
C.2.3.4 Collision Check Class

**SimulationObjects subclass: #Grasp**

instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: "" !

!Grasp class methods !!

!Grasp methods !

accept: aPart

Menu notify: 'Grasp cannot accept a part'.
false!

copy

| ob |

ob := super copySim.
ob iconSelected: (iconSelected copy).
| ob |

fromFile

| iconDict |

iconDict := Dictionary new.
iconDict at: '7' put: (FreeDrawing pictureDictionary at:
'GraspForm').
iconSelected := SimIcons new.
iconSelected forms: iconDict.
iconSelected inOffset: (1@1).
iconSelected outOffset: (1@1).
iconSelected whichOne: '7'!

iconSelected

| iconSelected: |

iconSelected: aForm

iconSelected := aForm!

menu

| PromptMenu |
prompt: 'Grasp Menu'
labels: 'Attach to' 'Detach from' 'Remove' 'finish' withCrs
lines: #3
selectors: #('attachToInteract' 'detachFrom' 'remove' 'finish')!
C.2.3.5 Transfer Class

SimulationObjects subclass: #Transfer
  instanceVariableNames: 
    'robotNoColl partName palletType robotPresent convPresent robDict convDict nextObjDict tempPlace capacity nextObjSet '
  classVariableNames: ''
  poolDictionaries: '' !

!Transfer class methods ! !

!Transfer methods !

accept: aPart
| name |

"The Transfer Object accepts a part only if it does not have a part already. The schedule plan of the part is also read"

(tempPlace size < capacity) ifTrue: [ 
  tempPlace add: aPart.
  name := aPart partType partName.
  robotNoColl := aPart robotNoColl.
  itsSimulator scheduleEvent: self at: (itsSimulator simTime)
  for: name.
  \
  \n  \false
] 
ifFalse: [false]!

attachTo: anObject
| x |

"Attaches the Transfer Object to active entity objects by following the Rules of Interaction. Also the identity of the conveyors and robots attached to the object are stored in convDict and robDict respectively. Also nextObjSet contains an instance of all the objects connected to the Transfer Object"

(self isMemberOf: Part) ifFalse: [ 
  (anObject isMemberOf: Enter) ifTrue: [ 
    Menu notify: 'Only Part icon can be attached to IN'.
    fail
  ].
].

(s self isMemberOf: Part) ifTrue: [
(anObject isMemberOf: Enter) ifFalse: [
    Menu notify: 'Part icon can only be attached to IN'.
    'nil
].
].

(anObject isMemberOf: Part) ifTrue: [
    Menu notify: 'Cannot attach anything to Part'.
    'nil
].
.

(self isMemberOf: Assemble) ifTrue: [
    (anObject isKindOf: Robot) ifFalse: [
        Menu notify: 'Assemble can only be attached to a Robot'.
        'nil
    ].
].

(anObject isMemberOf: Assemble) ifTrue: [
    Menu notify: 'Cannot attach anything to Assemble icon'.
    'nil
].
.

(self isMemberOf: Exit) ifTrue: [
    Menu notify: 'Cannot attach Exit icon to anything'.
    'nil
].
.

(self isMemberOf: Enter) ifTrue: [
    (anObject isKindOf: Conveyor) ifFalse: [
        (anObject isKindOf: Robot) ifFalse: [
            Menu notify: 'Interaction icons cannot be attached to each other'.
            'nil
        ].
    ].
].

(self isMemberOf: Grasp) ifTrue: [
    (anObject isKindOf: Conveyor) ifFalse: [
        (anObject isKindOf: Robot) ifFalse: [
            Menu notify: 'Interaction icons cannot be attached to each other'.
            'nil
        ].
    ].
].

(self isMemberOf: Transfer) ifTrue: [
    (anObject isKindOf: Conveyor) ifFalse: [
        (anObject isKindOf: Robot) ifFalse: [
            Menu notify: 'Interaction icons cannot be attached to each other'.
            'nil
        ].
    ].
].
(anObject isKindOf: Conveyor) ifTrue: [
  (nextObjDict includes: 'aConveyor')
  or: [(nextObjDict includes: 'aKinConveyor')
    or: [nextObjDict includes: 'aGeoConveyor']]
  ifFalse: [
    iconSelected addPointTo: (anObject inPosition).
    (convDict isEmpty) ifTrue: [
      convDict at: 1 put: anObject]
    ifFalse: [convDict at: 2 put: anObject].
    nextObjects := anObject.
    nextObjSet add: anObject.
    nextObjDict add: anObject.
    anObject receivesFrom: self]
  ifTrue: [
    Menu notify: 'Two conveyors cannot be a connected to Transfer'.
    @nil].
]

x := robDict size.
(anObject isKindOf: Robot) ifTrue: [
  robDict at: (x + 1) put: anObject.
  iconSelected addPointTo: (anObject inPosition).
  nextObjects := anObject.
  nextObjSet add: anObject.
  nextObjDict add: anObject.
  anObject receivesFrom: self]

capacity

@capacity!

capacity: anInteger

capacity := anInteger!

convDict

@convDict!

convDict: aDictionary

convDict := aDictionary!

convPresent

@convPresent!
convPresent: aBoolean

convPresent := aBoolean!

copy
| ob |

ob := super copySim.
ob palletType: (palletType copy).
ob robotPresent: (robotPresent copy).
ob convPresent: (convPresent copy).
ob robDict: (robDict copy).
ob convDict: (convDict copy).
ob nextObjDict: (nextObjDict copy).
ob tempPlace: (tempPlace copy).
ob capacity: (capacity copy).
ob nextObjSet: (nextObjSet copy).
ob iconSelected: (iconSelected copy).
| ob |
detachFrom: anObject
| y |

"Permits the detaching of objects from the Transfer Object. This method is used exclusively by the Transfer Object. After the objects are detached their instances are removed from each others databases."

(nextObjDict = nil) ifTrue: [cnil].

(anObject isKindOf: Conveyor) ifTrue: [
  convDict := nil].

(robDict includes: anObject) ifTrue: [
  y := robDict keyAtValue: anObject.
  (y = (robDict size)) ifTrue: [
    robDict removeKey: y
  ]
  ifFalse: [
    y to: ((robDict size) - 1) do: [: i |
      robDict at: i put: (robDict at: (i + 1))
    ],
    robDict removeKey: (robDict at: i) ifAbsent: []
  ].
].

self nextObjDict remove: anObject ifAbsent: [].
anObject previousObjDict remove: (self label) ifAbsent: [].
iconSelected removePointTo: (nextObjects inPosition).
anObject doesNotReceiveFrom: self!

detachToInteract
| ob nextPoint |

"Prompts the user to click on the object to detach the
Transfer object from

(itsSimulator = nil) ifTrue: [return].
((itsSimulator objectCount) = 1) ifTrue: [return].
nextPoint := self promptClick:
  "Click on object to detach from."
ob := itsSimulator findObjectAt: nextPoint.
(ob = nil) ifFalse: [
  (ob = self) ifFalse: [self detachFrom: ob]
].
itsSimulator startUp: self!

doIt: aName

partName := aName.\n(nextObjDict isEmpty) ifFalse: [self moveIt!]

fromFile
  |iconDict|

  iconDict := Dictionary new.
  iconDict at: '9' put: (FreeDrawing pictureDictionary at:
    'TransferForm').
  iconSelected := SimIcons new.
  iconSelected forms: iconDict.
  iconSelected inOffset: (1@1).
  iconSelected outOffset: (1@1).
  iconSelected whichOne: '9'!

iconSelected

iconSelected := aForm

iconSelected := aForm:

initialize

nextObjSet := Set new.
nextObjDict := Bag new.
tempPlace := OrderedCollection new.
capacity := 1.
robDict := Dictionary new.
convDict := Dictionary new.
super initialize!

menu

prompt: 'Transfer Menu'
labels: 'Attach to\nDetach from\nRemove\nfinish' withCrs
lines: #(3)
selectors: #('attachToInteract' 'detachFrom' 'remove'
'finish')!

moveIt
  | firstPart x y z array number robotTypeFound goToConv|

"Reads the schedule plan of the object and then scans all the
attached robot objects to find a match between the
identification number of the robot and the first number on the
schedule plan. If a match is found then, the part is passed
to the robot object, otherwise it is thrown out of the system
or passed to the conveyer object"

robotTypeFound := false.
goToConv := true.
array := Array new: (robDict size).

(tempPlace size > 0) ifTrue: []
  firstPart := tempPlace removeFirst.
  "robotNoColl := (firstPart partType schList) copy."
  (convDict isEmpty) ifFalse: [convPresent := true]
    ifTrue: [convPresent := false].
  (robDict size = 0) ifFalse: []
    robotPresent := true.
    1 to: robDict size do: [:i |
      array at: i put: (((robDict at: i) robotNo)
        asInteger)
    ].
  ifTrue: [robotPresent := false].
((convPresent = true) and: [robotPresent = false]) ifTrue: []
  ((convDict at: 1) accept: firstPart) ifTrue: []
    iconSelected flash.
    firstPart quickFrom: (iconSelected outPosition)
    to: ((convDict at: 1) inPosition) for: partName
  ifFalse: [tempPlace addFirst: firstPart].
]

((convPresent = true) and: [robotPresent = true]) ifTrue: []
  (robotNoColl isNil) ifFalse: []
    number := robotNoColl removeFirst.
    1 to: (array size) do: [:i |
      (((array at: i) = number) and: [robotTypeFound
        = false]) ifTrue: []
        (robDict at: i) accept: firstPart) ifTrue: []
          robotTypeFound := true.
          iconSelected flash.
          firstPart quickFrom: (iconSelected outPosition)
          to: ((robDict at: i) inPosition)
]
for: partName
]
ifFalse: [
tempPlace addFirst: firstPart.
robotNoColl addFirst: number.
goToConv := false
].
].
].
((robotTypeFound = false) and: [goToConv = true]) ifTrue: [
((convDict at: 1) accept: firstPart) ifTrue: [
(robotNoColl isNil) ifFalse: [
robotNoColl addFirst: number
].
iconSelected flash.
firstPart quickFrom: (iconSelected outPosition)
to: ((convDict at: 1) inPosition) for: partName
]
ifFalse: [
tempPlace addFirst: firstPart.
robotNoColl addFirst: number
].
].
]
((convPresent = false) and: [robDict size = 1]) ifTrue: [
(robotNoColl isNil) ifFalse: [
number := robotNoColl removeFirst.
((array at: 1) = number) ifTrue: [
((robDict at: 1) accept: firstPart) ifTrue: [
iconSelected flash.
firstPart quickFrom: (iconSelected outPosition)
to: ((robDict at: 1) inPosition) for: partName
]
ifFalse: [
tempPlace addFirst: firstPart.
robotNoColl addFirst: number
].
].
ifFalse: [
robotNoColl addFirst: number.
firstPart disappearAt: (iconSelected inPosition) for: partName
].
] ifTrue: [firstPart disappearAt: (iconSelected outPosition) for: partName
].
].
((convPresent = false) and: [robDict size > 1]) ifTrue: [ 
  (robotNoColl isNil) ifFalse: [ 
    number := robotNoColl removeFirst.
    1 to: (array size) do: [: i | 
      (((array at: i) = number) and: [robotTypeFound = false]) ifTrue: [ 
        ((robDict at: i) accept: firstPart) ifTrue: [ 
          robotTypeFound := true.
          iconSelected flash.
          firstPart quickFrom: (iconSelected outPosition)
          to: ((robDict at: i) inPosition) for: 
          partName
        ]
      ]
      ifFalse: [ 
        i to: (array size) do: [: j | 
          (((array at: j) = number) and: 
            [robotTypeFound = false])
          ifTrue: [ 
            ((robDict at: j) accept: firstPart) ifTrue: [ 
              firstPart ifTrue: [ 
                robotTypeFound := true.
                iconSelected flash.
                firstPart quickFrom: 
                (iconSelected outPosition)
                to: ((robDict at: j) 
                    inPosition) for: 
                partName
              ]
            ]
          ]
      ]
    ]
    ifFalse: [ 
      tempPlace addFirst: 
      firstPart.
      robotNoColl addFirst: number
    ]
  ].
].

(robotTypeFound = false) ifTrue: [ 
  firstPart disappearAt: (iconSelected outPosition) for: partName.
  robotNoColl addFirst: number
].

] ifTrue: [ 
  firstPart disappearAt: (iconSelected outPosition) for: partName
].

] ifTrue: [ 
  (tempPlace size) < capacity] ifTrue: [ 
    previousObjects do: [: each | each moveIt]
  ]!!
nextObjDict
   `nextObjDict!

nextObjDict: aBag
   nextObjDict := aBag!

nextObjSet
   `nextObjSet!

nextObjSet: aSet
   nextObjSet := aSet!

palletType
   `palletType!

palletType: anInteger
   palletType := anInteger!

readSchedule: box
   | input output x w y |
   "Reads the schedule plan from a file on the hard disk and
copied to a collection"

   input := File pathName: box.
   output := OrderedCollection new.

   [ input atEnd ]
     whileFalse: [
       w := input nextWord.
       (w = 'robot') ifTrue: [
         x := input nextWord.
         y := x asInteger.

         (y isKindOf: Integer) ifTrue: [
           output add: y
         ].
       ].
     ]

   `output!
remove

"Prompts the user for the removal of the Transfer Object.
Upon confirmation, the instance of the object is destroyed and
all events of the object are removed from the eventChain.
Also, the instances of the attached objects with the Transfer
object are destroyed"

i := Menu yesNo: 'Are you sure you want to remove this
object?'.
(i = nil) ifFalse: [
  (i = 0) ifTrue: [nil]
  ifFalse: [
    itsSimulator deSchedule: self.
    itsSimulator pullOutOf: self.
    (nextObjSet notNil) ifTrue: [
      nextObjSet do: [:each |
        each previousObjDict remove: (self label)
        ifAbsent: []
        self detachFrom: each
      ].
    ].
    (previousObjects notNil) ifTrue: [
      previousObjects do: [:each | each detachFrom:
      self
    ].
    (graphicsOn) ifTrue: [ self graphicsOff ].
    itsSimulator := nil
  ].
].
robDict := Dictionary new.
convDict := Dictionary new!

robDict

◆ robDict!

robDict: aDictionary

robDict := aDictionary!

robotPresent

◆ robotPresent!

robotPresent: aBoolean

robotPresent := aBoolean!
showAccept
    iconSelected flash!

tempPlace
    @tempPlace!

tempPlace: aCollection
    tempPlace := aCollection!!
C.3 Geometric Check Class

Object subclass: #Geometric

instanceVariableNames:
  'cc s t u output1 output2 ptMatrix1 edgeMatrix1
faceMatrix1 coordMatrix1 ptMatrix2 edgeMatrix2
faceMatrix2 coordMatrix2 intersectionPt startPt coordRow

classVariableNames: ''
poolDictionaries: '' !

!Geometric class methods !

collisionCheck: box1 with: box2

  self checkIntersection: box1 with: box2!

move: anArray by: aVector

  self translate: anArray by: aVector!

new

  @super new! !

!Geometric methods !

checkIntersection: firstBox with: secondBox

  | k n a v1 v4 b s |

  "Reads the data in arrays firstBox and secondBox and then
calculates the point positions, edge vectors, face normals and
surface vectors. Then all the face normals of the firstBox
and checked for orthoganality with the surface vectors of the
secondBox. The second check involves an intersection check
between the edge vectors of the firstBox with the surface
vectors of the secondBox."

v1 := Array new: 3.
v4 := Array new: 3.
ptMatrix1 := self matrixFromArray: firstBox from: 1 size: 8@4.
ptMatrix2 := self matrixFromArray: secondBox from: 1 size: 8@4.
edgeMatrix1 := self matrixFromArray: firstBox from: 33 size:
  12@6.
edgeMatrix2 := self matrixFromArray: secondBox from: 33 size:
  12@6.
faceMatrix1 := self matrixFromArray: firstBox from: 105 size:
6@8.
faceMatrix2 := self matrixFromArray: secondBox from: 105 size: 6@8.
coordMatrix1 := self matrixFromArray: firstBox from: 153 size: 6@9.
coordMatrix2 := self matrixFromArray: secondBox from: 153 size: 6@9.
k := 1.
1 to: 6 do: [:i | 
 2 to: 4 do: [:j | 
    coordRow := i.
    (v4 at: k put: (faceMatrix1 at: i@j)).
    k := k + 1].
  n := 1.
1 to: 12 do: [:l | 
 4 to: 6 do: [:m | 
    (v1 at: n put: (edgeMatrix2 at: l@m)).
    n := n + 1].
  k := 1.
  n := 1.
(a = 1) ifFalse: [ 
  startPt := edgeMatrix2 at: l@2.
  b := self edgeFaceStatus: v1.
  (b = 0) ifFalse: [ 
    $true].
].
].
false!

edgeFaceStatus: aVector
| z w b s |

"Determines whether an edge normally intersects a face. If the edge does not, the method returns 0. If the first endpoint of the edge intersects, the method returns 1. If the second endpoint intersects, the method returns 2. If the edge interior intersects, the method returns 3."

z := 0.
w := 1.

(b = 0) ifFalse: [ 
  (u > 0) ifTrue: [ 
    (u < 1) ifTrue: [$3].
  ].
  ((self realCompare: u with: z) = 1) ifTrue: [$1].
  ((self realCompare: u with: w) = 1) ifTrue: [$2].
].
false!

edgeLineFaceIntersect: aVector
\[ j \ k \]

"Determines whether an edge intersects a face in a normal fashion. If it does, it returns a value greater than 1. If not, it returns 0."

\[ k := \text{coordMatrix1 rowAt: coordRow}. \]
\[ j := \text{self ptCoordIntersect: k and: aVector.} \]
\[ (j = 0) \text{ifFalse: [g2]}. \]
\[ \text{g0!} \]

\texttt{matrixFromArray: anArray from: start size: aPoint}
\[ | k aMatrix | \]

"Creates a matrix from the array anArray of size x @ y and starting from the kth element"

\[ k := \text{start.} \]
\[ aMatrix := \text{Matrix new: aPoint.} \]
\[ 1 \text{to. (aPoint x) do: [: i] } \]
\[ 1 \text{to. (aPoint y) do: [: j] } \]
\[ aMatrix at: i@j \text{ put: (anArray at: k).} \]
\[ k := k + 1. \]
\[ \text{g}aMatrix! \]

\texttt{orthoCheck: oneVector with: anotherVector}
\[ | z y | \]

"Checks whether the two vectors are orthogonal"

\[ z := 0. \]
\[ y := \text{self scalarProduct: oneVector with: anotherVector.} \]
\[ (\text{self realCompare: y with: z} = 1) \]
\[ \text{ifTrue: [g1]} \]
\[ \text{ifFalse: [g0]}! \]

\texttt{ptCoordIntersect: anArray and: aVector}
\[ | z a | \]

"Accepts an array and a vector. The array contains a coordinate system and a point. It subsequently determines if the vector emanating from the point intersects the coordinate system. If there is no intersection, the method returns 0. If the point lies outside the coordinate system, the method returns 2. Finally, if the point lies on the coordinate system, the method returns 3."

\[ z := 0. \]
\[ a := \text{self ptLineCoordIntersect: anArray and: aVector.} \]
\[ (a = 0) \text{ifFalse: [} \]
\[ (\text{self realCompare: u with: z} = 1) \text{ifTrue: [g3]} \]
\[ \text{ifFalse: [} \]
\[ (u > 0) \text{ifTrue: [g2]} \]
ifFalse: [  
(u < 0) ifTrue: [c1].
].

] 
ifTrue: [c0]!

ptLineCoordIntersect: anArray and: aVector
| z a w ss |

"Determines if a vector emanating from a point intersects the
coordinate system. If there is no intersection, the method
returns 0. If there is, it returns 1."

z := 0.
w := 1.
(a = 0) ifFalse: [  
(s between: 0 and: 1) ifTrue: [  
(t between: 0 and: 1) ifTrue: [  
c1].

] ]

] .

c0!

ptLinePlaneIntersect: anArray and: aVector
| b k d n p i z c s coll1 coll2 coll3 coll4 coll5 coll6 ss |

"Accepts the values of a plane, a point, and a vector. It
subsequently determines if the vector emanating from the point
intersects the plane. If there is no intersection, the method
returns 0. If there is, it returns 1."

b := Matrix new: 3@1.
k := Matrix new: 3@1.
d := Matrix new: 3@3.
intersectionPt := Array new: 3.
i := 1.
n := 4.
z := 0.

2 to: 4 do: [: j |  
k at: i at1 put: ((ptMatrix2 at: startPt@j) - (anArray at:  
i)).
b at: i at1 put: (-1*aVector at: i)).
i := i + 1].
1 to: 2 do: [: m |
1 to: 3 do: [: l |  
d at: l at m put: (anArray at: n).
n := n + 1]  
].
1 to: 3 do: [: p |  
d at: p at3 put: (b at: p@1)].

((self realCompare: (aVector at: 1) with: z) = 0) ifTrue: [
c := self stack3SimulSolve: d and: k
ifFalse: [
   (((aVector at: 2) = 0) not) ifTrue: [
      coll1 := d rowAt: 1.
      coll2 := d rowAt: 2.
      coll3 := k at: 1@1.
      coll4 := k at: 2@1.
      d replaceFrom: 1@1 to: 1@3 with: coll2.
      d replaceFrom: 2@1 to: 2@3 with: coll1.
      k at: 1@1 put: coll4.
      k at: 2@1 put: coll3.
      c := self stack3SimulSolve: d and: k
   ]
   ifFalse: [
      (self realCompare: (aVector at: 3) with: z) = 0
      ifTrue: [
         coll1 := d rowAt: 1.
         coll2 := d rowAt: 2.
         coll3 := d rowAt: 3.
         coll4 := k at: 1@1.
         coll5 := k at: 2@1.
         coll6 := k at: 3@1.
         d replaceFrom: 1@1 to: 1@3 with: coll5.
         d replaceFrom: 2@1 to: 2@3 with: coll1.
         d replaceFrom: 3@1 to: 3@3 with: coll2.
         k at: 1@1 put: coll6.
         k at: 2@1 put: coll3.
         k at: 3@1 put: coll4.
         c := self stack3SimulSolve: d and: k].
   ].
   (c = 1) ifTrue: [
      intersectionPt at: 1 put: (u * (aVector at: 1) + (ptMatrix2 at: startPt@2)).
      intersectionPt at: 2 put: (u * (aVector at: 2) + (ptMatrix2 at: startPt@3)).
      intersectionPt at: 3 put: (u * (aVector at: 3) + (ptMatrix2 at: startPt@4)).
      c1
      ifFalse: [0]!
   ]
]
readAsChar: box
| string input output anArray anArray1 x|
"Reads data from a file and stores it into an ordered collection"

input := File pathName: box.
output := OrderedCollection new.

[input atEnd]
whileFalse: [
   output add: input nextWord].
anArray1 := Array new: ((output size) - 1).
anArray := output asArray.
x := ((anArray size) - 1) .
1 to: x do: [:i |
(anArray1 at: i put: (anArray at: i))
].

ganArray1!

readDataOf: box
| s string input output anArray anArray1 x|

input := File pathName: box.
output := OrderedCollection new.

[input atEnd]
  whileFalse:
    output add: input nextWord.
  anArray1 := Array new: ((output size) - 1)
anArray := output asArray.
x := ((anArray size) - 1).
1 to: x do: [: i |
  (anArray1 at: i put: ((anArray at: i) asInteger))
].
s := FreeDrawing l.
ganArray1!

realCompare: firstNumber with: secondNumber

"Compares the values of the firstNumber and the secondNumber
to determine if they are equal"

((firstNumber - secondNumber) < (1/1000))
  ifTrue: [
    ((firstNumber - secondNumber) > (-1/1000))
      ifTrue: [g1]
      ifFalse: [g0].
    ]!

scalarProduct: oneVector with: anotherVector

"Scalar multiplication of two vectors"

g(((oneVector at: 1)*(anotherVector at: 1)) + ((oneVector at: 2)*(anotherVector at: 2)) + ((oneVector at: 3)*(anotherVector at: 3)))

stack3SimulSolve. fM and: sM
| a b c d e f g x1 x2 x3 z ss |

  a := ((fM at: 2@3)/(fM at: 1@3)).
b := ((fM at: 3@3)/(fM at: 1@3)).
c := ((fM at: 2@1) - (a*(fM at: 1@1))).
d := ((fM at: 2@2) - (a*(fM at: 1@2))).
z := 0.
((self realCompare: c with: z) = 1) iTrue: [
  ((self realCompare: d with: z) = 1) iTrue: [
    u := (sM at: 1@1)/(fM at: 1@3).
    s := 0.
    t := 0].
  ]
  ifFalse: [
    ((self realCompare: c with: z) = 0) iTrue: [
      ((self realCompare: d with: z) = 1) iTrue: [
        e := d/c.
        f := (((fM at: 3@2) - (e*(fM at: 3@1)) - (b*(fM at: 1@1)))/((fM at: 1@2))).
        iTrue: [
          t := 0.
          s := ((sM at: 2@1) - (a*(sM at: 1@1)))/c.
          u := (((sM at: 1@1) - ((fM at: 1@1)*s))/((fM at: 1@3))]
        ]
        ifFalse: [
          g := ((sM at: 2@1) - (a*(sM at: 1@1)))/c.
          t := ((sM at: 3@1) - (g*(b*(fM at: 1@1) + (fM at: 3@1)) + (b*(sM at: 1@1)))/f.
          s := g - (e*t).
          u := ((sM at: 1@1) - ((fM at: 1@1)*s) - (((fM at: 1@2)*t))/((fM at: 1@3))]
      ]
      ifFalse: [
        (((self realCompare: c with: z) = 1) and: (self realCompare: d with: z) = 0)
        iTrue: [
          e := c/d.
          f := (((fM at: 3@1) - (e*(fM at: 3@2)) - (b*(fM at: 1@2)))/((fM at: 1@1))).
          iTrue: [
            s := 0.
            t := ((sM at: 2@1) - (a*(sM at: 1@1)))/d.
            u := ((sM at: 1@1) - ((fM at: 1@2)*t))/((fM at: 1@3))]
          ]
          ifFalse: [
            g := ((sM at: 2@1) - (a*(sM at: 1@1)))/d.
            s := ((sM at: 3@1) - (g*(b*(fM at: 1@2) + (fM at: 3@2)) + (b*(sM at: 1@1)))/f.
            t := g - (e*s).
            u := ((sM at: 1@1) - ((fM at: 1@1)*s) - ((fM at: 1@2)*t))/((fM at: 1@3)]
          ]
        ]
      ]
    ]
  ]
ifFalse:
  e := d/c.
  f := (((fM at: 3@2) - (e*((fM at: 3@1) -
    (b*(fM at: 1@1)))) -
    (b*(fM at: 1@2))))).
  (self realCompare: f with: z) = 1) ifTrue:
    [ 
      e := c/d.
      f := (((fM at: 3@1) - (e*((fM at: 3@2)
        - (b*(fM at: 1@2)))) -
        (b*(fM at: 1@1))).
      (self realCompare: f with: z) = 1)
    ifTrue:
      [ s := 0.
        t := ((sM at: 2@1) - (a*(sM at:
          1@1)))/d.
        u := ((sM at: 1@1) - ((fM at:
          1@2)*t))/(fM at: 1@3) ]
    ifFalse:
      [ g := ((sM at: 2@1) - (a*(sM at:
          1@1)))/d.
        s := ((sM at: 3@1) - (g*(b*(fM
          at: 1@2) + (fM at: 3@2)))
          + (b*(sM at: 1@1)))/f.
        t := g - (e*s).
        u := ((sM at: 1@1) - ((fM at:
          1@1)*s) -
          ((fM at: 1@2)*t))/(fM at: 1@3) ]
  ifFalse:
    [ g := ((sM at: 2@1) - (a*(sM at:
        1@1)))/c.
      t := ((sM at: 3@1) - (g*(b*(fM
        at: 1@2) + (fM at: 3@2))) +
        (b*(sM at: 1@1)))/f.
      s := g - (e*t).
      u := ((sM at: 1@1) - ((fM at: 1@1)*s)
        - ((fM at: 1@2)*t))/(fM at: 1@3) ]
  ].

x1 := (((fM at: 1@1)*s) + ((fM at: 1@2)*t) + ((fM at: 1@3)*u)).

x2 := (((fM at: 2@1)*s) + ((fM at: 2@2)*t) + ((fM at: 2@3)*u)).

x3 := (((fM at: 3@1)*s) + ((fM at: 3@2)*t) + ((fM at: 3@3)*u)).

((self realCompare: x1 with: (sM at: 1@1)) = 1)
ifTrue: [
    (self realCompare: x2 with: (sM at: 2@1)) = 1
    ifTrue: [
        (self realCompare: x3 with: (sM at: 3@1)) = 1;
          
        ifTrue: [ธร1]
        ifFalse: [ธร0].
    ].
].
ธร0!

translate: anArray by: aVector
| i j m n p s a b d v |

"Moves the array anArray by a vector aVector"

i := -2.
j := 0.
to: 8 do: [i |  
i := i + 4.
j := j + 4.
m := 1.
i to: j do: [k |  
anArray at: k put: ((anArray at: k) + (aVector at: m)).
m := m + 1].
].

n := 153.
p := 155.
to: 3 do: [q |  
s := 2.
n to: p do: [r |  
anArray at: r put: (anArray at: s).
s := s + 1].
n := n + 9.
].

a := 180.
b := 182.
to: 3 do: [c |  
d := 22.
a to: b do: [v |  
anArray at: v put: (anArray at: d).
d := d + 1].
a := a + 9.
b := b + 9.
].
ธรanArray! !