Delft University of Technology

Faculty of Electrical Engineering,
Mathematics and Computer Science

System Validation Project

Group 22

Authors, Student Nos.:

Kaustubh Agarwal, 4823168

Ana Aldescu, 4929411

Aniket Ashwin Samant, 4838866

Şerban Vădineanu, 4824989

October 2018

Contents

Chapter 1

Introduction

The System Validation course of TU Delft proposes a project assignment concerning the design of a controller for a small distributed embedded system. The goal of the assignment is to design, model, and verify a control system for a simplified machine that prepares wafers for the production of microchips. Following the different stages of this project, we should be able to properly formulate system requirements and represent the model using labeled transition systems. This document presents the constituent phases of the project. Chapter 2 presents the requirements of the entire system that are formally verifiable. In chapter 3 we present the interactions taking place between the components defined in chapter 2. The architecture of the system is shown in chapter 4. Chapter 6 presents the modeled behaviour of all the controllers presented in the architecture. In chapter 5 the model is verified with the translated requirements presented in chapter 2. Conclusions and other remarks are presented in chapter 8.

For a better understanding of the system's operation (in the semantic domain), this chapter also presents a list of physical components present in the system that work together to satisfy the requirements, which will be presented in the later chapters. Moreover, an informal description of the sequences of actions performed by the various components is also presented before proceeding towards the formal requirements, modelling, and verification.

1.1 System Components

- An UV lamp L - projects the wafer when placed underneath
- A vacuum chamber
- Two airlocks A1, A2
- Two sets of doors, one set per airlock
  - Inner doors: DI1, DI2
  - Outer doors: DO1, DO2
- Three robots for moving wafers around

  - Outside the vacuum chamber: R1, R2
  - Inside the vacuum chamber: R3
- Two sets of wafer stacks where wafers are loaded
  - Input stacks: I1, I2
  - Output stacks: O1, O2
- One lamp sensor - Lamp has wafer/does not have a wafer underneath
- Two airlock sensors - Corresponding airlock has/does not have a wafer
- Two input stack sensors - Corresponding stack is empty/not empty
- Two output stack sensors - Corresponding stack is full/not full

1.2 Sequences

We have identified two main sequences of the process. The first sequence describes the steps needed to move a blank wafer from the input stack(s) to the lamp, while the second sequence describes how a wafer reaches the output stack(s) after being projected by the lamp.

1.2.1 Wafer to lamp

1. Outer robot picks up a wafer from the corresponding input stack
2. Airlock's outer door opens
3. Robot drops wafer inside airlock
4. Airlock's outer door closes
5. Airlock's inner door opens
6. Inner robot picks up wafer from airlock
7. Inner robot drops wafer to lamp
8. Lamp turns on and projects the wafer

1.2.2 Wafer to output stack

1. Lamp turns off
2. Inner robot picks up wafer from lamp
3. Inner robot drops wafer inside airlock
4. Airlock's inner door closes
5. Airlock's outer door opens

6. Outer robot picks up wafer from airlock
7. Outer robot drops wafer in the corresponding output stack

Chapter 2

Requirements

The first step of the design process is to identify the requirements for the wafer projecting system we are considering. The components described in the previous chapter will have to follow some defined set of behaviors so as to meet all the requirements of the system, as will be presented hence.
The requirements are presented in the form of safety requirements and liveness requirements, and are verifiable using formal logic.

2.1 Safety Requirements

These requirements ensure that the system is always in a stable state, that is, each one of them needs to be satisfied by the components for us to say that the system is behaving normally; the violation of even one of these requirements would mean that the system has malfunctioned.

1. Airlocks:
   (a) DI1/DI2 can only be opened if DO1/DO2 is closed.
   (b) DO1/DO2 can only be opened if DI1/DI2 is closed.
2. Lamp:
   (a) Each wafer dropped below the light can be projected only once before being picked up.
3. Robots:
   (a) R1/R2 can only drop a wafer into O1/O2 if the stack is not full.
   (b) R1/R2 can only pick-up a wafer from I1/I2 if the stack is not empty.
   (c) R1/R2 can only pick-up a wafer from A1/A2 if DO1/DO2 is open.
   (d) R1/R2 can only drop a wafer in A1/A2 if DO1/DO2 is open.
   (e) R3 can only pick-up a wafer from A1/A2 if DI1/DI2 is open.

(f) R3 can only drop a projected wafer in A1/A2 if DI1/DI2 is open.

(g) After picking up a wafer from A1/A2, R3 can only drop it back to A1/A2 after the wafer was dropped to the lamp.

(h) After picking up a wafer from its corresponding input stack, a robot can only drop it into its corresponding airlock.

(i) After picking up a wafer from its corresponding airlock, a robot can only drop it into its corresponding output stack.

(j) A wafer can only be placed into the output stack if it was projected by the lamp.

2.2 Liveness Requirements

The following liveness requirements ensure the movement of the wafers throughout the system.

1. From the input stack, a wafer should always be able to reach the lamp.
2. From the lamp, a wafer should always be able to reach the output stack.
3. The system must be deadlock-free.

Chapter 3

Interactions

This chapter presents the interactions that take place between the components we have described previously. The meaning of the data types mentioned below is as follows:

1. (Input) Stack IDs: IS1, IS2
2. (Output) Stack IDs: OS1, OS2
3. Robot IDs: R1, R2, R3
4. Airlock IDs: A1, A2
5. Location IDs:
   (Location IDs are used as a uniform datatype for certain interactions - for instance, the outer robots can move to three locations, viz. the correponding input and output stacks, and the airlock. The location IDs facilitate such interactions using the outer robot location IDs.)
   (a) Inner Robot Location IDs: I_A1, I_A2, LAMP, I_ND
   (Here, the *I_ND* stands for non-deterministic, as will be explained later)
   (b) Outer Robot Location IDs: O_AIRLOCK, INP_STACK, OUT_STACK
6. Airlock Door Type: INNER, OUTER
7. Robot Action Type: PICKUP, DROP
8. Wafer Sensor State: WAFERPRESENT, NOWAFER
9. Wafer State: PROJECTED, UNPROJECTED
10. Door State: OPEN, CLOSED
11. Lamp State: ON, OFF
12. Input Stack State: EMPTY, NEMPTY
13. Output Stack State: FULL, NFULL

Note that we have used overloaded methods in the code so as to avoid having confusing names for the same kind of actions.

3.1 External Interactions

The following actions describe the interactions that can be noticed when observing the system. These include any physical actions like the robots' picking up and dropping wafers, the airlocks' doors opening and closing, etc. If there are user actions, those are also ideally considered the external (inter)actions of the system. Communication taking place between the components is not included here, and will be dealt with in another section.

3.1.1 Actuator Actions

Here, the actions (acts) are of the form in which one of the parameters is the target of the action, and the other parameters are to characterize the physical action to be performed by the target.

airlock_setInnerDoorState (aID, doorState)	(aID: A1, A2 – doorState: OPEN, CLOSED)	Set the airlock's inner door as open or closed.
airlock_setOuterDoorState (aID, doorState)	(aID: A1, A2 – doorState: OPEN, CLOSED)	Set the airlock's outer door as open or closed.
robot_pickUpWafer (robotID, outerRobotLocationID)	(robotID: R1, R2 – outerRobotLocationID: O_AIRLOCK, INP_STACK)	Pick up the wafer from the given location
robot_pickUpWafer (innerRobotLocationID)	(innerRobotLocationID: I_A1, I_A2, LAMP)	Causes the inner robot R3 to pick up the wafer from the given location
robot_dropWafer (robotID, outerRobotLocationID)	(robotID: R1, R2 – outerRobotLocationID: O_AIRLOCK, OUT_STACK)	Cause the robot to drop the wafer at the given location
robot_dropWafer (innerRobotLocationID)	(innerRobotLocationID: I_A1, I_A2, LAMP)	Causes the inner robot R3 to drop the wafer at the given location
robot_checkInputStackState (stackID, inputStackState)	(stackID: IS1, IS2 – inputStackState: EMPTY, NEMPTY)	Check whether the input stack is empty or not empty
robot_checkOutputStackState (stackID, outputStackState)	(stackID: OS1, OS2 – outputStackState: FULL, NFULL)	Check whether the output stack is full or not full
outerRobot_moveToLocation (robotID, airlockID)	(robotID: R1, R2 – airlockID: A1, A2)	Move an outer robot to an airlock

outerRobot_moveToLocation (robotID, stackID)	(robotID: R1, R2 – stackID: IS1, IS2, OS1, OS2)	Move an outer robot to an input/output stack
innerRobot_moveToLocation (innerRobotLocationID)	(innerRobotLocationID: I_A1, I_A2, LAMP, I_ND)	Move the inner robot to A1, A2, the lamp, or to a non-deterministic (neutral) location
lamp_projectWafer()		External action of the lamp to project the wafer

There are some points to note here - for the outer robot, only 3 locations are defined (O_AIRLOCK, INP_STACK, and OUT_STACK) - this is because by the very definition of the outer robot, its movement is restricted to the corresponding locations (see the problem statement). Hence, we can simplify by defining only 3 locations - for instance, when we refer to R1's O_AIRLOCK, it can be no other location but Airlock 1 (and likewise for the stacks). However, for the sake of making the meaning of moving to a location a bit more clear, we have included overloaded methods for the outer robot which take in airlockIDs and stackIDs as parameters too. (For instance, outerRobot_moveToLocation takes in airlockID in one version, and stackID in the other)

For the inner robot, there is a location defined as *I_ND* - this is a *non-deterministic* (neutral) state at which the robot starts and ends its wafer retrieval and placing cycle, in order to have non-determinism (i.e. when the robot is idle, and both airlocks have an unprojected wafer, the robot should have the non-deterministic quality of going to either).

3.1.2 User Actions

These actions are included so that in case any faulty action occurs, the user can intervene. These actions are limited to the input stacks becoming empty, and the output stacks becoming full, for simplicity. However, it is expected that the user's intervention isn't required to keep the system running otherwise.

user_fillStack(stackID):	(stackID: IS1, IS2)	Fill the corresponding stack with wafers.
user_emptyStack(stackID):	(stackID: OS1, OS2)	Empty the corresponding output stack.

3.2 Communication Interactions

Communication between the several controllers of the system occurs in the form of send-receive interactions. Because this exchange happens at the same time, the send-receive pairs can be merged into a single interaction. The following list describes how data is exchanged between the system's controllers using the merged communication interactions.

comm_robotActionAck(robotID, robotActionType, outerRobotLocationID)	(robotID: R1, R2 – robotActionType: PICKUP, DROP – outerRobotLocationID: O_AIRLOCK, INP_STACK, OUT_STACK)	Communicate to a controller that the outer robot has performed the given action at the given location.
comm_robotActionAck(robotActionType, innerRobotLocationID)	(robotActionType: PICKUP, DROP – innerRobotLocationID: LAMP, I_A1, I_A2)	Communicate to a controller that the inner robot R3 has performed the given action at the given location.
comm_inputStackState(stackID, inputStackState)	(stackID: IS1, IS2 – inputStackState: EMPTY, NEMPTY)	Communicate to a controller the state of the input stack
comm_outputStackState(stackID, outputStackState)	(stackID: OS1, OS2 – outputStackState: FULL, NFULL)	Communicate to a controller the state of the output stack
comm_airlockWaferProjectionState(airlockID, waferState)	(airlockID: A1, A2 – waferState: PROJECTED, UNPROJECTED)	Communicate to a controller whether the wafer inside the airlock is projected or not
comm_airlockDoorState (airlockID, airlockDoorType, doorState)	(airlockID: A1, A2 – airlockDoorType: INNER, OUTER – doorState: OPEN, CLOSED)	Communicate to a controller whether the airlock's inner/outer door is open/closed

Similar to the external interactions, there are some methods in the communication actions that are overloaded for keeping the code clean and readable.

Chapter 4

Architecture

The controller system's architecture has been modelled such that the safety and liveness requirements are always met, by using the various sensors, actuators, and other components that are present in the system.

4.1 Assumptions

The main assumptions about the system are as follows:

- The robots perform the instructed actions without any error
- The lamp performs the projecting process without malfunctioning
- The other components may require the user to act upon them based on certain sensor data. To be precise,
  - the airlock doors may not work
  - the input stacks may become empty, and
  - the output stacks may become full

However, to keep our model simple, we assume that the user intervenes where necessary and repairs the door(s), refills the input stack(s), and clears the output stack(s) as and when required. Thus, we have not included the user's actions in the model.

4.2 System Architecture

The architecture of the system is represented graphically using the Figure 4.1. It consists of several controllers working in parallel to meet the requirements stated in chapter 2. The distribution of the above mentioned controllers is as follows:

- 2 outer robot controllers
- 2 airlock controllers

- 1 inner robot controller

For the most part, the system is expected to function without the need for any user input, but certain cases have been identified in which external "user interactions" are needed to meet the system's liveness requirements. These are indicated by green arrows in the diagram (but as stated in the assumptions section, they are assumed to be performed immediately as needed).

All physical actions of the system are modelled as external interactions (as stated in the requirements chapter, chapter 2). They are represented in the architecture diagram using black arrows.

There are various controllers for the components, and communication between the controllers is represented using blue arrows in the architecture diagram. These interactions take place between the controllers continuously and ensure that the system runs automatically. The controllers send the state of their associated components and poll for the states of other components as required, and make state transitions by performing the appropriate actions. The state transitions should not end up causing a deadlock within the overall system, at any cost.

There are some *passive* components as well that are not controllers but are needed for the system's functioning - the stacks and the lamp. These are indicated in the architecture diagram too, since the controllers make use of them through external actions to signify relevant physical actions.

The system never reaches a state of "completion" since it is expected to run continuously till one of the conditions requiring an external *user interaction* is met. Theoretically, if the user continually replenishes the input stack with new wafers and clears the output stack at the same rate, and if the airlock doors do not malfunction, the system can run for an indefinite period of time.

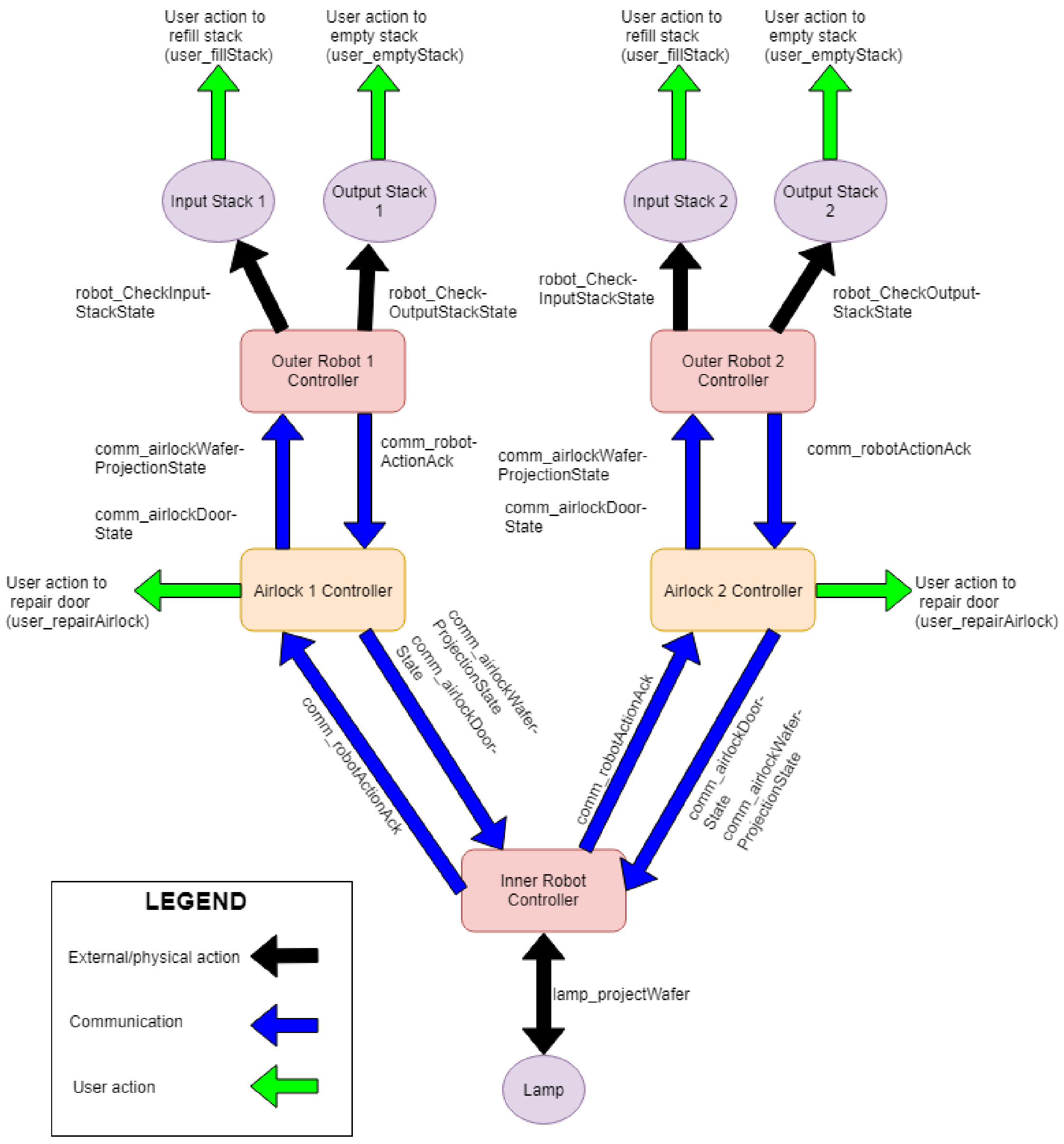

Figure 4.1: System architecture diagram

Chapter 5

Translated Requirements

The requirements stated in chapter 2 are translated to interactions between the controllers, and the resulting model is verified using μ-calculus. This chapter lists the requirements translated into the corresponding μ-calculus representation, taking one requirement at a time.

5.1 Translated safety requirements

1. Airlocks:

   (a) DI1/DI2 can only be opened if DO1/DO2 is closed.
   $[true*]\forall aID : airlockID.[airlock\_setOuterDoorState(aID, OPEN)$
   $.(!(airlock\_setOuterDoorState(aID, CLOSED)))*$
   $.airlock\_setInnerDoorState(aID, OPEN)]false$

   (b) DO1/DO2 can only be opened if DI1/DI2 is closed.
   $[true*]\forall aID : airlockID.[airlock\_setInnerDoorState(aID, OPEN)$
   $.(!(airlock\_setInnerDoorState(aID, CLOSED)))*$
   $.airlock\_setOuterDoorState(aID, OPEN)]false$

2. Lamp:

   (a) Each wafer dropped below the lamp can be projected only once before being picked up.
   $[true*][lamp\_projectWafer.(!robot\_pickUpWafer(R3, LAMP))*$
   $.lamp\_projectWafer]false$

3. Robots:

   (a) R1/R2 can only drop a wafer into O1/O2 if the stack is not full.

      i. $[true*]\forall rID : robotID.val(rID == R1 || rID == R2) => $
      $[outerRobot\_moveToLocation(rID, matchRobotOutputStack(rID))$
      $.robot\_checkOutputStackState(matchRobotOutputStack(rID), FULL)$
      $.robot\_dropWafer(rID, OUT\_STACK)]false$

ii. $[true*]\forall rID : robotID.val(rID == R1 || rID == R2) =>$
$[outerRobot\_moveToLocation(rID, matchRobotOutputStack(rID))$
$.(!robot\_checkOutputStackState(matchRobotOutputStack(rID), NFULL))*$

$.robot\_dropWafer(rID, OUT\_STACK)]false$

(b) R1/R2 can only pick up a wafer from I1/I2 if the stack is not empty.

i. $[true*]\forall rID : robotID.val(rID == R1 || rID == R2) =>$
$[outerRobot\_moveToLocation(rID, matchRobotInputStack(rID))$
$.robot\_checkInputStackState(matchRobotInputStack(rID), EMPTY)$
$.robot\_pickUpWafer(rID, INP\_STACK)]false$

ii. $[true*]\forall rID : robotID.val(rID == R1 || rID == R2) =>$
$[outerRobot\_moveToLocation(rID, matchRobotInputStack(rID))$
$.(!robot\_checkInputStackState(matchRobotInputStack(rID), NEMPTY))$
$.robot\_pickUpWafer(rID, INP\_STACK)]false$

(c) R1/R2 can only pick up a wafer from A1/A2 if DO1/DO2 is open.
$[true*]\forall rID : robotID.val(rID == R1 || rID == R2) =>$
$[airlock\_setOuterDoorState(matchRobotAirlock(rID), CLOSED)$
$.(!airlock\_setOuterDoorState(matchRobotAirlock(rID), OPEN)) *$
$.outerRobot\_moveToLocation(rID, matchRobotAirlock(rID))$
$.robot\_pickUpWafer(rID, O\_AIRLOCK)]false$

(d) R1/R2 can only drop a wafer in A1/A2 if DO1/DO2 is open.
$[true*]\forall rID : robotID.val(rID == R1 || rID == R2) =>$
$[airlock\_setOuterDoorState(matchRobotAirlock(rID), CLOSED)$
$.(!airlock\_setOuterDoorState(matchRobotAirlock(rID), OPEN)) *$
$.outerRobot\_moveToLocation(rID, matchRobotAirlock(rID))$
$.robot\_dropWafer(rID, O\_AIRLOCK)]false$

(e) R3 can only pick up a wafer from A1/A2 if DI1/DI2 is open.
$[true*]\forall aID : airlockID.val(aID == A1 || aID == A2) =>$
$[airlock\_setInnerDoorState(aID, CLOSED)$
$.(!airlock\_setInnerDoorState(aID, OPEN)) *$
$.robot\_pickUpWafer(matchAirlockInnerRobotLocation(aID))]false$

(f) R3 can only drop a projected wafer in A1/A2 if DI1/DI2 is open.
$[true*]\forall aID : airlockID.val(aID == A1 || aID == A2) =>$
$[airlock\_setInnerDoorState(aID, CLOSED)$
$.(!airlock\_setInnerDoorState(aID, OPEN)) *$
$.robot\_dropWafer(matchAirlockInnerRobotLocation(aID))]false$

(g) After picking up a wafer from A1/A2, R3 can only drop it back to A1/A2 after the wafer was dropped to the lamp.
$[true*]\forall aID : airlockID.val(aID == A1 || aID == A2) =>$
$.robot\_pickUpWafer(matchAirlockInnerRobotLocation(aID))$
$.(!robot\_dropWafer(LAMP)) *$
$.robot\_dropWafer(matchAirlockInnerRobotLocation(aID))]false$

(h) After picking up a wafer from its corresponding input stack, a robot can

only drop it into its corresponding airlock.
$[true*]\forall rID : robotID.val(rID == R1||rID == R2) =>$
$[robot\_dropWafer(rID, OUT\_STACK)$
$.(!robot\_pickupWafer(rID, INP\_STACK))*$
$.outerRobot\_moveToLocation(rID, matchRobotAirlock(rID))$
$.robot\_dropWafer(rID, O\_AIRLOCK)]false$

(i) After picking up a wafer from its corresponding airlock, a robot can only drop it into its corresponding output stack.
$[true*]\forall rID : robotID.val(rID == R1||rID == R2) =>$
$[robot\_dropWafer(rID, O\_AIRLOCK)$
$.(!robot\_pickupWafer(rID, O\_AIRLOCK))*$
$.outerRobot\_moveToLocation(rID, matchRobotOutputStack(rID))$
$.robot\_dropWafer(rID, OUT\_STACK)]false$

(j) A wafer can only be placed into the output stack if it was projected by the lamp.
$[true*]\forall rID : robotID.val(rID == R1||rID == R2) =>$
$[robot\_pickUpWafer(rID, INP\_STACK)$
$.(!lamp\_projectWafer)*$
$.robot\_dropWafer(rID, OUT\_STACK)]false$

5.2 Translated liveness requirements

1. From the input stack, a wafer should always be able to reach the lamp.
$[true*]\forall rID : robotID.val(rID == R1||rID == R2) =>$
$[robot\_pickUpWafer(rID, INP\_STACK)] < true*.lamp\_projectWafer > true$

2. From the lamp, a wafer should always be able to reach the output stack.
$[true*]\forall rID : robotID.val(rID == R1||rID == R2) =>$
$[lamp\_projectWafer] < true * .robot\_dropWafer(rID, OUT\_STACK) > true$

3. The system must be deadlock-free.
$[true*] < true > true$

Chapter 6

System Modelling

The system has been modelled using controllers for most of the components. To avoid duplication of code, the two airlocks have the same controller code, and the two controllers are distinguished by means of airlock IDs.

Likewise, both output stacks' interactions, the input stacks' interactions, and the two outer robot controller interactions have the same code in each case, and are differentiated using output stack IDs, input stack IDs, and robot IDs, respectively.

Each controller has a set of states defined by variables as required to determine the uniqueness of the states. State information is exchanged between controllers by means of communication actions described in chapter 3, and state transitions are made accordingly, based on information received from other controllers.
Here, we describe in brief how the controllers are modelled, and their respective communication actions (the exact code can be found in the Appendix):

1. Outer Robot Controller: Makes the robot check the corresponding input stack's state and pick up a wafer if the stack is not empty. Communicates with the airlock controller to determine when to make the robot pick up or drop a wafer. If the robot has a wafer, the controller makes it check the output stack to drop a wafer if the stack is not full.

2. Inner Robot Controller: Communicates with the airlock to determine when to pick up or drop a wafer. Makes the robot check the lamp sensor to determine if the wafer is projected or not. Has non-determinism in its logic to make the robot visit one of the airlocks without any priority in case both happen to have unprojected wafers at the same time.

3. Airlock Controller: Communicates with the inner robot and the airlock's corresponding outer robot to determine which door to open/close. Ensures that the airlock never has both doors open at the same time.

Since there are lots of *IDs* involved, maps have been used to *match* the various IDs across controllers (for instance, when referring to interactions between robot 1 and airlock 1, the IDs will always be matched as $R1 \rightarrow A1$ or $A1 \rightarrow R1$.)

Moreover, as stated previously, interactions with the stacks and the lamp are treated as external interactions, and explicit stack and lamp controllers aren't created. We could model controllers for those components as well, but for the sake of keeping the system simple and operational with the existing controllers, we have treated the stacks and lamp as mere passive elements with which the existing controllers interact.

The `mcrl22lps` command is used to generate the lps for the mclr2 code, and is translated to an lts using the `lps2lts` command. Given the number of parallel components, it is difficult to get a good understanding of the system based on its LTS, and hence the `ltsview` command is used to visualize the system and its states.

The following figure is the visual representation of the system:

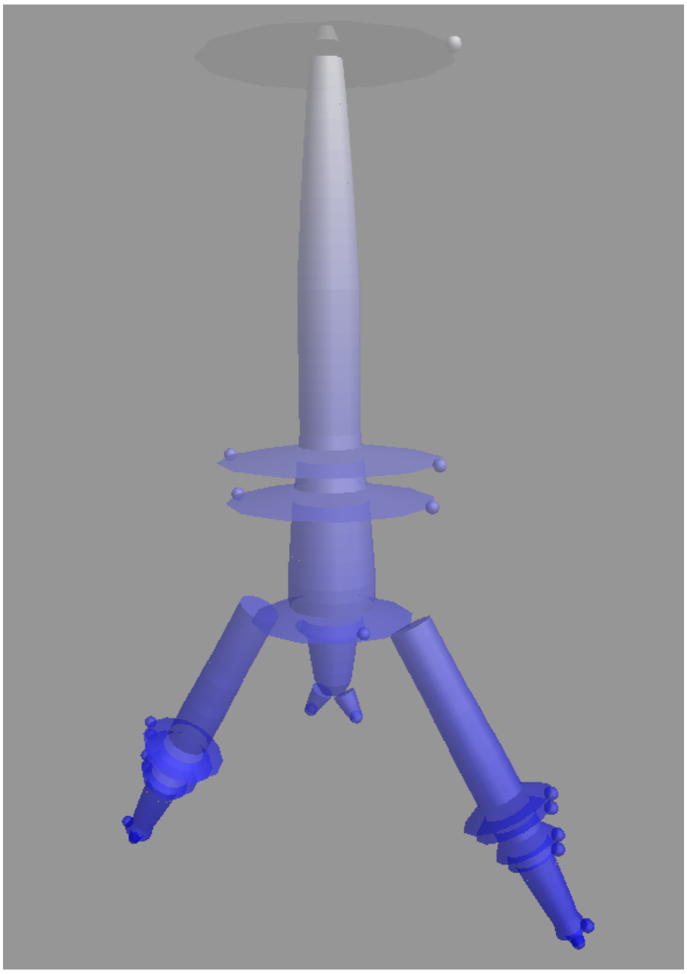

Figure 6.1: LTS visualization

Chapter 7

Verification

The system's LTS visualization can be checked for determining the absence of deadlocks, and the system can be verified using μ-calculus formulas to verify the plain text requirements stated previously. `mcf` files are used to verify the requirements translated to μ-calculus formulas.

7.1 Tools used for verification

The MCRL2 version used for performing the modelling and verification of the system is 201808.0.

A makefile has been created and used to make performing the verification tests easier. It is included in the zip archive and the accompanying readme file explains how it is to be used.
`mcrl22lps` is used for generating the lps file, and the `lps2lts` command is used for generating the corresponding lts file.

7.2 Verification process

Once the lps is generated, we call lps2pbes on it for every MCF (μ-calculus formula) file corresponding to each of the safety and liveness requirements.
Once we have the .pbes files, we call `pbes2bool` on each of them so as to get a boolean value representing whether the test condition holds or not. Ideally, in order to say that our requirements have been met, we would want each `pbes2bool` call to return a *true* value.

As expected, in our system, running the requirements tests through the MCF files returns a *true* in each case, thus proving that all the requirements are met.

We tried running a variant with no communications enabled between the components, and two of the liveness requirements fail by returning a false, as expected. However, one of the liveness requirements does not rely on communication between the components and hence it returned a true. But we can clearly see that all the liveness requirements were not met in that case.

7.3 Deadlock absence verification

The system's LTS is used to generate a 3D figure of the state-space using the `ltsview` command, and the figure can be used to verify the absence of deadlocks through a visual inspection (a "Mark deadlocks" option in the MCRL2 GUI shows deadlocks present in the system, if any, as red spheres). Moreover, the μ-calculus formula for showing that a system is deadlock-free is: $[true*]\langle true\rangle true$ - it should return a true if there are no deadlocks.

In our case, the visualization rendered does not show any deadlocks, as can be seen below:

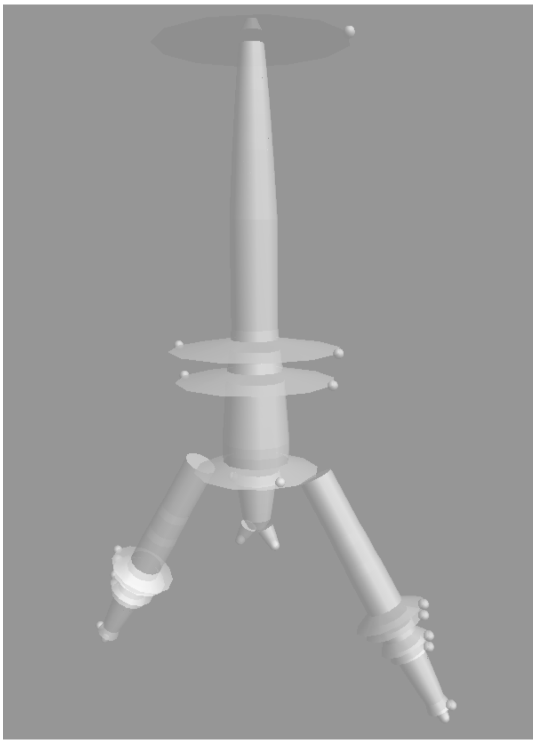

Figure 7.1: LTS view marking deadlocks

The μ-calculus formula returns true for our system, and hence we can say that our system is deadlock-free.

The LTS generated can be visualized in the form of a 3D shape using the `ltsview` command on the previously generated lts file.

Chapter 8

Conclusions

This project established that modelling and verifying a fully-functional system takes quite some effort and is far more complicated than it seems initially, when its idea is still in the informal stage (semantic domain). In a nutshell, we can conclude that it takes formal verification and modelling methods in order to say with certainty that the system is logically sound, as otherwise many corner cases may be missed, given the nature of logic itself.

- We started our project with an informal discussion stage, in which we discussed the system and its components from a logical perspective, listing down the expected behaviour and the possible corner cases which would need to be tackled in the implementation phase. This phase involved drawing state diagrams by hand and thinking of all relevant state transitions.
- Once we had the mental model ready, we started with the process modelling on paper, in simple steps so as to understand how to translate the model to MCRL2 code. At this stage, things seemed clear enough and hence we proceeded with writing the MCRL2 code for our model.
- However, once we had the code ready, generating the LTS revealed a lot of lapses in our original conception of the system's logic. There were a lot of possible cases we hadn't thought of, and that was shown by the many deadlocks that surfaced in the LTS. This step was repeated a few times, and finally we were able to generate a model that was deadlock-free.
- Once we had the final model ready, we translated the requirements from plain text to μ-calculus and verified the cases. This stage also revealed a few minor changes to be made to the MCRL2 code, and after that, we were able to verify the requirements correctly.

The main problem was the modelling part, since we had to think of the controllers from a parallel perspective as individual components interacting with the environment, rather than the system only as a sequence of steps to move wafers around. It was a good learning experience in formal methods for verification and modelling a system.

Chapter 9

Appendices

9.1 MCRL2 Code

```
% Robot-based wafer projection system model

% Though we have modelled the input and the output stacks, and the lamp as controllers here, we haven't used them in the interactions for simplicity's sake. We've treated their respective actions as external actions.



sort
 % Unique identifiers for system components that have more than one instance
 robotID      = struct R1 | R2 | R3;
 airlockID       = struct A1 | A2;
 stackID      = struct IS1 | IS2 | OS1 | OS2;
 innerRobotLocationID = struct I_A1 | I_A2 | LAMP | I_ND; % I_ND = Non-deterministic
 outerRobotLocationID = struct O_AIRLOCK | INP_STACK | OUT_STACK;

 % Types and states as required in communications
 airlockDoorType  = struct INNER | OUTER;
 robotActionType  = struct PICKUP | DROP | IDLE; % Maybe IDLE not needed
 waferSensorState  = struct WAFERPRESENT | NOWAFER;
 waferState    = struct PROJECTED | UNPROJECTED;
 doorState    = struct OPEN | CLOSED;
 lampState    = struct ON | OFF;
 inputStackState   = struct EMPTY | NEMPTY;
 outputStackState  = struct FULL | NFULL;
```

```
 % All maps are defined as "match<component1><component2> = <component1> to <component2>"
map

 % Airlock -> Outer Robot
 matchAirlockRobot: airlockID -> robotID;

 % Outer Robot ->  Airlock
 matchRobotAirlock: robotID -> airlockID;

 % Input Stack -> Outer Robot
 matchInputStackRobot: stackID -> robotID;

 % Outer Robot -> Input Stack
 matchRobotInputStack: robotID -> stackID;

 % OutputStack -> Robot
 matchOutputStackRobot:  stackID -> robotID;

 % Robot -> OutputStack
 matchRobotOutputStack:  robotID -> stackID;

 % Airlock -> Inner Robot Location
 matchAirlockInnerRobotLocation: airlockID -> innerRobotLocationID;

eqn

 % Airlock -> Robot
 matchAirlockRobot(A1) = R1;
 matchAirlockRobot(A2) = R2;

 % Robot ->  Airlock
 matchRobotAirlock(R1) = A1;
 matchRobotAirlock(R2) = A2;

 % Input Stack -> Robot
 matchInputStackRobot(IS1) = R1;
 matchInputStackRobot(IS2) = R2;

 % Robot -> Input Stack
 matchRobotInputStack(R1) = IS1;
 matchRobotInputStack(R2) = IS2;

 % Output Stack -> Robot
```

```
  matchOutputStackRobot(OS1) = R1;
  matchOutputStackRobot(OS2) = R2;

  % Robot -> OutputStack
  matchRobotOutputStack(R1) = OS1;
  matchRobotOutputStack(R2) = OS2;

  % Airlock -> InnerRobotLocation
  matchAirlockInnerRobotLocation(A1) = I_A1;
  matchAirlockInnerRobotLocation(A2) = I_A2;

act
  % External commands. Perhaps not needed.
  user_fillStack  : stackID;
  user_emptyStack : stackID;

  % External actions for the robot. Location is assumed as
     the robot's current location.
  robot_pickUpWafer  : robotID # outerRobotLocationID;
  robot_dropWafer    : robotID # outerRobotLocationID;

  robot_pickUpWafer  : innerRobotLocationID;
  robot_dropWafer    : innerRobotLocationID;

  robot_checkInputStackState  : stackID # inputStackState;
  robot_checkOutputStackState : stackID # outputStackState;

  outerRobot_moveToLocation : robotID # outerRobotLocationID
     ;
  outerRobot_moveToLocation : robotID # airlockID;
  outerRobot_moveToLocation : robotID # stackID;
  innerRobot_moveToLocation  : innerRobotLocationID;

  % External actions for the airlocks to open/close their
     doors.
  airlock_setInnerDoorState : airlockID # doorState;
  airlock_setOuterDoorState : airlockID # doorState;

  % External action for the lamp to project the wafer.
  lamp_projectWafer;

  % Communication actions; may or may not be hidden.

  % For the outer robots
  s_robotActionAck,
  r_robotActionAck,
```

```
  comm_robotActionAck : robotID # robotActionType #
      outerRobotLocationID;

  % For the inner robot
  s_robotActionAck,
  r_robotActionAck,
  comm_robotActionAck : robotActionType #
      innerRobotLocationID;

  s_inputStackState,
  r_inputStackState,
  comm_inputStackState : stackID # inputStackState;

  s_outputStackState,
  r_outputStackState,
  comm_outputStackState : stackID # outputStackState;

  s_airlockWaferProjectionState,
  r_airlockWaferProjectionState,
  comm_airlockWaferProjectionState : airlockID # waferState;

  s_airlockDoorState,
  r_airlockDoorState,
  comm_airlockDoorState : airlockID # airlockDoorType #
      doorState;

  % Sensors, as external actions. Perhaps not needed.
  sense_inputStack :  stackID # inputStackState;
  sense_outputStack :  stackID # outputStackState;
  sense_airlock  :  airlockID # Bool;
  sense_lamp   :  Bool  # waferState;

proc

  %%%%%%%%%%%%%%%%%%%       AIRLOCK CONTROLLER
      %%%%%%%%%%%%%%%%%%%%%%%


  Airlock (aID: airlockID, waferSensorState:
      waferSensorState, projectionState: waferState,
      innerDoorState: doorState, outerDoorState: doorState) =

  % If an outer robot drops a wafer in the airlock, change
      the state of the airlock
  (waferSensorState == NOWAFER && outerDoorState == OPEN) ->
       r_robotActionAck(matchAirlockRobot(aID), DROP,
      O_AIRLOCK) . airlock_setOuterDoorState(aID, CLOSED) .
```

```
Airlock (waferSensorState = WAFERPRESENT, projectionState = UNPROJECTED, outerDoorState = CLOSED)

% If there is an unprojected wafer in the airlock and the inner door is closed, open it
+ (waferSensorState == WAFERPRESENT && projectionState == UNPROJECTED && innerDoorState == CLOSED) -> airlock_setInnerDoorState(aID, OPEN) . Airlock(innerDoorState = OPEN)

% If robot R3 picks up the wafer from the airlock, change the state of the airlock
+ (waferSensorState == WAFERPRESENT && projectionState == UNPROJECTED && innerDoorState == OPEN) -> r_robotActionAck(PICKUP, matchAirlockInnerRobotLocation(aID)) . Airlock(waferSensorState = NOWAFER)

% If robot R3 drops a projected wafer in the airlock, change the state of the airlock
+ (waferSensorState == NOWAFER && projectionState == UNPROJECTED && innerDoorState == OPEN) -> r_robotActionAck(DROP, matchAirlockInnerRobotLocation(aID)) . airlock_setInnerDoorState(aID, CLOSED) . Airlock(waferSensorState = WAFERPRESENT, projectionState = PROJECTED, innerDoorState = CLOSED)

% If there is a projected wafer in the airlock, the inner door is closed, open it
+ (waferSensorState == WAFERPRESENT && projectionState == PROJECTED && innerDoorState == CLOSED) -> airlock_setOuterDoorState(aID, OPEN) . Airlock(outerDoorState = OPEN)

% If an outer robot picks up the projected wafer, change the wafer sensor and wafer projected states
+ (waferSensorState == WAFERPRESENT && projectionState == PROJECTED && outerDoorState == OPEN) -> r_robotActionAck(matchAirlockRobot(aID), PICKUP, O_AIRLOCK) . Airlock (waferSensorState = NOWAFER, projectionState = UNPROJECTED)

% Send the states of the airlock
% Wafer Projection state
+ s_airlockWaferProjectionState(aID, projectionState) . Airlock()

% Inner door state
```

```
+ s_airlockDoorState(aID, INNER, innerDoorState) . Airlock()

% Outer door state
+ s_airlockDoorState(aID, OUTER, outerDoorState) . Airlock();


%%%%%%%%%%%%%%%%%%%        OUTER ROBOT CONTROLLER %%%%%%%%%%%%%%%%%%%%%%


   OuterRobot(rID: robotID, hasWafer: Bool, location: outerRobotLocationID) =
   % If the robot has no wafer, location is the corresponding input stack, then check stack; if it is empty, do nothing
(hasWafer == false && location == INP_STACK) -> robot_checkInputStackState(matchRobotInputStack(rID), EMPTY) . user_fillStack(matchRobotInputStack(rID)) . OuterRobot()

% If the robot has no wafer, location is corresponding input stack and it's not empty, pick up a wafer
+ (hasWafer == false && location == INP_STACK) -> robot_checkInputStackState(matchRobotInputStack(rID), NEMPTY) . robot_pickUpWafer(rID, INP_STACK) . OuterRobot(hasWafer = true)

% If the robot has a wafer but the airlock outer door is closed, do nothing
+ (hasWafer == true && location == INP_STACK) -> r_airlockDoorState(matchRobotAirlock(rID), OUTER, CLOSED) . OuterRobot()

% If the robot has picked up a wafer, go to airlock if its outer door is open, and drop the wafer
+ (hasWafer == true && location == INP_STACK) -> r_airlockDoorState(matchRobotAirlock(rID), OUTER, OPEN) . outerRobot_moveToLocation(rID, matchRobotAirlock(rID)) . robot_dropWafer(rID, O_AIRLOCK) . s_robotActionAck(rID, DROP, O_AIRLOCK) . OuterRobot(hasWafer = false, location = O_AIRLOCK)

% If the robot is at the airlock, and the wafer state is unprojected, do nothing
```

```
+ (hasWafer == false && location == O_AIRLOCK) -> r_airlockWaferProjectionState(matchRobotAirlock(rID), UNPROJECTED) . OuterRobot()

% If the robot is at the airlock, the outer door is open, and the wafer is projected, pick it up
+ (hasWafer == false && location == O_AIRLOCK) -> r_airlockDoorState(matchRobotAirlock(rID), OUTER, OPEN) . r_airlockWaferProjectionState(matchRobotAirlock(rID), PROJECTED) . robot_pickUpWafer(rID, O_AIRLOCK) . s_robotActionAck(rID, PICKUP, O_AIRLOCK) . outerRobot_moveToLocation(rID, matchRobotOutputStack(rID)) . OuterRobot(hasWafer = true, location = OUT_STACK)

  % If the robot has a projected wafer, location is the corresponding output stack, then check stack; if it is not full, drop the wafer and move to the input stack
+ (hasWafer == true && location == OUT_STACK) -> robot_checkOutputStackState(matchRobotOutputStack(rID), NFULL) . robot_dropWafer(rID, OUT_STACK) . outerRobot_moveToLocation(rID, matchRobotInputStack(rID)) . OuterRobot(hasWafer = false, location = INP_STACK)

  % If the robot has a projected wafer, location is the corresponding output stack, then check stack; if it is full, do nothing
+ (hasWafer == true && location == OUT_STACK) -> robot_checkOutputStackState(matchRobotOutputStack(rID), FULL) . user_emptyStack(matchRobotOutputStack(rID)) . OuterRobot();


%%%%%%%%%%%%%%%%%%%       INNER ROBOT CONTROLLER %%%%%%%%%%%%%%%%%%%%%%%%


InnerRobot (location: innerRobotLocationID, airlockCycle: airlockID, hasWafer: Bool, waferState: waferState) =

% If the robot is idle and airlock 1's or airlock 2's inner door is closed, do nothing
(location == I_ND && hasWafer == false) -> r_airlockDoorState(A1, INNER, CLOSED) . InnerRobot()

```

```
+ (location == I_ND && hasWafer == false) -> r_airlockDoorState(A2, INNER, CLOSED) . InnerRobot()

% If the robot is idle and airlock 1 has an unprojected wafer, go to airlock 1
+ (location == I_ND && hasWafer == false) -> r_airlockDoorState(A1, INNER, OPEN) . r_airlockWaferProjectionState(A1, UNPROJECTED) . innerRobot_moveToLocation(I_A1) . InnerRobot(location = I_A1, airlockCycle = A1)

% If the robot is at airlock 1, pick up the wafer and move to the lamp
+ (location == I_A1 && hasWafer == false && airlockCycle == A1) -> robot_pickUpWafer(I_A1) . s_robotActionAck (PICKUP, I_A1) . innerRobot_moveToLocation(LAMP) . InnerRobot(location = LAMP, hasWafer = true, waferState = UNPROJECTED)

% If the robot is idle and airlock 2 has an unprojected wafer, go to airlock 2
+ (location == I_ND && hasWafer == false) -> r_airlockDoorState(A2, INNER, OPEN) . r_airlockWaferProjectionState(A2, UNPROJECTED) . innerRobot_moveToLocation(I_A2) . InnerRobot(location = I_A2, airlockCycle = A2)

% If the robot is at airlock 2, pick up the wafer and move to the lamp
+ (location == I_A2 && hasWafer == false && airlockCycle == A2) -> robot_pickUpWafer(I_A2) . s_robotActionAck (PICKUP, I_A2) . innerRobot_moveToLocation(LAMP) . InnerRobot(location = LAMP, hasWafer = true, waferState = UNPROJECTED)

% If the robot is at the lamp and has an unprojected wafer, drop it at the lamp
+ (location == LAMP && hasWafer == true && waferState == UNPROJECTED) -> robot_dropWafer(LAMP) . InnerRobot(hasWafer = false)

% If the robot is at the lamp and does not have a wafer, pick it up from the lamp after projection
+ (location == LAMP && hasWafer == false) -> lamp_projectWafer . robot_pickUpWafer(LAMP) . InnerRobot(hasWafer = true, waferState = PROJECTED)

```

```
% If the robot is at the lamp and has a projected wafer, move to the airlock
+ (location == LAMP  && waferState == PROJECTED && airlockCycle == A1) -> innerRobot_moveToLocation(I_A1) . InnerRobot(location = I_A1)

% If the robot is at A1, drop the wafer, sense the door as closed, and move to a neutral location
+ (location == I_A1  && hasWafer == true && waferState == PROJECTED) -> robot_dropWafer(I_A1) . s_robotActionAck(DROP, I_A1) . InnerRobot(location = I_ND, hasWafer = false)

% If the robot is at the lamp and has a projected wafer, move to the airlock
+ (location == LAMP  && waferState == PROJECTED && airlockCycle == A2) -> innerRobot_moveToLocation(I_A2) . InnerRobot(location = I_A2)

% If the robot is at A2, drop the wafer, sense the door as closed, and move to a neutral location
+ (location == I_A2  && hasWafer == true && waferState == PROJECTED) -> robot_dropWafer(I_A2) . s_robotActionAck(DROP, I_A2) . InnerRobot(location = I_ND, hasWafer = false);

init
 hide({comm_robotActionAck, comm_inputStackState, comm_outputStackState, comm_airlockWaferProjectionState, comm_airlockDoorState},

  allow({ user_fillStack , user_emptyStack ,
          robot_pickUpWafer , robot_dropWafer, robot_checkInputStackState, robot_checkOutputStackState, outerRobot_moveToLocation, innerRobot_moveToLocation,
    airlock_setInnerDoorState, airlock_setOuterDoorState,
    lamp_projectWafer,
    comm_robotActionAck, comm_airlockWaferProjectionState,
    comm_airlockDoorState},

  comm({ s_robotActionAck| r_robotActionAck -> comm_robotActionAck, s_inputStackState| r_inputStackState -> comm_inputStackState, s_outputStackState |r_outputStackState  ->
```

```
    comm_outputStackState , s_airlockWaferProjectionState
    |r_airlockWaferProjectionState  ->
    comm_airlockWaferProjectionState, s_airlockDoorState|
    r_airlockDoorState  -> comm_airlockDoorState},

  OuterRobot(R1, false, INP_STACK) || OuterRobot(R2,
     false, INP_STACK)  ||
  Airlock(A1, NOWAFER, UNPROJECTED, CLOSED, OPEN) ||
     Airlock(A2, NOWAFER, UNPROJECTED, CLOSED, OPEN)  ||
  InnerRobot(I_ND, A1, false, UNPROJECTED)
  )
 )
);
```

9.2 Requirements

```
Airlocks
1.a
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
 => [lamp_projectWafer]<true*
 . robot_dropWafer(rID, OUT_STACK)> true

1.b
[true*] forall aID : airlockID
. [airlock_setInnerDoorState(aID, OPEN) . (!
(airlock_setInnerDoorState(aID, CLOSED)))* .
 airlock_setOuterDoorState(aID, OPEN)] false

Lamp
2.a
[true*] [lamp_projectWafer . (!robot_pickUpWafer(LAMP))*
. lamp_projectWafer] false

Robots
3.a.i
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
 => [outerRobot_moveToLocation(rID,
matchRobotOutputStack(rID)) .
robot_checkOutputStackState(matchRobotOutputStack(rID),
FULL) . robot_dropWafer(rID, OUT_STACK)] false

3.a.ii
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
 => [outerRobot_moveToLocation(rID,
matchRobotOutputStack(rID)) .
```

```
(!robot_checkOutputStackState(matchRobotOutputStack(rID),
NFULL))* . robot_dropWafer(rID, OUT_STACK)] false

3.b.i
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
=> [outerRobot_moveToLocation(rID,
matchRobotInputStack(rID)) .
robot_checkInputStackState(matchRobotInputStack(rID), EMPTY) .
robot_pickUpWafer(rID, INP_STACK)] false

3.b.ii
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
=> [outerRobot_moveToLocation(rID,
matchRobotInputStack(rID)) .
(!robot_checkInputStackState(matchRobotInputStack(rID),
NEMPTY)) . robot_pickUpWafer(rID, INP_STACK)] false

3.c
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
=> [airlock_setOuterDoorState(matchRobotAirlock(rID), CLOSED)
. (!
airlock_setOuterDoorState(matchRobotAirlock(rID), OPEN))*
. outerRobot_moveToLocation(rID, matchRobotAirlock(rID)) .
robot_pickUpWafer(rID, O_AIRLOCK)] false

3.d
[true*] forall rID: robotID . val(rID == R1 || rID == R2) =>
[airlock_setOuterDoorState(matchRobotAirlock(rID), CLOSED) . (!
airlock_setOuterDoorState(matchRobotAirlock(rID), OPEN))*
. outerRobot_moveToLocation(rID, matchRobotAirlock(rID)) .
robot_dropWafer(rID, O_AIRLOCK)] false

3.e
[true*] forall aID: airlockID . val(aID == A1 || aID == A2)
=> [airlock_setInnerDoorState(aID, CLOSED) .
(!airlock_setInnerDoorState(aID, OPEN))* .
robot_pickUpWafer(matchAirlockInnerRobotLocation(aID))] false

3.f
[true*] forall aID: airlockID . val(aID == A1 || aID == A2)
=> [airlock_setInnerDoorState(aID, CLOSED) .
(!airlock_setInnerDoorState(aID, OPEN))* .
robot_dropWafer(matchAirlockInnerRobotLocation(aID))] false

3.g
[true*] forall aID: airlockID . val(aID == A1 || aID == A2) =>
[robot_pickUpWafer(matchAirlockInnerRobotLocation(aID)) .
```

```
!robot_dropWafer(LAMP)* .
robot_dropWafer(matchAirlockInnerRobotLocation(aID)) ] false

3.h
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
=> [robot_dropWafer(rID, OUT_STACK) .
!robot_pickUpWafer(rID, INP_STACK)* .
outerRobot_moveToLocation(rID, matchRobotAirlock(rID)) .
robot_dropWafer(rID, O_AIRLOCK)] false

3.i
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
=> [robot_dropWafer(rID, O_AIRLOCK) .
!robot_pickUpWafer(rID, O_AIRLOCK)* .
outerRobot_moveToLocation(rID, matchRobotOutputStack(rID)) .
robot_dropWafer(rID, OUT_STACK)] false

3.j
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
=> [robot_pickUpWafer(rID, INP_STACK) .
(!lamp_projectWafer)* . robot_dropWafer(rID, OUT_STACK)] false

Liveness Requirements
1
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
=> [robot_pickUpWafer(rID, INP_STACK)]
<true* . lamp_projectWafer> true

2
[true*] forall rID: robotID . val(rID == R1 || rID == R2)
=> [lamp_projectWafer]<true* .
robot_dropWafer(rID, OUT_STACK)> true

3
[true*]<true>true
```