Politecnico di Milano
Formal Methods for Concurrent
and
Real-time Systems
Homework project

Collaborative Robotics Modeling

Aldeghi Gabriele
Sacco Alessio
Mantovani Mirko
Sonzogni Stefano

June 8, 2018

Contents

1 Formalization of the problem

1.1 Problem description

The TRIO+ model presented in this document aims to formalize the interaction between an operator and a KUKA robot.

The goal of the robot is to move unfinished pieces from the bin area to the tombstone. After the piece has been worked, the robot will move it to the conveyor belt. The operator is assigned to a supervision role and he will interact physically with the end effector of the machine. The model has to guarantee the safety of the operator as well as the utility constraints.

1.2 Definitions and Acronyms of Components

- **R**: The whole KUKA mobile robot
- **EE**: The End-Effector of the robot's arm
- **O**: The Operator that works in the same environment of the Robot and interacts with it
- **L**: The entire Layout in which Robot and Operator work
- **BA**: The Bin Area (top-right)
- **WP**: The single WorkPiece which is transported by the robot
- **HDI**: The Human Device Interface used by the Operator to control the Robot

1.3 Constants

- **N**: The capacity of the local bin of the Robot

1.4 World discretization

1.4.1 Human body parts

- **Head area**: Highly sensitive areas
- **Arms area**: Very delicate areas
- **Body area**: Delicate areas

1.4.2 Robot parts

- **Arm**: Consisting of 2 segments and 3 links, the last of which connects the farther segment to the EE
- **End Effector**: The robotic hand of R, capable of holding a WP and releasing it.
- **Cart**: The cart is the main part of the robot, it is what can move across areas in the layout.
- **Local Bin**: The local bin of R is located on top of the cart and is able to contain WPs.

1.4.3 Robot speed

- **None**: Null speed, the robot cannot move
- **Low**: Low speed, the robot can move every other time instant, which means it will move to an adjacent area at most every 2 time instants
- **Normal**: Normal speed, the robot can move to an adjacent area at each time instant.

1.4.4 Layout areas

We discretized the layout in such a way that the most important and critical areas, in which Human-Robot interaction are very likely to happen, have a fine-grained grid, whereas the other areas, in which the Operator should not be present to assist the Robot, are modeled as bigger blocks in order not to introduce unnecessary complexity in the model and to avoid state space explosion.

The most critical areas are the ones with indices from 1 to 12, particular attention must be paid to L_0 too, since it is where the EE and O could be working together.

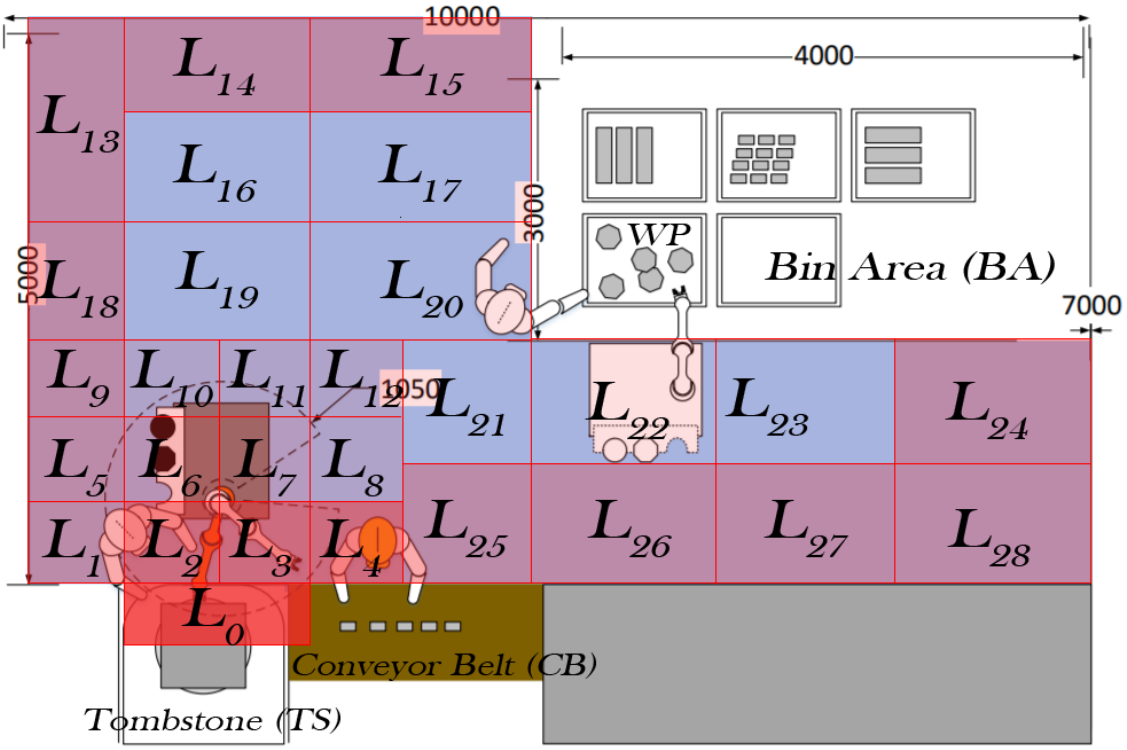


Figure 1: Subdivision of the layout. The highligthed areas are the dangerous ones.

1.5 Assumptions and modeling

Robot The robot is discretized in two parts, the cart and the arm. The cart is such that it can only occupy one area at the time: this is a strong assumption, however it is vital because it reduces the complexity of our model. The arm is modeled by considering two joints and the end effector. The first joint is assumed to be the connection between the cart and the arm, therefore its position is at all times the same as the cart position. The second joint allows broader movement to the arm and it links the first joint to the end effector. The whole arm can reach at most a distance of two adjacent areas. The response of the actuators to a change in the control variables is assumed to be short enough to be modeled as instantaneous.

Human The human is discretized in body, head and arms. Each one of the parts can only occupy one area. The movement can only be observed, but not controlled, by the robot.

Bin Area We operated under the assumption that the Bin Area is always full, or better, it is never empty in any subarea of its space. With this assumption the KUKA robot will only have to stretch out from an adjacent area and pick as many WP as it wants without having to move to another adjacent area because there are no WP left where it is.

Local Bin The local bin of R is where it stores the WPs in order to transport them from one place to the other. For the sake of simplicity we made as an assumption that the local bin can contain at most 3 pieces, so we could write formulae in a simple way, of course this can be generalized to N pieces in a trivial way.

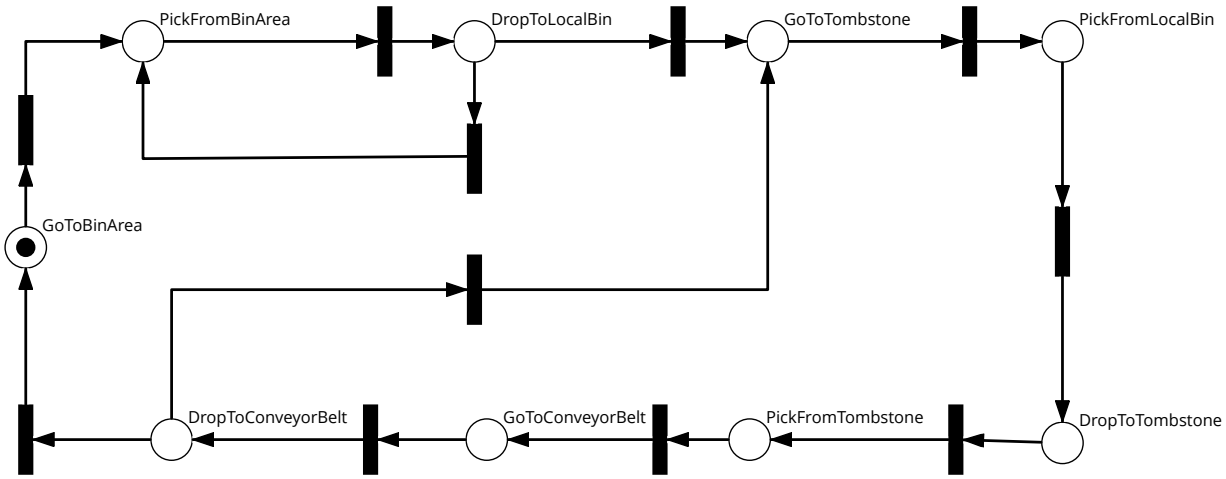
1.6 KUKA workflow

The workflow of KUKA can be described by a FSA, we divided the tasks that R has to perform in 9 main elementary actions, whose formalization can be found in the **RobotController** TRIO+ class. Those atomic actions whose name is self-explaining are:

1. **PickFromBinArea**
2. **DropToLocalBin**
3. **GoToTombstone**
4. **PickFromLocalBin**
5. **DropToTombstone**
6. **PickFromTombstone**
7. **GoToConveyorBelt**
8. **DropToConveyorBelt**
9. **GoToBinArea**

1.7 Petri Net of KUKA workflow

The following Petri net that represents the correct flow of actions that the robot has to perform.



2 Archi-TRIO model

2.1 Class overview

We organized the model in 12 TRIO+ classes, here we provide an overview of each one and a UML-like diagram to better visualize the connections among them.

2.1.1 Grid

The layout is discretized in this class, here we defined the adjacency predicate and the enumeration of subareas of L, as well as other specific predicates for special areas contained in L.

2.1.2 Sensors

The main superclass of sensors is **PositionSensor**, it uses as module the **Grid** and has 2 predicates: Position(layout area) and moved(), the subclasses which inherits them are the following ones:

- **OperatorHeadPositionSensor**: The sensor for the head of O
- **OperatorBodyPositionSensor**: The sensor for the body of O
- **OperatorArmPositionSensor**: The sensor for the arm of O
- **RobotArmPositionSensor**: The sensor for the arm of O, in which the linkings and constraints are made consistent
- **RobotCartPositionSensor**: The sensor for the Cart position in the layout

Then 2 main wrappers are created around the sensors, in which constraints among the various specific sensors are defined:

- **OperatorPositionSensor**: The wrapper sensor for O which imports as modules OperatorHeadPositionSensor, OperatorBodyPositionSensor, OperatorArmPositionSensor and the Grid
- **RobotPositionSensor**: The wrapper sensor for R which imports as modules RobotArmPositionSensor, RobotCartPositionSensor and the Grid

2.1.3 RobotStatus

In this class we have the main predicates about the state of the robot and of its local bin (isFull, isEmpty), here we also define the targetCartSpeed and targetEESpeed which are the control variables, and currentCartSpeed, currentEESpeed which are read from sensors and defined as a part of the state of the robot itself.

2.1.4 RobotController

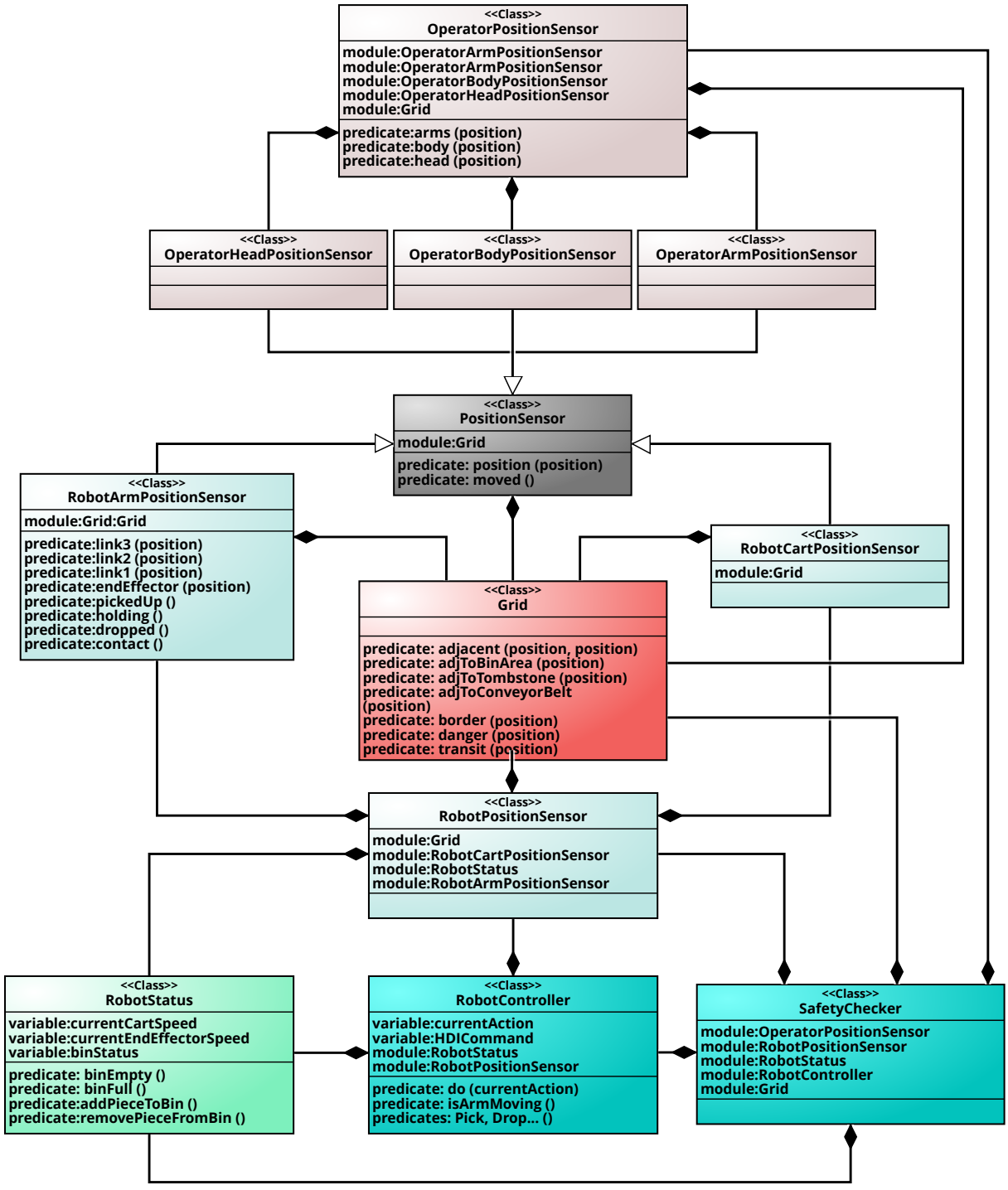
This is one of the most important classes, it handles all the actions performed by R, it specifies preconditions, axioms with constraints on what should be true while an action is being performed, and the actual sequence of events that the action is composed by.

2.1.5 SafetyChecker

Here we define all the constraints necessary for the safety of O in the presence of a working KUKA R around the layout, all the actions performed by R that could cause harm to O are avoided or limited as much as possible.

2.2 UML diagram

This diagram shows the connections and hierarchies among classes.



3 TRIO+ specification

3.1 Definition of Grid class

```
-- This class is the one that provides the predicates for adjacency
-- and the different typologies of position that we can have.
-- We have three typologies:
--     - Border:  They are all the areas near the walls of the room or near the
--                working positions
--     - Danger:  They are all the areas in which the operator can work.
--                The robot need to be extra careful in these areas, as
--                through its arm it can severely harm the operator
--     - Transit: They are all the remaining areas that are not Border or
--                Danger areas.
--
-- All the trio classes that needs to control the adjacency between
-- different position will import this module.

class Grid
    visible adjacent;

    temporal domain integer;

    TI items
        predicates
            adjacent(position, position),

            adjToBinArea({L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                          L10, L11, L12, L13, L14, L15, L16, L17,
                          L18, L19, L20, L21, L22, L23, L24, L25,
                          L26, L27, L28, LBA, LCB}),

            adjToTombstone({L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                            L10, L11, L12, L13, L14, L15, L16, L17,
                            L18, L19, L20, L21, L22, L23, L24, L25,
                            L26, L27, L28, LBA, LCB}),

            adjToConveyorBelt({L0, L1, L2, L3, L4, L5, L6, L7, L8,
                               L9, L10, L11, L12, L13, L14, L15,
                               L16, L17, L18, L19, L20, L21, L22,
                               L23, L24, L25, L26, L27, L28, LBA,
                               LCB}),

            border({L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                    L10, L11, L12, L13, L14, L15, L16, L17,
                    L18, L19, L20, L21, L22, L23, L24, L25,
                    L26, L27, L28, LBA, LCB}),


            danger({L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                    L10, L11, L12, L13, L14, L15, L16, L17,
                    L18, L19, L20, L21, L22, L23, L24, L25,
                    L26, L27, L28, LBA, LCB}),
```

```
50
51              transit({L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
52                       L10, L11, L12, L13, L14, L15, L16, L17,
53                       L18, L19, L20, L21, L22, L23, L24, L25,
54                       L26, L27, L28, LBA, LCB});
55
56      axioms
57
58          -- This is the formula that appears into the document's appendix.
59          -- Specifies which areas are adjacent one with another
60          adjacency:
61              ∀x, y(adjacent(x, y) ⇒ ...);
62
63          -- Definition of all the areas that are danger and which are not.
64          dangerArea:
65              danger(L0) ∧ danger(L2) ∧ danger(L3) ∧ danger(L4) ∧ danger(L6) ∧ danger(L7)∧
66              danger(L8) ∧ danger(L10) ∧ danger(L11) ∧ danger(L12) ∧ danger(L15) ∧ danger(L17)∧
67              danger(L20) ∧ danger(L22) ∧ danger(L23) ∧ danger(L24)∧
68             ¬danger(L1) ∧ ¬danger(L5) ∧ ¬danger(L9) ∧ ¬danger(L13) ∧ ¬danger(L14)∧
69             ¬danger(L16) ∧ ¬danger(L18) ∧ ¬danger(L19) ∧ ¬danger(L21) ∧ ¬danger(L25)∧
70             ¬danger(L26) ∧ ¬danger(L27) ∧ ¬danger(L28);
71
72          -- Definition of all the border areas.
73          borderArea:
74              border(L1) ∧ border(L2) ∧ border(L3) ∧ border(L4) ∧ border(L25) ∧ border(L27)∧
75              border(L28) ∧ border(L24) ∧ border(L23) ∧ border(L22) ∧ border(L20) ∧ border(L17)∧
76              border(L15) ∧ border(L14) ∧ border(L13) ∧ border(L18) ∧ border(L9) ∧ border(L5)∧
77             ¬border(L0) ∧ ¬border(L6) ∧ ¬border(L7) ∧ ¬border(L8) ∧ ¬border(L10)∧
78             ¬border(L11) ∧ ¬border(L12) ∧ ¬border(L21) ∧ ¬border(L16) ∧ ¬border(L19);
79
80          -- Definition of all the transit areas.
81          transitArea:
82              ∀x(transit(x) ⟺ ¬border(x) ∨ ¬danger(x));
83
84          binArea:
85              ∀x(adjToBinArea(x) ⟺ adjacent(x, LBA);
86
87          tombstone:
88              ∀x(adjToTombstone(x) ⟺ adjacent(x, L0);
89
90          conveyorBelt:
91              ∀x(adjToConveyorBelt(x) ⟺ adjacent(x, LCB);
92  end Grid.
```

3.2 Definition of the PositionSensor class

```
1
2  -- This is the basic sensor of our system. Allows to define a predicate position,
3  -- that will be used by both the robot's and operator's position sensors.
4
5  class PositionSensor
6      visible position, adjacent;
7
8      temporal domain integer;
9
10     -- The domain of the predicate position lies inside all the possible areas
11     -- of interest.
12     -- The predicate moved allows us to manage the robot's movement. This is due
13     -- to the fact that the robot can move at three speed intensity. Therefore,
14     -- for a correct modelization of this movement, we needed a predicate that
15     -- tell us whether the robot has moved in this time instant.
16     TD items
17         predicates  position({L0, L1, L2, L3, L4, L5, L6, L7,
18                               L8, L9, L10, L11, L12, L13, L14,
19                               L15, L16, L17, L18, L19, L20, L21,
20                               L22, L23, L24, L25, L26, L27, L28,
21                               LBA, LCB});
22                     moved();
23
24     modules: Grid: Grid;
25
26     connection: {(Grid.adjacent, adjacent)};
27     axioms
28         -- If the predicate moved is true, this means that the current position
29         -- must be different from the one assumed one time instant in the past.
30         moved:
31             moved() ⟺ ∃x Past(position(x), 1) ∧ ¬position(x);
32
33 end PositionSensor.
```

3.3 Definition of the operator's sensor classes

```
-- Simple class that specify the predicates for the position of
-- the operator's arm. The arm can be in only one position in
-- the map at each time instant.
-- One thing to notice is that in our model the operator will have
-- two arms.

class OperatorArmPositionSensor
    inherits PositionSensor

    visible position;

    temporal domain integer;

    axioms
        -- the arm always exists
        existsArm: ∃x(position(x));

        -- the arm is only in one area
        uniqueArm: ∀x(position(x) ⟹ ∄y(position(y) ∧ x ≠ y));

end OperatorArmPositionSensor.
```

```
-- Simple class that specify the predicate position for the
-- operator's body. The body can be in only one position at
-- each time instant.

class OperatorBodyPositionSensor
    inherits PositionSensor

    visible position;

    temporal domain integer;

    axioms
        -- the body always exists
        existsBody: ∃x(position(x));

        -- the body is only in one area
        uniqueBody: ∀x(position(x) ⟹ ∄y(position(y) ∧ x ≠ y));

end OperatorBodyPositionSensor.
```

```
-- This class defines the position that the head of the
-- operator assumes at each time instant. This position
-- will be crucial for the definition of the safety
-- properties.
-- We assume also that the head of hhe operator can be in
-- only one position at each time instant.

class OperatorHeadPositionSensor
    inherits PositionSensor

    visible position;

    temporal domain integer;

    axioms
        -- the head always exists
        existsHead: ∃x(position(x));

        -- the head is only in one area
        uniqueHead: ∀x(position(x) ⟹ ∄y(position(y) ∧ x ≠ y));

end OperatorHeadPositionSensor.
```

```
class OperatorPositionSensor
    inherits PositionSensor

    visible arms, body, head;

    temporal domain integer;

    TD items
        predicates
            arms({L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                  L10, L11, L12, L13, L14, L15, L16, L17, L18, L19,
                  L20, L21, L22, L23, L24, L25, L26, L27, L28 }),
            body({L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                  L10, L11, L12, L13, L14, L15, L16, L17, L18, L19,
                  L20, L21, L22, L23, L24, L25, L26, L27, L28 }),
            head({L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                  L10, L11, L12, L13, L14, L15, L16, L17, L18, L19,
                  L20, L21, L22, L23, L24, L25, L26, L27, L28 });


    modules LeftArm: OperatorArmPositionSensor,
             RightArm: OperatorArmPositionSensor,
             Body: OperatorBodyPositionSensor,
             Head: OperatorHeadPositionSensor,
             Grid: Grid;

    axioms
        -- connect the predicates between the modules
        arms:∀x(arms(x) ⟺ (LeftArm.position(x) ∨ RightArm.position(x)));
        body:∀x(body(x) ⟺ Body.position(x));
```

```
31          head : ∀x(head(x) ⟺ Head.position(x));
32
33          -- head is on the body or in a close by cell
34          headOnTheBody : ∀x(head(x) ⇒ body(x) ∨ ∃y(body(y) ∧ Grid.adjacent(x, y)));
35
36          -- arms are on the body or in a close by cell
37          armsOnTheBody : ∀x(arms(x) ⇒ body(x) ∨ ∃y(body(y) ∧ Grid.adjacent(x, y)));
38
39          -- the operator can move one area in each time instant
40          movement : ∀x(body(x) ⇒ ∃y(Future(body(y), 1) ∧ (x == y ∨ Grid.adjacent(x, y))));
41
42  end OperatorPositionSensor.
```

3.4 Definition of the robot's sensor classes

```
-- This class specify the sensors for the robot's arm. We decide
-- to model the arm as the real one present on the KUKA unit.
-- The arm is composed of 3 links and one end effector.
-- There are three predicates that will be use to determine
-- when the robot is switching between an action an another,
-- and they are pickedUp(), holding() and dropped().

class RobotArmPositionSensor
    inherits PositionSensor

    temporal domain integer;

    visible position, link3, link2, endEffector, pickedUp, holding, dropped;

    modules Grid: Grid;

    TD items
        predicates
            -- Link3 is the first segment of the arm from the body of the robot
            link3({ L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                    L10, L11, L12, L13, L14, L15, L16, L17,
                    L18, L19, L20, L21, L22, L23, L24, L25,
                    L26, L27, L28, LBA, LCB}),

            -- Link2 is the second segment of the arm, linked to link3 and link1
            link2({ L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                    L10, L11, L12, L13, L14, L15, L16, L17,
                    L18, L19, L20, L21, L22, L23, L24, L25,
                    L26, L27, L28, LBA, LCB}),

            -- Link1 is the third segment of the arm, which is the end effector
            link1({ L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                    L10, L11, L12, L13, L14, L15, L16, L17,
                    L18, L19, L20, L21, L22, L23, L24, L25,
                    L26, L27, L28, LBA, LCB}),

            -- same as link1, used for clarity
            endEffector({ L0, L1, L2, L3, L4, L5, L6, L7, L8, L9,
                          L10, L11, L12, L13, L14, L15, L16, L17,
                          L18, L19, L20, L21, L22, L23, L24, L25,
                          L26, L27, L28, LBA, LCB}),


            -- Signal if the pick action has been completed successfully
            pickedUp(),


            -- Signal to model the fact that the robot is holding a piece
            holding(),

            -- Signal to model the fact that the robot has dropped the piece it
```

```
52              -- was holding
53              dropped();
54
55              -- Signal to model the fact that the robot's arm is being touched by
56              -- the operator
57              contact();
58
59      axioms:
60
61          -- The position is all the cells that are occupied by the arm
62          armPosition:
63              ∀x(position(x) ⟺ link2(x) ∨ link1(x));
64
65          -- link 2 is connected to link3
66          connection32:
67              ∀x∀y(link2(x) ⟹
68                      link3(x) ∨ (x ≠ y ∧ link3(y) ∧ Grid.adjacent(x, y)));
69
70          -- link1 is connected to link2
71          connection21:
72              ∀x∀y(link1(x) ⟹
73                      link2(x) ∨ (x ≠ y ∧ link2(y) ∧ Grid.adjacent(x, y)));
74
75          -- end effector is link1
76          endEffectorOnLink1:
77              ∀x(endEffector(x) ⟺ link1(x));
78
79          -- the link1 always exists
80          existsLink1: ∃x(link1(x));
81
82          -- the link1 is only in one area
83          uniqueLink1: ∀x(link1(x) ⟹ ∄y(link1(y) ∧ x ≠ y));
84
85          -- the link2 always exists
86          existsLink2: ∃x(link2(x));
87
88          -- the link2 is only in one area
89          uniqueLink2: ∀x(link2(x) ⟹ ∄y(link2(y) ∧ x ≠ y));
90
91          -- the link3 always exists
92          existsLink3: ∃x(link3(x));
93
94          -- the link3 is only in one area
95          uniqueLink3: ∀x(link3(x) ⟹ ∄y(link3(y) ∧ x ≠ y));
96
97          -- the arm cannot teleport
98          armCannotTeleport:
99              ∀x(endEffector(x) ⟹
100                 (Future(endEffector(x) ∨ (∃y(endEffector(y) ∧ Grid.adjacent(x, y))), 1)));
101
102         -- Only one of pickedUp(), holding() and dropped() can be true in
103         -- each instant
104         onlyOne:
```

```
            (pickedUp() ⟹
                ¬holding() ∧ ¬dropped())∧
            (holding() ⟹
                ¬pickedUp() ∧ ¬dropped())∧
            (dropped() ⟹
                ¬pickedUp() ∧ ¬holding());

        -- holding() holds between pickedUp() and dropped()
        sequence:
            Becomes(¬pickedUp()) ∧ Until(holding(), dropped());

end RobotArmPositionSensor.
```

```
-- The robot has a position, and from that position it can be present only in
-- Grid.adjacent areas from the position. The Position predicate represent
-- these adjacent position, while the position predicate specify the current
-- central position of the robot.
-- Another condition we need to supply is the fact that the robot cannot
-- "teleport". This condition is guaranteed by the fact that any new position
-- that the robot assumes must be Grid.adjacent to at least one position of
-- the robot in the past.

class RobotCartPositionSensor;
    inherits PositionSensor

    visible position, moved;

    temporal domain integer;

    modules Grid: Grid;

    axioms

        -- the cart always exists
        existsCart: ∃x(position(x));

        -- the cart is only in one area
        uniqueCart: ∀x(position(x) ⟹ ∄y(position(y) ∧ x ≠ y));

        -- An area occupied by the robot must be Grid.adjacent with an area
        -- occupied by the robot one time instant in the past
        doNotTeleport:
            moved() ⟹ ∃x∃y(x ≠ y ∧ Grid.adjacent(x, y) ∧ position(x) ∧ Past(position(y), 1));


end RobotCartPositionSensor.
```

```
-- This is the class that collects all the sensor of the
-- robot.
-- In this class we defined a predicate for the robot's velocities,
-- both for the arm and the cart. We defined axioms to check the
-- robot's movement. As example, if the robot is moving at low speed,
-- it can move in a adjacent area in two time unit. This modelization
-- is permitted through the use of a support predicate moved().
-- We also define a predicate contact to model the fact that the robot's
-- arm can be touched by the operator to guide its working process.
-- Some specific axioms has been defined to model the situation
-- explained before.

class RobotPositionSensor
    temporal domain integer

    visible Cart, Arm;

    modules Cart: RobotCartPositionSensor,
             Arm: RobotArmPositionSensor,
             Status: RobotStatus,
             Grid: Grid;

    axioms
        -- link3 is in the same position as the cart
        link3OnCart:
            ∀x(Arm.link3(x) ⟺ Body.position(x));

        -- the cart can move only when the speed is not equal to None.
        -- if the speed is normal, then the robot can move at every time instant
        -- if the speed is low, then the robot can move only every other time instant
        cartMovement:
            Cart.moved() ⟺ Past(Status.currentCartSpeed == Normal, 1)∨
                Past(Status.currentCartSpeed == Low ∧ ¬Cart.moved(), 1)

        --The three reasons why the arm can move are mutually exclusive
        cartSpeedNotNone:
            RobotStatus.currentCartSpeed ≠ None ⟹
                RobotStatus.currentEndEffectorSpeed == None ∧ ¬Arm.contact()

        armSpeedNotNone:
            RobotStatus.currentEndEffectorSpeed ≠ None⇒
                RobotStatus.currentCartSpeed == None ∧ ¬Arm.contact()

        noSpeedWhileTouching:
            Arm.contact()⇒
                RobotStatus.currentCartSpeed == None ∧ RobotStatus.currentEndEffectorSpeed == None


        -- if there is no contact between the arm and the operator, then the arm can
        -- move only if the speed is not None
        armMovement:
            Arm.moved() ⟺
                (Cart.moved()∨
```

```
54                  (Past(RobotStatus.currentEndEffectorSpeed == Normal, 1)∨
55                      Past(RobotStatus.currentEndEffectorSpeed == Low ∧ ¬Arm.moved(), 1))∨
56                  Past(Arm.contact(), 1) ∧ Arm.contact()∧
57                      (∃x∃z(x ≠ z ∧ Past(Operator.arm(x), 1) ∧ Operator.arm(z) ∧ (
58                          (Past(Arm.link2(x), 1) ∧ Arm.link2(z))∨
59                          (Past(Arm.endEffector(x), 1) ∧ Arm.endEffector(z))))))
60
61          -- if the cart is moving, then the arm must stay on top of the cart,
62          -- both the end effector and the link
63          armOnTopMovingCart:
64              Cart.moved() ⟹
65                  Past(∀x(Cart.position(x) ⟺ Arm.position(x)), 1) ∧ ∀x(Cart.position(x) ⟺ Arm.position(x))
66
67          --------------------
68          -- CONTACT AXIOMS --
69          --------------------
70
71          -- the Arm.contact() predicate can be true only when the arms of the
72          -- operator are in the same area as the arm of the robot
73          armContact:
74              ∀x(Arm.position(x) ∧ Arm.contact() ⟹
75                  Operator.arms(x));
76
77  end RobotPositionSensor.
```

3.5 Definition of the state of the robot

```
-- In this class we define all the predicates relative to the robot's bin
-- and its velocity.
-- We defined an ordering on how the bin is filled and empty. We defined as
-- capacity of our bin only three pieces.
class RobotStatus

    temporal domain integer;

    visible targetCartSpeed, targetEndEffectorSpeed,
             binEmpty, binFull, addPieceToBin, removePieceFromBin,
             currentCartSpeed, currentEndEffectorSpeed;

    TD items:
        predicates
            binEmpty(),
            binFull(),
            addPieceToBin(),
            removePieceFromBin();

        vars
            currentCartSpeed({None, Low, Normal}),
            currentEndEffectorSpeed({None, Low, Normal}),
            binStatus({Empty, 1, 2, Full});

    axioms
        -- bin empty
        binIsEmpty:
            binEmpty() ⟺ binStatus == Empty;

        -- bin full
        binIsFull:
            binFull() ⟺ binStatus == Full;

        -- ordering of bin status
        ordering1:
            binStatus == Empty ∧ addPieceToBin() ⟹ ∃x(Becomes(binStatus == 1, x));
        ordering2:
            binStatus == 1 ∧ addPieceToBin() ⟹ ∃x(Becomes(binStatus == 2, x));
        ordering3:
            binStatus == 2 ∧ addPieceToBin() ⟹ ∃x(Becomes(binStatus == Full, x));
        ordering4:
            binStatus == Full ∧ removePieceFromBin() ⟹ ∃x(Becomes(binStatus == 2, x));
        ordering5:
            binStatus == 2 ∧ removePieceFromBin() ⟹ ∃x(Becomes(binStatus == 1, x));


        ordering6:
            binStatus == 1 ∧ removePieceFromBin() ⟹ ∃x(Becomes(binStatus == Empty, x));

        -- a piece can be added to the local bin only if there is
        -- enough capacity
```

```
52          stillNotFull:
53              addPieceToBin() ⟹ ¬binStatus == Full;
54
55          -- a piece can be removed from the local bin only if there is
56          -- at least one in the bin
57          stillNotEmpty:
58              removePieceFromBin() ⟹ ¬binStatus == Empty;
59
60          -- addPieceToBin() and removePieceFromBin() are instantaneous events
61          singleInstantEvents1:
62              addPieceToBin() ⟹ Future(¬addPieceToBin(), 1);
63          singleInstantEvents2:
64              removePieceFromBin() ⟹ Future(¬removePieceFromBin(), 1);
65
66          -- binStatus stays constant if there is no action performed
67          constantBin:
68              ∀x(binStatus == x ∧ ¬(addPieceToBin() ∨ removePieceFromBin()) ⟹
69                  Future(binStatus == x, 1));
70
71          -- The arm can move only when the cart is stopped
72          armSpeedRelation:
73              currentEndEffectorSpeed ≠ None ⟹ currentCartSpeed == None
74
75  end RobotStatus.
```

3.6 Definition of the RobotController class

```
class RobotController
  -- RobotController handles all the actions that the robot can execute,
  -- specifying preconditions, axioms over the duration of the action
  -- and the sequences of events that must happen during the action

  temporal domain integer;

  TD items

  predicates
    do({
    PickFromBinArea,
    DropToLocalBin,
    GoToTombstone,
    PickFromLocalBin,
    DropToTombstone,
    PickFromTombstone,
    GoToConveyorBelt,
    DropToConveyorBelt,
    GoToBinArea
    });

    isArmMoving(),
    PickFromBinAreaT0(), PickFromBinAreaT1(),
    PickFromBinAreaT2(), PickFromBinAreaT3(),
    DropToTombstoneT0(), DropToTombstoneT1(),
    DropToTombstoneT2(), DropToTombstoneT3(),
    PickFromTombstoneT0(), PickFromTombstoneT1(),
    PickFromTombstoneT2(), PickFromTombstoneT3();
    DropToConveyorBeltT0(), DropToConveyorBeltT1(),
    DropToConveyorBeltT2(), DropToConveyorBeltT3();

  variables
    currentAction({
    PickFromBinArea,
    DropToLocalBin,
    GoToTombstone,
    PickFromLocalBin,
    DropToTombstone,
    PickFromTombstone,
    GoToConveyorBelt,
    DropToConveyorBelt,
    GoToBinArea
    }),

    -- None -> Proceed normally with the execution
    -- Emergency -> Stop the robot and wait for the HDICommand to return to None
    -- Continue -> when the robot has placed a piece on the tombstone, it needs
    -- to wait for the lavoration to be done and for the operator to signal that
    -- the lavoration has terminated.

```

```
52      HDICommand({
53      None,
54      Emergency,
55      Continue
56      });
57
58    TI items
59
60
61    modules RobotStatus: RobotStatus,
62                  Robot: RobotPositionSensor;
63
64
65
66    axioms
67
68    -- CORRECT FLOW OF ACTIONS
69
70      -- the actions must be done in the correct order
71      correctActionOrder:
72        (currentAction == PickFromBinArea ⇒
73          Until(currentAction == PickFromBinArea, currentAction == DropToLocalBin)) ∧
74
75        -- After having dropped a piece in the local bin, the robot either takes
76        -- another or goes to the tombstone
77        (currentAction == DropToLocalBin ⇒
78          (Until(currentAction == DropToLocalBin, currentAction == PickFromBinArea) ||
79          Until(currentAction == DropToLocalBin, currentAction == GoToTombstone))) ∧
80
81        -- After having gone to the tombstone, the robot takes the stored working
82        -- piece with the end effector
83        (currentAction == GoToTombstone ⇒
84          (Until(currentAction == GoToTombstone, currentAction == PickFromLocalBin))) ∧
85
86        -- After having picked up a piece from the local bin, the robot drops it
87        -- into the tombstone
88        (currentAction == PickFromLocalBin ⇒
89          Until(currentAction == PickFromLocalBin, currentAction == DropToTombstone)) ∧
90
91        -- After having dropped a piece into the tombstone, the robot waits for
92        -- the operator's signal and then picks up the reshaped piece
93        (currentAction == DropToTombstone ⇒
94          Until(currentAction == DropToTombstone, currentAction = PickFromTombstone)) ∧
95
96        -- After having picked up the piece from the tombstone, the robot goes to
97        -- the conveyor belt
98        (currentAction == PickFromTombstone ⇒
99          Until(currentAction == PickFromTombstone, currentAction = GoToConveyorBelt)) ∧
100
101
102       -- After having gone to the conveyor belt, the robot drops the piece onto
103       -- it
104       (currentAction == GoToConveyorBelt ⇒
```

```
105         Until(currentAction == GoToConveyorBelt, currentAction == DropToConveyorBelt) ∧
106
107       -- After having dropped the piece onto the conveyor belt, the robot either
108       -- goes back to the tombstone or it goes to the bin area if local bin is
109       -- empty
110       (currentAction == DropToConveyorBelt ⇒
111         Until(currentAction == DropToConveyorBelt, currentAction == GoToTombstone) ||
112         Until(currentAction == DropToConveyorBelt, currentAction == GoToBinArea)) ∧
113
114       -- After having gone to the bin area, the robot picks up a piece from the
115       -- bin area
116       (currentAction == GoToBinArea ⇒
117         Until(currentAction == GoToBinArea, currentAction == PickFromBinArea);
118
119     -- the change in currentAction must be preceded by a do request
120     correctInit:
121       ∀x(Becomes(currentAction == x) ⟺ do(x));
122
123
124     -- during emergency mode, the robot needs to be stopped immediately
125     emergencyMode:
126       (HDICommand == Emergency) ⇒
127         (RobotStatus.currentCartSpeed == None) ∧ (RobotStatus.currentEndEffectorSpeed == None);
128
129     -- do is an instantaneous event (lasts only one time instant)
130     instantaneousDo:
131         ∀x(do(x) ⇒ Future(¬do(x), 1));
132
133   -- DEFINITION OF PRECONDITIONS, AXIOMS HOLDING DURING THE ACTION
134   -- AND SEQUENCES OF EVENTS
135
136   -- PickFromBinArea
137     -- the robot has space in the local bin and is at the bin area
138     prePickFromBinArea:
139         Becomes(currentAction == PickFromBinArea) ⇒
140             ¬RobotStatus.binFull() ∧ ¬Robot.Arm.holding()∧
141             ∀x(Robot.Cart.position(x) ⇒ Grid.adjToBinArea(x));
142
143     -- the robot must be still
144     duringPickFromBinArea:
145         currentAction == PickFromBinArea ⇒ RobotStatus.currentCartSpeed == None;
146
147
148     -- t0: Start -> end effector on top of cart, not holding
149     -- t1: end effector on top of bin area, not holding
150     -- t2: end effector on top of bin area, holding
151     -- t3: End -> end effector on top of cart, holding
152     PickFromBinAreaT0() ⟺ (currentAction == PickFromBinArea ∧ ¬Robot.Arm.holding()∧
153         ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x)));
154
155     PickFromBinAreaT1() ⟺ (currentAction == PickFromBinArea ∧ ¬Robot.Arm.holding()∧
156         Robot.Arm.endEffector(LBA)));
157
```

```
158     PickFromBinAreaT2() ⟺ (currentAction == PickFromBinArea ∧ Robot.Arm.holding()∧
159         Robot.Arm.endEffector(LBA));
160
161     PickFromBinAreaT3() ⟺ (currentAction == PickFromBinArea ∧ Robot.Arm.holding()∧
162         ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x)));
163
164     -- to complete the PickFromBinArea action we need to have reached
165     -- PickFromBinAreaT3()
166     Becomes(¬currentAction == PickFromBinArea) ⟹ Past(PickFromBinAreaT3(), 1);
167
168     -- from T3 to T2 the only thing that changes is the position
169     -- of the end effector
170     PickFromBinAreaT3() ⟹ ∃t(LastTime(PickFromBinAreaT2(), t)∧
171         Lasted_ie(currentAction == PickFromBinArea ∧ Robot.Arm.holding(), t));
172
173     -- from T2 to T1 the only thing that changes is the fact that
174     -- the end effector is holding a piece
175     PickFromBinAreaT2() ⟹ ∃t(LastTime(PickFromBinAreaT1(), t)∧
176         Lasted_ie(currentAction == PickFromBinArea ∧ Robot.Arm.endEffector(LBA)));
177
178     -- from T1 to T0 the only thing that changes is the position
179     -- of the end effector
180     PickFromBinAreaT1() ⟹ ∃t(LastTime(PickFromBinAreaT0(), t)∧
181         Lasted_ie(currentAction == PickFromBinArea ∧ ¬Robot.Arm.holding()));
182
183 -- DropToLocalBin
184     -- the robot holds a piece, the local bin isn't full and the endEffector
185     -- is on the cart
186     preDropToLocalBin:
187         Becomes(currentAction == DropToLocalBin) ⇒
188             ¬RobotStatus.binFull() ∧ Robot.Arm.holding()∧
189             ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x));
190
191     -- the robot cannot move, the endEffector must stay in the same zone
192     duringDropToLocalBin:
193         currentAction == DropToLocalBin ⇒
194             (RobotStatus.currentCartSpeed == None∧
195             RobotStatus.currentEndEffectorSpeed == None);
196
197     -- there exists a time t in which the robot switches from
198     -- holding to not holding
199     dropSequence:
200         Becomes(¬currentAction == DropToLocalBin) ⇒
201             ∃t(LastTime(currentAction == DropToLocalBin ∧ Robot.Arm.holding(), t)∧
202             Lasted_ie(currentAction == DropToLocalBin, t));
203
204     -- after the drop to local bin action has finished, then the amount of pieces
205     -- that are stored inside the local bin has increased by one unit
206     -- t_1: the time when DropToLocalBin has started
207     -- t_2: the time during which the piece has been dropped from the
208     --      end effector to the local bin
209     dropOnePieceToLocalBin:
210         ∃t_1(LastTime(¬currentAction == DropToLocalBin, t_1)∧
```

```
211                 ∃t2(0 < t2 < t1 ∧ Past(RobotStatus.addPieceToBin(), t2)∧
212                     ∀t3(0 < t3 < t1 ∧ t3 ≠ t2 ⟹ ¬Past(RobotStatus.addPieceToBin(), t3)));
213
214   -- GoToTombstone
215     -- the robot's local bin is full, it isn't holding any piece and the
216     -- endEffector is on the cart
217     preGoToTombstone:
218         Becomes(currentAction == GoToTombstone) ⇒
219             RobotStatus.binFull() ∧ ¬RobotStatus.holding();
220
221     -- in case the robot stops himself during the action of going towards the
222     -- tombstone, then the endEffector cannot move
223     -- in RobotPositionSensor we  already stated that if the cart is moving then
224     -- the endEffector must stay on top of the cart
225     duringGoToTombstone:
226         currentAction == GoToTombstone ⇒
227             (RobotStatus.currentCartSpeed == None ⇒
228                 RobotStatus.currentEndEffectorSpeed == None);
229
230
231
232     -- there exists a time t smaller than MaxTravelTime when the robot is in a
233     -- position adjacent to the tombstone
234     goToTombstoneSequence:
235     Becomes(¬currentAction == GoToTombstone) ⟹
236         ∃x(Robot.Cart.position(x) ∧ Grid.adjToTombstone(x));
237
238   -- PickFromLocalBin
239     -- the robot isn't holding any piece, the local bin isn't empty and the
240     -- endEffector is in the same position as the cart
241     prePickFromLocalBin:
242         Becomes(currentAction == PickFromLocalBin) ⇒
243             ¬RobotStatus.binEmpty() ∧ ¬Robot.Arm.holding() ∧
244             ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x));
245
246     -- the robot cannot move if the action is PickFromLocalBin, and also the
247     -- endEffector does not have to move
248     duringPickFromLocalBin:
249         currentAction == PickFromLocalBin ⇒
250             (RobotStatus.currentCartSpeed == None ∧
251             RobotStatus.currentEndEffectorSpeed == None);
252
253     postPickFromLocalBin:
254         Becomes(¬currentAction == PickFromLocalBin) ⟹ Robot.Arm.holding()∧
255             ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x));
256
257     -- t1 is the starting time of the action
258     -- t2 is the time during which the piece is taken from the local bin
259     removedOnlyOnePieceFromBin:
260         Becomes(¬currentAction == PickFromLocalBin) ⟹
261             ∃t1(LastTime(¬currentAction == PickFromLocalBin, t1)∧
262                 ∃t2(0 < t2 < t1 ∧ Past(RobotStatus.removePieceFromBin(), t2)∧
263                     ∀t3(0 < t3 < t1 ∧ t2 ≠ t3 ⟹ ¬Past(RobotStatus.removePieceFromBin(), t3)));
```

```
264
265   -- DropToTombstone
266     -- the robot is holding a piece and it is in a position adjacent
267     -- to the tombstone
268     preDropToTombstone:
269         Becomes(currentAction == DropToTombstone) ⇒
270             Robot.Arm.holding() ∧ ∀x(Robot.Cart.position(x) ⇒ Grid.adjToTombstone(x));
271
272     -- the robot cart isn't moving
273     duringDropToTombstone:
274         currentAction == DropToTombstone ⇒
275             RobotStatus.currentCartSpeed == None;
276
277     -- t0: Start -> end effector on top of cart, holding
278     -- t1: end effector on top of tombstone, holding
279     -- t2: end effector on top of tombstone, not holding
280     -- t3: End -> end effector on top of cart, not holding
281     DropToTombstoneT0() ⟺ (currentAction == DropToTombstone ∧ Robot.Arm.holding()∧
282         ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x)));
283
284     DropToTombstoneT1() ⟺ (currentAction == DropToTombstone ∧ Robot.Arm.holding()∧
285         Robot.Arm.endEffector(L0));
286
287     DropToTombstoneT2() ⟺ (currentAction == DropToTombstone ∧ ¬Robot.Arm.holding()∧
288         Robot.Arm.endEffector(L0));
289
290     DropToTombstoneT3() ⟺ (currentAction == DropToTombstone ∧ ¬Robot.Arm.holding()∧
291         ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x)));
292
293     -- to complete the DropToTombstone action we need to have reached
294     -- DropToTombstoneT3()
295     Becomes(¬currentAction == DropToTombstone) ⟹ Past(DropToTombstoneT3(), 1);
296
297     -- from T3 to T2 the only thing that changes is the position
298     -- of the end effector
299     DropToTombstoneT3() ⟹ ∃t(LastTime(DropToTombstoneT2(), t)∧
300         Lasted_ie(currentAction == DropToTombstone ∧ ¬Robot.Arm.holding(), t));
301
302     -- from T2 to T1 the only thing that changes is the fact that
303     -- the end effector is holding a piece
304     DropToTombstoneT2() ⟹ ∃t(LastTime(DropToTombstoneT1(), t)∧
305         Lasted_ie(currentAction == DropToTombstone ∧ Robot.Arm.endEffector(L0)));
306
307     -- from T1 to T0 the only thing that changes is the position
308     -- of the end effector
309     DropToTombstoneT1() ⟹ ∃t(LastTime(DropToTombstoneT0(), t)∧
310         Lasted_ie(currentAction == DropToTombstone ∧ Robot.Arm.holding()));
311
312   -- PickFromTombstone
313     -- the robot has received the Continue HDI signal, it isn't holding anything
314     -- and it is in a position adjacent to the tombstone
315     prePickFromTombstone:
316         Becomes(currentAction == PickFromTombstone) ⇒
```

```
317                 HDICommand == Continue ∧
318                 ¬Robot.Arm.holding() ∧
319                 ∀x(Robot.Cart.position(x) ⇒ Grid.adjToTombstone(x));
320
321     -- the robot cart isn't moving
322     duringPickFromTombstone:
323         currentAction == PickFromTombstone ⇒ RobotStatus.currentCartSpeed == None;
324
325
326     -- t0: Start -> end effector on top of cart, not holding
327     -- t1: end effector on top of tombstone, not holding
328     -- t2: end effector on top of tombstone, holding
329     -- t3: End -> end effector on top of cart, holding
330     PickFromTombstoneT0() ⟺ (currentAction == PickFromTombstone ∧ ¬Robot.Arm.holding()∧
331         ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x)));
332
333     PickFromTombstoneT1() ⟺ (currentAction == PickFromTombstone ∧ ¬Robot.Arm.holding()∧
334         Robot.Arm.endEffector(L0)));
335
336     PickFromTombstoneT2() ⟺ (currentAction == PickFromTombstone ∧ Robot.Arm.holding()∧
337         Robot.Arm.endEffector(L0)));
338
339     PickFromTombstoneT3() ⟺ (currentAction == PickFromTombstone ∧ Robot.Arm.holding()∧
340         ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x)));
341
342     -- to complete the PickFromTombstone action we need to have reached
343     -- PickFromTombstoneT3()
344     Becomes(¬currentAction == PickFromTombstone) ⟹ Past(PickFromTombstoneT3(), 1);
345
346     -- from T3 to T2 the only thing that changes is the position
347     -- of the end effector
348     PickFromTombstoneT3() ⟹ ∃t(LastTime(PickFromTombstoneT2(), t)∧
349         Lasted_ie(currentAction == PickFromTombstone ∧ Robot.Arm.holding(), t));
350
351     -- from T2 to T1 the only thing that changes is the fact that
352     -- the end effector is holding a piece
353     PickFromTombstoneT2() ⟹ ∃t(LastTime(PickFromTombstoneT1(), t)∧
354         Lasted_ie(currentAction == PickFromTombstone ∧ Robot.Arm.endEffector(L0)));
355
356     -- from T1 to T0 the only thing that changes is the position
357     -- of the end effector
358     PickFromTombstoneT1() ⟹ ∃t(LastTime(PickFromTombstoneT0(), t)∧
359         Lasted_ie(currentAction == PickFromTombstone ∧ ¬Robot.Arm.holding()));
360
361 -- GoToConveyorBelt
362     -- the robot is holding a piece
363     preGoToConveyorBelt:
364         Becomes(currentAction == GoToConveyorBelt) ⇒ RobotStatus.holding();
365
366     -- in case the robot stops himself during the action of going towards the
367     -- conveyor belt, then the endEffector cannot move
368     -- in RobotPositionSensor we already stated that if the cart is moving then
369     -- the endEffector must stay on top of the cart
```

```
    duringGoToConveyorBelt:
        currentAction == GoToConveyorBelt ⇒
            ((RobotStatus.currentCartSpeed == None
                ⇒ RobotStatus.currentEndEffectorSpeed == None)
            ∧Robot.Arm.holding());

    postGoToConveyorBelt:
        Becomes(¬currentAction == GoToConveyorBelt) ⟹
            ∃x(Robot.Cart.position(x) ∧ Grid.adjToConveyorBelt(x));

-- DropToConveyorBelt
    -- the robot is holding a piece and it is in a position adjacent to the
    -- conveyor belt
    preDropToConveyorBelt:
      Becomes(currentAction == DropToConveyorBelt) ⇒
        Robot.Arm.holding() ∧ ∀x(Robot.Cart.position(x) ⇒
          Grid.adjToConveyorBelt(x));

    -- the robot cart isn't moving
    duringDropToConveyorBelt:
      currentAction == DropToConveyorBelt ⇒ RobotStatus.currentCartSpeed == None;

    -- t0: Start -> end effector on top of cart, holding
    -- t1: end effector on top of conveyor belt, holding
    -- t2: end effector on top of conveyor belt, not holding
    -- t3: End -> end effector on top of cart, not holding
    DropToConveyorBeltT0() ⟺ (currentAction == DropToConveyorBelt ∧ Robot.Arm.holding()∧
        ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x)));

    DropToConveyorBeltT1() ⟺ (currentAction == DropToConveyorBelt ∧ Robot.Arm.holding()∧
        Robot.Arm.endEffector(LCB));

    DropToConveyorBeltT2() ⟺ (currentAction == DropToConveyorBelt ∧ ¬Robot.Arm.holding()∧
        Robot.Arm.endEffector(LCB));

    DropToConveyorBeltT3() ⟺ (currentAction == DropToConveyorBelt ∧ ¬Robot.Arm.holding()∧
        ∃x(Robot.Cart.position(x) ∧ Robot.Arm.endEffector(x)));

    -- to complete the DropToConveyorBelt action we need to have reached
    -- DropToConveyorBeltT3()
    Becomes(¬currentAction == DropToConveyorBelt) ⟹ Past(DropToConveyorBeltT3(), 1);

    -- from T3 to T2 the only thing that changes is the position
    -- of the end effector
    DropToConveyorBeltT3() ⟹ ∃t(LastTime(DropToConveyorBeltT2(), t)∧
        Lasted_ie(currentAction == DropToConveyorBelt ∧ ¬Robot.Arm.holding(), t));

    -- from T2 to T1 the only thing that changes is the fact that
    -- the end effector is holding a piece
    DropToConveyorBeltT2() ⟹ ∃t(LastTime(DropToConveyorBeltT1(), t)∧
        Lasted_ie(currentAction == DropToConveyorBelt ∧ Robot.Arm.endEffector(LCB)));

    -- from T1 to T0 the only thing that changes is the position
```

```
423       -- of the end effector
424       DropToConveyorBeltT1() ⟹ ∃t(LastTime(DropToConveyorBeltT0(), t)∧
425             Lasted_ie(currentAction == DropToConveyorBelt ∧ Robot.Arm.holding()));
426
427     -- GoToBinArea
428       -- the robot doesn't hold any piece and the bin is empty
429       preGoToBinArea:
430             Becomes(currentAction == GoToBinArea) ⇒
431                   ¬RobotStatus.holding() ∧ RobotStatus.binEmpty();
432
433       -- in case the robot stops himself during the action of going towards the
434       -- bin area, then the endEffector cannot move
435       -- in RobotPositionSensor we already stated that if the cart is moving then
436       -- the endEffector must stay on top of the cart
437       duringGoToBinArea:
438             currentAction == GoToBinArea ⇒
439                   (RobotStatus.currentCartSpeed == None ⇒
440                         RobotStatus.currentEndEffectorSpeed == None);
441
442       postGoToBinArea:
443             Becomes(¬currentAction == GoToBinArea) ⇒
444                   ∃x(Robot.Cart.position(x)  ∧ Grid.adjToBinArea(x));
445
446     -- The robot can move the arm only when the cart is not moving
447     isArmMoving:
448       isArmMoving() ⟺
449             ¬(currentAction == GoToBinArea ∨ currentAction == GoToTombstone∨
450             currentAction == GoToConveyorBelt);
451
452 end RobotController.
```

3.7 Definition of the SafetyChecker class

```
-- Here we define all the constraints necessary for the safety of O
-- in the presence of a working KUKA R around the layout, all the actions performed
-- by R that could cause harm to O are avoided or limited as much as possible.
class SafetyChecker
    temporal domain integer;

    modules  Operator: OperatorPositionSensor,
             Robot: RobotPositionSensor,
             RobotStatus: RobotStatus,
             RobotController: RobotController,
             Grid: Grid;

        axioms

        ------------------------
        -- SAFETY ARM AXIOMS --
        ------------------------

        -- if the end effector is in the same area as the head of the operator,
        -- then the end effector is stopped
        headSameAreaAsRobotArm:
            ∀x(Robot.Arm.position(x) ∧ Operator.head(x) ∧ RobotController.isArmMoving() ⇒
                RobotStatus.currentEndEffectorSpeed == None);

        -- if the head of the operator is close to the arm of the robot, then
        -- the arm is either moving slowly or staying still
        headCloseToRobotArm:
            ∀x, y(x ≠ y ∧ Robot.Arm.position(x) ∧ Operator.head(y)∧
                Grid.adjacent(x, y) ∧ RobotController.isArmMoving() ⇒
                    (RobotStatus.currentEndEffectorSpeed == None∨
                    RobotStatus.currentEndEffectorSpeed == Low));

        -- if the arms of the operator are in the same area as the arm
        -- of the robot, then either the arm of the robot is staying
        -- still or is moving slow.
        sameAreaArms:
            ∀x(Robot.Arm.position(x) ∧ Operator.arms(x) ∧ ¬Robot.Arm.contact() ⇒
                RobotStatus.currentEndEffectorSpeed == None∨
                RobotStatus.currentEndEffectorSpeed == Low);

        -- if the arms of the operator are touching the arm of the robot,
        -- then the arm is not moving
        sameAreaArmsWithContact:
            ∀x(Robot.Arm.position(x) ∧ Operator.arms(x) ∧ Robot.Arm.contact() ⇒
                RobotStatus.currentEndEffectorSpeed == None);

        -- if the arm of the robot is in an area adjacent to the arms
        -- of the operator, then the robot is at most moving slow
        armsCloseArm:
            ∀x, y(Robot.Arm.position(x) ∧ Operator.arms(y) ∧ x ≠ y ∧ Grid.adjacent(x, y) ⇒
                RobotStatus.currentEndEffectorSpeed == None∨
```

```
52                  RobotStatus.currentEndEffectorSpeed == Slow);
53
54          -- if the arm of the robot is in the same area as the operator's body,
55          -- then the arm is not moving
56          armCloseBody:
57              ∀x(Robot.Arm.position(x) ∧ Operator.body(x) ⇒
58                  RobotStatus.currentEndEffectorSpeed == None);
59
60          -- if the arm of the robot is in an area adjacent to the body
61          -- of the operator, then the robot is at most moving slow
62          bodyCloseArm:
63              ∀x, y(Robot.Arm.position(x) ∧ Operator.body(y) ∧ x ≠ y ∧ Grid.adjacent(x, y) ⇒
64                      RobotStatus.currentEndEffectorSpeed == None∨
65                      RobotStatus.currentEndEffectorSpeed == Slow);
66
67          ------------------------
68          -- SAFETY CART AXIOMS --
69          ------------------------
70
71          -- if the operator is in an area close to the wall or a danger area and the
72          -- robot is in an area adjacent to it, then the robot will not
73          -- move to an area adjacent to the one occupied by the
74          -- operator
75          operatorCloseToWall:
76              ∀x, y(Operator.body(x) ∧ (Grid.border(x) ∨ Grid.danger(x))∧
77                  Robot.Cart.position(y) ∧ Grid.adjacent(x, y) ⇒
78                      ((Future(RobotStatus.currentCardSpeed == None ∧ Cart.position(y) ∧ Operator.body(y), 1))∨
79                       (Future(∃w∃z(z ≠ w ∧ Cart.position(w) ∧ Operator.body(z)), 1))
80
81          -- The robot will not go to an area that is dangerous or an area near the
82          -- border, that may be occupied in the next time instant by the operator.
83          doNotRunOverOperator:
84              ∀x∀y∀z(Operator.body(x) ∧ Cart.position(y) ∧ (Grid.border(z) ∨ Grid.danger(z))∧
85                      ((Grid.adjacent(x, z) ∧ Grid.adjacent(z, y) ∨ (x = z ∧ Grid.adjacent(x, y)))) ⇒
86                          Future(¬Cart.position(z), 1))
87
88          cartDoesNotMoveIfOperatorIsInSameArea:
89              ∀x((Cart.position(x) ∧ Operator.body(x) ∧ (Grid.danger(x) ∨ Grid.border(x))) ⇒
90                  RobotStatus.currentCardSpeed == None);
91
92          -- if the operator is in a transit area and the robot is in
93          -- the same area, then the robot needs to move slow
94          operatorTransitZone:
95              ∀x(Operator.body(x) ∧ Grid.transit(x) ∧ Robot.Cart.position(x) ⇒
96                  RobotStatus.currentCartSpeed ≠ Normal);
97
98  end SafetyChecker.
```

3.8 Safety Properties

The model needs to ensure that there will be no harmful contact between the robot and the operator. For each safety property we need to show that the conjuction of the model and the negation of the safety property is unsatisfiable: this proves that no history produced by the analysis of the TRIO axioms can lead to an unsafe situation. All the following safety properties are stated without the use of the Always operator, but it is implicit in their definitions.

The **first** and most important safety property that we need to ensure is that the robot needs to avoid at all costs hitting the most delicate parts of the operator: the head and the arms.

$$\neg\exists x(Robot.Arm.endEffector(x) \wedge Operator.head(x) \wedge \neg(RobotStatus.currentEndEffectorSpeed == None))$$

$$\neg\exists x(Robot.Arm.endEffector(x) \wedge Operator.arms(x) \wedge \neg(RobotStatus.currentEndEffectorSpeed == None \vee RobotStatus.currentEndEffectorSpeed == Low))$$

These safety properties are ensured by the axioms `headSameAreaAsRobotArm`, `headCloseToRobotArm`, `sameAreaArms`, `sameAreaArmsWithContact` of the SafetyChecker class.

The **second** property we want to prove is the fact that the cart will never have a speed different from none if the operator is in the same area as the cart, and the area is a danger or border one.

$$\neg\exists x((Grid.danger(x) \vee Grid.border(x)) \wedge Cart.position(x) \wedge Operator.body(x) \wedge RobotStatus.currentCartSpeed \neq None)$$

This property is ensured by the axiom `cartDoesNotMoveIfOperatorIsInSameArea` of the SafetyChecker class.

The **third** property ensures that the robot can only move slowly or stand still if it's in a transit zone with the operator.

$$\neg\exists x(Grid.transit(x) \wedge Robot.Cart.position(x) \wedge Operator.body(x) \wedge RobotStatus.currentCartSpeed == Normal)$$

This property is ensured by the axiom `operatorTransitZone` of the SafetyChecker class.

The **fourth** property we want to prove is the fact that the cart does not move into a dangerous or border area already occupied by the operator.

$$\neg\exists x((Grid.danger(x) \vee Grid.border(x)) \wedge Cart.position(x) \wedge Operator.body(x) \wedge Cart.moved())$$

This property is ensured by the `doNotRunOverOperator` axiom of the SafetyChecker class.

A Area modeling

Here we provide a sample of the formula that states for every possible combination of 2 cells in the layout, which one are adjacent to each other and which one are not.

$$\neg Adj(L0, L0) \wedge Adj(L0, L1) \wedge Adj(L0, L2) \wedge Adj(L0, L3) \wedge Adj(L0, L4) \wedge \neg Adj(L0, L5) \wedge \neg Adj(L0, L6)$$
$$\wedge \neg Adj(L0, L7) \wedge \neg Adj(L0, L8) \wedge \neg Adj(L0, L9) \wedge \neg Adj(L0, L10) \wedge \neg Adj(L0, L11) \wedge \neg Adj(L0, L12)$$
$$\wedge \neg Adj(L0, L13) \wedge \neg Adj(L0, L14) \wedge \neg Adj(L0, L15) \wedge \neg Adj(L0, L16) \wedge \neg Adj(L0, L17) \wedge \neg Adj(L0, L18)$$
$$\wedge \neg Adj(L0, L19) \wedge \neg Adj(L0, L20) \wedge \neg Adj(L0, L21) \wedge \neg Adj(L0, L22) \wedge \neg Adj(L0, L23) \wedge \neg Adj(L0, L24)$$
$$\wedge \neg Adj(L0, L25) \wedge \neg Adj(L0, L26) \wedge \neg Adj(L0, L27) \wedge \neg Adj(L0, L28) \wedge Adj(L1, L0) \wedge \neg Adj(L1, L1) \wedge Adj(L1, L2)$$
$$\wedge \neg Adj(L1, L3) \wedge \neg Adj(L1, L4) \wedge Adj(L1, L5) \wedge Adj(L1, L6) \wedge \neg Adj(L1, L7) \wedge \neg Adj(L1, L8) \wedge \neg Adj(L1, L9)$$
$$\wedge \neg Adj(L1, L10) \wedge \neg Adj(L1, L11) \wedge \neg Adj(L1, L12) \wedge \neg Adj(L1, L13) \wedge \neg Adj(L1, L14) \wedge \neg Adj(L1, L15)$$
$$\wedge \neg Adj(L1, L16) \wedge \neg Adj(L1, L17) \wedge \neg Adj(L1, L18) \wedge \neg Adj(L1, L19) \wedge \neg Adj(L1, L20) \wedge \neg Adj(L1, L21)$$
$$\wedge \neg Adj(L1, L22) \wedge \neg Adj(L1, L23) \wedge \neg Adj(L1, L24) \wedge \neg Adj(L1, L25) \wedge \neg Adj(L1, L26) \wedge \neg Adj(L1, L27)$$
$$\wedge \neg Adj(L1, L28) \wedge Adj(L2, L0) \wedge Adj(L2, L1) \wedge \neg Adj(L2, L2) \wedge Adj(L2, L3) \wedge \neg Adj(L2, L4) \wedge Adj(L2, L5)$$
$$\wedge Adj(L2, L6) \wedge Adj(L2, L7) \wedge \neg Adj(L2, L8) \wedge \neg Adj(L2, L9) \wedge \neg Adj(L2, L10) \wedge \neg Adj(L2, L11)$$
$$\wedge \neg Adj(L2, L12) \wedge \neg Adj(L2, L13) \wedge \neg Adj(L2, L14) \wedge \neg Adj(L2, L15) \wedge \neg Adj(L2, L16) \wedge \neg Adj(L2, L17)$$
$$\wedge \neg Adj(L2, L18) \wedge \neg Adj(L2, L19) \wedge \neg Adj(L2, L20) \wedge \neg Adj(L2, L21) \wedge \neg Adj(L2, L22) \wedge \neg Adj(L2, L23)$$
$$\wedge \neg Adj(L2, L24) \wedge \neg Adj(L2, L25) \wedge \neg Adj(L2, L26) \wedge \neg Adj(L2, L27) \wedge \neg Adj(L2, L28) \wedge Adj(L3, L0)$$
$$\wedge \neg Adj(L3, L1) \wedge Adj(L3, L2) \wedge \neg Adj(L3, L3) \wedge Adj(L3, L4) \wedge \neg Adj(L3, L5) \wedge Adj(L3, L6)$$
$$\ldots$$