ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING

Department of electrical, electronic, and information engineering
"Guglielmo Marconi" - DEI

MASTER'S DEGREE IN AUTOMATION ENGINEERING

MASTER'S THESIS
in
MECHATRONICS SYSTEMS MODELING AND CONTROL M

DEVELOPMENT OF GENERALIZED MOTION APPLICATIONS FOR LINEAR TRANSPORT SYSTEMS WITH INDIVIDUAL MOVERS

CANDIDATE
Gregorio Verri

SUPERVISOR
Prof. Alessandro Macchelli

CO-SUPERVISORS
Eng. Davide Azzolini
Eng. Fabio De Marco

Academic Year 2022-2023

Session I

Acknowledgments

I would like to thank my supervisor, Prof. Macchelli, who guided me through this project and helped me finalise it.

I'm extremely grateful to all the people at IMA Automation that helped me achieve this result: in particular, Mr. Azzolini for the opportunity and Mr. De Marco for the assistance.

I would also like to thank my parents and my whole family: without them, this would have not been possible.

I wish to offer my special thanks to Jessica, my never-ending source of inspiration and support.

Lastly, many thanks to all my fellow students for the years spent together.

Abstract

This thesis aims to produce two generalized motion applications for the integration of XTS systems into automatic machines. Given the lack of some software tools from the manufacturer, in the industrial setting of IMA Automation, the need for specific functionalities has arisen. In particular, different solutions to the rephasing problem are presented (restoring movers after being set in a torque-off state) and a generalized structure for the workstations on the track is proposed.

In this work, the applications are implemented in Function Blocks coded in ST according to IEC 61131-3. The proposed solutions are tested using the XTS Simulator included in the Beckhoff IDE TwinCAT 3. The rephasing tool is tested using virtual commissioning 3D models and is successful in every configuration analysed, while the generalized workstation is proven effective by rapidly building a simulation for an automatic machine layout in its very early development stages.

Keywords: *motion application, linear transport system, XTS, automatic machine, rephasing, workstation*

Contents

1 Development setup **13**
1.1 TwinCAT 3 13
1.2 IEC 61131-3 13
1.3 XTS simulation 14
1.3.1 System Overview 14
1.3.2 Simulation 14
1.4 Hardware 15

2 OMAC PackML **17**
2.1 Machine state types 17
2.2 State descriptions 18
2.3 Machine control modes 19

3 XTS rephasing tool **21**
3.1 Native control functions for XTS and collision avoidance implementation 21
3.2 Necessity for a rephasing tool: the deadlock condition 22
3.3 Assumptions 22
3.4 Proposed solutions without collision avoidance 24
3.4.1 Global restore rephasing mode 25
3.4.2 Priority restore rephasing mode 26
3.4.3 Implementation 27
3.5 Proposed solutions with collision avoidance 31
3.5.1 Global restore CA rephasing mode 32
3.5.2 Priority restore CA rephasing mode 34
3.5.3 Implementation 34
3.6 Testing 38
3.6.1 TwinCAT setup 38
3.6.2 Virtual Commissioning setup 39
3.6.3 Results 40

4 XTS workstation **41**
4.1 Preface . . . 41
4.2 Proposed solution . . . 42
4.2.1 Movers Adoption . . . 43
4.2.2 Movers Requests Handling . . . 47
4.2.3 Implementation . . . 47
4.3 Testing . . . 52
4.3.1 TwinCAT setup . . . 52
4.3.2 Results . . . 53

5 Conclusions **57**
5.1 Achievements . . . 57
5.2 Future developments . . . 57

List of Figures

1 IMA Automation logo 11
2 Beckhoff XTS 12

1.1 TwinCAT XTS simulator tool 15
1.2 Beckhoff IPC 16

2.1 PackML production mode FSM 20

3.1 Deadlock example 23
3.2 Deadlock example 25
3.3 Flowchart of Global Restore rephasing mode 26
3.4 Priority mode example 28
3.5 Flowchart of Priority Restore rephasing mode 29
3.6 Function block I/O interface. 30
3.7 Method example 32
3.8 Deadlock example 33
3.9 Deadlock Avoidance example 34
3.10 Flowchart of Global Restore CA rephasing mode 35
3.11 Flowchart of Priority Restore CA rephasing mode 36
3.12 Function block I/O interface. 37
3.13 XTS Viewer in the TwinCAT 3 XTS extension: test setup . . 38
3.14 Virtual Commissioning 3D model 39
3.15 Average rephasing times results 40

4.1 XTS workstation structure 42
4.2 Flowchart of Movers Adoption routine 44
4.3 Mover adoption example 45
4.4 Skipping station example 46
4.5 Flowchart of Movers Requests Handling algorithm 48
4.6 Function block I/O interface. 49
4.7 XTS Viewer in the TwinCAT 3 XTS extension: test setup . . 52
4.8 Cycle Time Trace 53

4.9 Trace of the numbers of movers in queue 55

Introduction

In recent years, automatic machines have seen the rise of new solutions regarding conveyance technologies: in many applications, the rigidity of conveyor belts and purely mechanical transfers has been replaced with the flexibility of linear transport systems based on individually actuated movers. In this kind of solutions, a modular closed (or open) track actuates a set of passive sliders along its path: they are completely independent one from the other. Unlike traditional permanent magnet linear synchronous motors (PMLSM), which consist of a passive linear guide and an active slider, these electric drives use passive permanent magnet movers and a segmented track composed of active modules to generate the required electromagnetic fields: this configuration frees the moving parts from any wired connection allowing periodic motion along the path and flexibility in the operation. The replacement of traditional conveyance systems brings many advantages and enables innovative design in the whole machine, such as software-based format change and dynamic grouping of product units. To this day, many commercial solutions exist that implement such technology in industrial settings.

This thesis has been carried out at IMA Automation, a segment of the IMA S.p.A. group, a world leader in the design and manufacturing of automatic machines, and the focus has been on XTS: a solution from Beckhoff, which also provided courses on their automation software TwinCAT 3 and relative XTS extension. The goal of the project has been to develop motion applications for XTS systems integrated into automatic machines while maintaining the most generalized approach possible.

A simulator for the XTS system is available on TwinCAT 3 XAE, the integrated development environment. It can be set up with an arbitrary con-

Figure 1: IMA Automation logo.

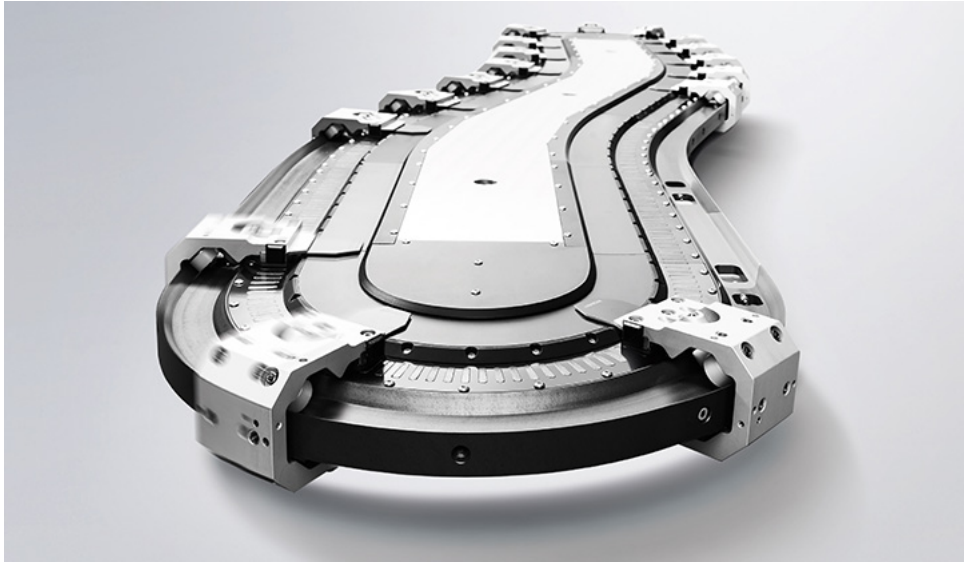


Figure 2: Beckhoff XTS. Picture reprinted from [2].

figuration of motor modules and movers: it has been used throughout the internship to test the developed code. A summary of the development setup will be presented In the first chapter, the OMAC PackML software design framework will be shown in the second one, while my suggested implementations for two generalized motion applications will be displayed in the third and fourth chapters. In the fifth and final chapter, brief conclusions will be drawn, alongside a discussion on possible future developments.

Chapter 1

Development setup

To understand the source code of an automatic machine, adequate training has been necessary on the development tools for the specific platform at hand: in particular, during the internship, the Beckhoff Automation systems have been studied. In the next few paragraphs, a brief overview of the IDE and the programming standards used by the manufacturer is presented, alongside the hardware used for testing.

1.1 TwinCAT 3

TwinCAT is a PC-based control system for industrial automation: it includes eXtended Automation Engineering (XAE), the development environment, and eXtended Automation Runtime (XAR), the Real-Time extension for Windows. XAE is integrated into the Visual Studio IDE and includes editors and compilers for the IEC 61131-3 standard languages. XAR, on the other hand, is a Real-Time extension for Windows OS which allows for preemptive scheduling, priority control and deterministic execution of tasks, as seen in [1]. The version of TwinCAT 3 used for this project has been 4024.40.

1.2 IEC 61131-3

The IEC 61131-3 is an open standard for Programmable Logic Controllers (PLCs) and defines, among other things, 5 different programming languages, both graphical and textual, as shown in [8]. In this project, the whole source code has been developed in Structured Text (ST).

1.3 XTS simulation

1.3.1 System Overview

As discussed in the introduction, the XTS system is a linear actuator composed of a segmented active stator (composable track) and several individually actuated passive movers along the same path. As seen in [2], from the programmer's point of view every mover is controlled as a "standard" independent servo axis thanks to the specific XTS TwinCAT extension. For every axis, the SoftDrive object, which represents its I/O (contains velocity and position control loops), is connected to the NC axis (controlled by PLC as usual). The connection to the movers' hardware, for each one of them, is managed by the XTS Processing Unit (XPU), responsible for real-time communication. The XTS Driver is, for every control cycle and every mover, in charge of:

- collection of all position sensor signals,
- calculating the absolute position,
- calculating the velocity,
- position control,
- velocity control,
- phase transformation,
- setting phase current values to motor modules (while handling boundaries smoothly).

For performance reasons the whole control cycle needs to be completed in $250\mu s$, therefore the XTS task on the PLC needs to run with a cycle time of $250\mu s$. This is, for real systems with a lot of movers, quite a challenging deadline to meet: especially the calculation of phase currents of all involved motor modules and the compensation of boundary effects (for mover transitions from one motor to the next). Since on TwinCAT 3, there is support for multi-core CPUs, in [2] is recommended to run the XTS task alone on an isolated core (unavailable to Windows OS or any other PLC task).

1.3.2 Simulation

Given the system architecture description, it's clear that the simulator is implemented in the XPU of the XTS Drive: instead of communicating with

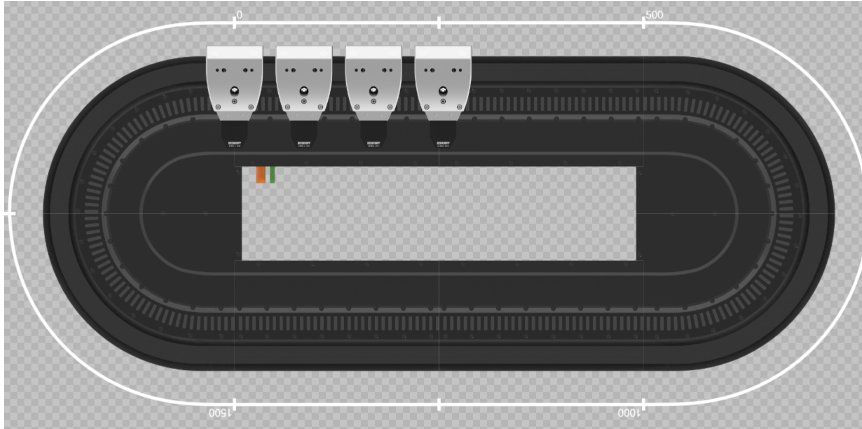


Figure 1.1: TwinCAT XTS simulator tool screenshot. This simple loop is 2 meters long with two 500mm straight segments and two 500m 180° curves. On the outside of the track is indicated the reference system: the top left mover is in position 0 (straight-curve junction) and the positive direction is clockwise. In this example, 4 movers are simulated.

some Hardware, the XPU simulates the feedback and doesn't need to write current phase outputs. This mode is much less computationally demanding than controlling a real XTS system so it has been possible to run it on a basic compact IPC. In figure 1.1 is shown an example with 4 movers built in the simulator tool.

1.4 Hardware

A compact Industrial PC (IPC) from Beckhoff has been used to test the developed applications and simulate the XTS system: it can be seen in figure 1.2. The PC is equipped with a quad-core Intel CPU @3.4GHz, 8GB of memory and Windows 10 64bit. This machine is not a very powerful configuration and wouldn't be recommended to control an XTS system but, since in this case only the simulator is being used, its performance is more than enough.

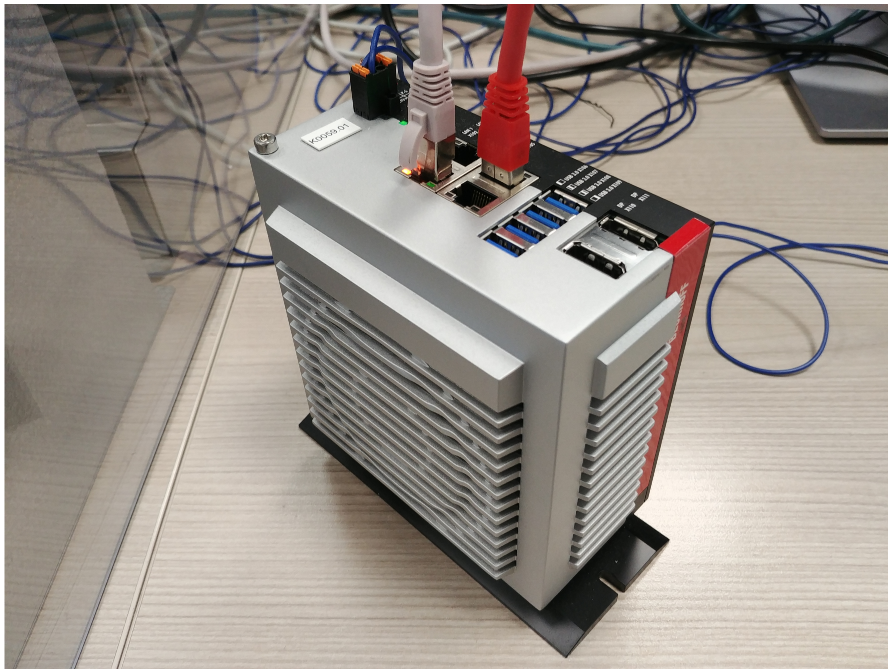


Figure 1.2: Beckhoff IPC model C6030-0060.

Chapter 2

OMAC PackML

To test the proposed solutions on a real application, it has been necessary to implement the software into the source code of an existing automatic machine developed at IMA Automation. They have adopted, as a company, the OMAC Packaging Machine Language (PackML) as a standard to develop software for automated machines and, to modify their existing project, it's been necessary to understand the design framework of the overall source code.

In documents such as [7] is delineated how PackML is made to provide, among other things:

- a definition of machine state types,
- state descriptions,
- state transitions,
- a definition of machine control modes.

The overall behaviour of automatic machines following this standard is delineated to fit into the framework determined by some predetermined Finite State Machines (FSMs) where states, transitions and operating modes are systematized.

2.1 Machine state types

The machine states are meant to completely define the current condition of the machine and in [7] are established two types:

Acting state A state which represents some processing activity. It implies the single or repeated execution of processing steps in a logical order,

for a finite time or until a specific condition has been reached. In [6] these are referred to as transient states, those states ending in “ING”.

Wait state A state used to identify that a machine has achieved a defined set of conditions. In such a state, the machine is maintaining a status until transitioning to an acting state. In [6] this was referred to as a “final” or “quiescent” state.

2.2 State descriptions

There are a fixed number of states defined in the base state model and in [7] is shown a sample enumerated set of possible machine states, which includes:

EXECUTE (Type: Acting) Once the machine is processing materials it is in the EXECUTE state. Different machine modes will result in specific types of EXECUTE activities. For example, if the machine is in the “Production” mode, the EXECUTE will result in products being produced, while in the “Clean Out” mode the EXECUTE state refers to the action of cleaning the machine.

ABORTING (Type: Acting) The ABORTING state can be entered at any time in response to the ABORT command or on the occurrence of a machine fault. The aborting logic will bring the machine to a rapid safe stop.

ABORTED (Type: Wait) The machine maintains status information relevant to the ABORT condition. The machine can only exit the ABORTED state after an explicit CLEAR command, subsequently to manual intervention to correct and reset the detected machine faults.

CLEARING (Type: Acting) Initiated by a state command to clear faults that may have occurred when ABORTING, and are present in the ABORTED state before proceeding to a STOPPED state.

STOPPING (Type: Acting) This state is entered in response to a STOP command. While in this state the machine executes the logic which brings it to a controlled stop as reflected by the STOPPED state. Normal STARTING of the machine cannot be initiated unless RESETTING had taken place.

STOPPED (Type: Wait) The machine is powered and stationary after completing the STOPPING state. All communications with other sys-

tems are functioning (if applicable). A RESET command will cause an exit from STOPPED to the RESETTING state.

RESETTING (Type: Acting) This state is the result of a RESET command from the STOPPED or COMPLETE state. Faults and stop causes are reset. RESETTING will typically cause safety devices to be energized and place the machine in the IDLE state where it will wait for a START command. No hazardous motion should happen in this state.

IDLE (Type: Wait) This is the state which indicates that RESETTING is complete. The machine will maintain the conditions which were achieved during the RESETTING state, and perform operations required when the machine is in IDLE.

STARTING (Type: Acting) The machine completes the steps needed to start. This state is entered as a result of a STARTING command (local or remote). Following this command, the machine will begin to "execute".

2.3 Machine control modes

As presented in [7], a machine control mode determines the subset of states, state commands, and state transitions that determine the strategy for carrying out a machine's process. Typical machine control modes are production, maintenance, manual etc. The distinguishing elements between these unit control modes are the selected subset of states, state commands, and state transitions. For example, the production mode can be realized by implementing all defined states, commands and transition into a *Base state model*: state models completely define the behaviour of a machine in one control mode. Its FSM is represented in figure 2.1.

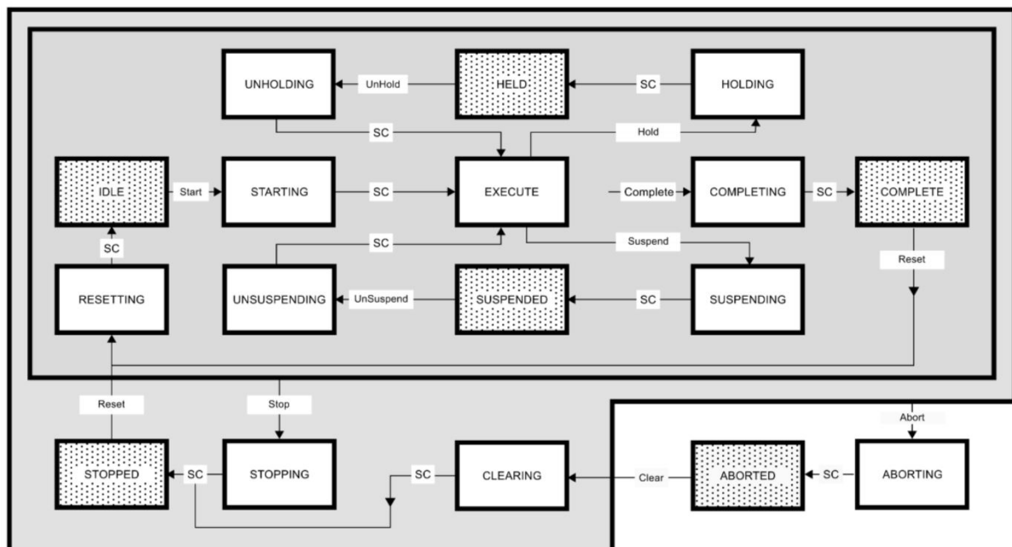


Figure 2.1: PackML production mode FSM example reprinted from [7].

Chapter 3

XTS rephasing tool

In this chapter, the design of generalized Function Blocks able to handle the rephasing process of all XTS movers after being set in a torque-off state will be presented.

3.1 Native control functions for XTS and collision avoidance implementation

Beckhoff Automation provides, alongside the XTS system, a set of software tools designed to handle typical XTS requirements. These functionalities are contained in the following TwinCAT libraries (as shown in [2] and [5]):

Tc3_McCoordinatedMotion Includes control function blocks for Axis Group handling [4]. The Axis Group object contains all movers within one loop and is required for the next library.

Tc3_McCollisionAvoidance Comprises control function blocks for Collision Avoidance (CA hereafter) [3].

Every XTS mover (as stated previously) is linked to an NC axis and, consequently, can be controlled independently using the standard PLCopen Function Blocks such as MC_MoveRelative, MC_MoveAbsolute etc. Nevertheless, it is possible to set up an *Axes group* containing all movers in the loop and a *Collision avoidance group* object: by controlling the movers motion with Function Blocks from the specific collision avoidance library, it's possible to ensure that the difference between their position setpoints will never drop below a predetermined *Gap*, regardless of the requested motion. For example, if a set of movers within the collision avoidance group is sent to a specific position along the track, only one of them will reach the target position while

all the others will stop one “gap length” from one another. All the Function Block instances utilized (of those unable to reach the target) will remain *Busy* as long as the target position isn't reached: in the previous case, only the one mover able to get to its goal will output *Done*. Once the shared target position is freed from the mover occupying it, the closest one will take its place and all the others will move accordingly, maintaining the minimum gap. Besides avoiding unwanted collisions, this mechanism is very convenient to handle movers queues within the loop, since they need only one position to be specified and the collision avoidance group takes care of the movers advancement while the queues are depleted.

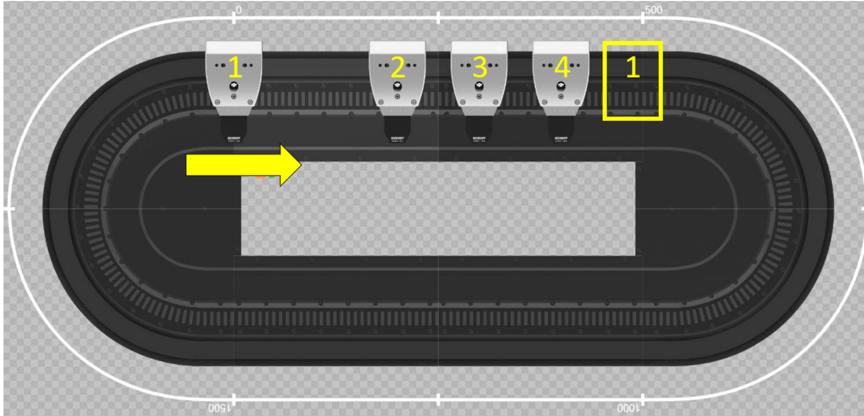
3.2 Necessity for a rephasing tool: the deadlock condition

The functionalities just presented, as useful as they are, fall short of assisting in the *rephasing process*: the problem consists in restoring the position of all movers in a loop to the last configuration they had before being set in a torque-off state. Once no forces are acting on the movers they may move, for example, due to gravity or due to an operator's intervention: it's not granted that the position found once the axes are re-enabled is the same as it was before. What's given is the set of positions to restore and the current one, while needs to be elaborated the sequence of movement commands required to obtain the latter from the former.

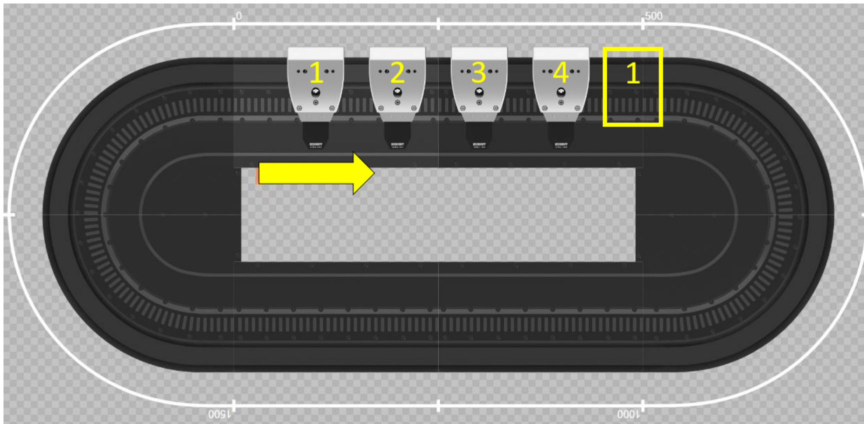
If, for example, collision avoidance is used, simply sending all movers to their target position isn't a solution because they can cause a *deadlock*. Analogously to the deadlock phenomenon studied in computer science, here a *deadlock* is defined as the condition in which a mover needs to reach a point beyond another one but isn't able to proceed: collision avoidance will stop it before it crashes but will not be useful in making it reach its goal. This can happen, for example, for a single mover trying to move past another (as in figure 3.1) or could happen between two movers going in opposite directions one towards the other.

3.3 Assumptions

In this work is considered only the case of a single static XTS loop without any track segments moved during operation: even if manufacturers of these kinds of transport systems (Beckhoff included) allow for track changes, the simplest case of one uninterrupted loop remains a very relevant simplifica-



(a) Mover 1 is trying to reach the target position indicated by the yellow rectangle, moving in positive direction (clockwise, indicated by the arrow).



(b) Unable to move past 2, Collision Avoidance is stopping 1 just before it. The system is standstill due to a *deadlock*.

Figure 3.1: Deadlock example in the XTS simulator tool.

tion. In all projects developed at IMA Automation, they have deployed this fixed structure. This layout has allowed some definitions to be stated and assumptions to be made (independent of the shape and length of the system at hand):

1. A set of movers positions will be called *configuration*.
2. Since the track is a closed loop that doesn't allow movers to go past each other, every mover will always be followed by the same other one (clearly, this also means that every mover will always follow the same other one). Consequently, regardless if they are numbered in ascending or descending order, they will always appear sorted by ID number.

3. Two mover configurations in which is possible to obtain one from the other with a finite sequence of physically possible movements (see point 2, e.g. movers can't go past each other) will be called *homologous configurations*.

4. As a consequence, all configurations found in an XTS loop during operation are *homologous configurations*. Hence, it is always possible to solve the rephasing problem. This can also be proven by considering that every movement in the torque-off state can be reversed with a motion command, ideally, and if the reverse sequence is performed, the original state can be restored.

3.4 Proposed solutions without collision avoidance

In this thesis have been produced a few different solutions for the rephasing process: algorithms have been developed able to generate a finite sequence of movement commands to bring any configuration of movers into any other homologous. They have been implemented in Function Blocks for PLCs. The scope has been to generate the most general solutions possible, given the assumptions made in section 3.3. As previously stated: the number of movers, length of the loop and track layout are irrelevant.

In the following sections, four different algorithms and implementations will be presented: the first two have been designed to handle the rephasing process without making use of the collision avoidance tools, therefore only one mover is being moved at any time to avoid crashes, while the last two exploit CA functions. Even if the first ones may seem inefficient in the approach (and for a large number of movers, they are) the focus has been, in these first attempts, to develop a rock-solid process on which to build up in the future. Using the collision avoidance tools allows for many movers to advance at the same time, but also makes the rephasing procedure much more delicate to handle: it's not trivial to avoid deadlocks by moving many axes all at the same time in different directions.

From now on, for every mover, the position that needs to be restored will be referred to as *target position*, while the *current* or *actual position* will refer to the starting point of the rephasing process.

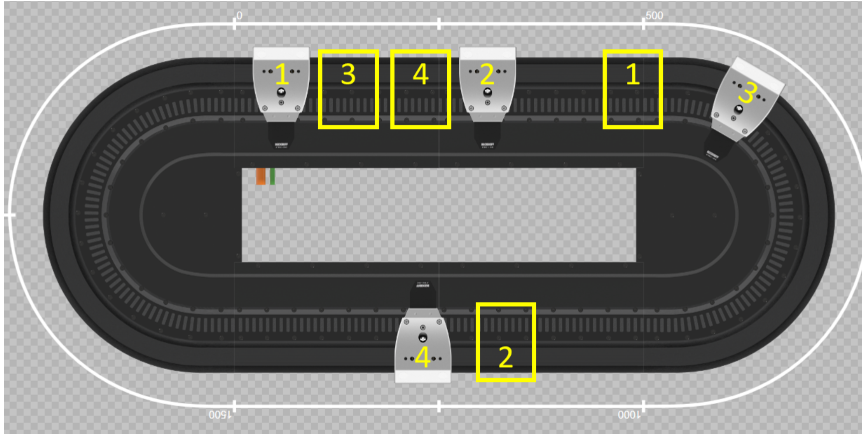


Figure 3.2: Deadlock example. If the movers need to reach the respective target positions (indicated with numbered yellow boxes) it's clear that no one of them has a free path to the destination. However, it's possible to avoid the deadlock and reach the target configuration by advancing all of them little by little: this will require more than 4 direct movements but it's doable.

3.4.1 Global restore rephasing mode

The first attempt at developing a rephasing procedure has been carried out by considering how to move from an arbitrary configuration to a homologous one: it has been called *global restore* mode because, in a real application, the goal would be to restore the position of every single mover to where it was before being put in the torque-off state. Since usually a homing procedure isn't available, because half-processed products in a production line need to return to the correct positions, a first, intuitive, approach could be to look for movers that can reach their target without any obstacles: by moving them one at a time and updating the free paths every time, this logic can solve many configurations. However, the movers can be in a situation in which there is no one able to reach its target right away: for example consider the case in figure 3.2. To avoid the deadlock, an alternative procedure is implemented if there is no mover with a free path to its target: a single direction is decided for the rephasing procedure by looking for the least overall distance needed to be travelled by all movers, and then the mover able to travel the furthest in that direction is sent as far as it can go. This simple algorithm can reach any configuration homologous to the starting condition and in figure 3.3 it's illustrated its flowchart.

Even if, in practical cases, this mode's performance could be satisfying due to the limited number of movers to rephase, the number of individual commands needed to "untangle" some configurations grows very fast with

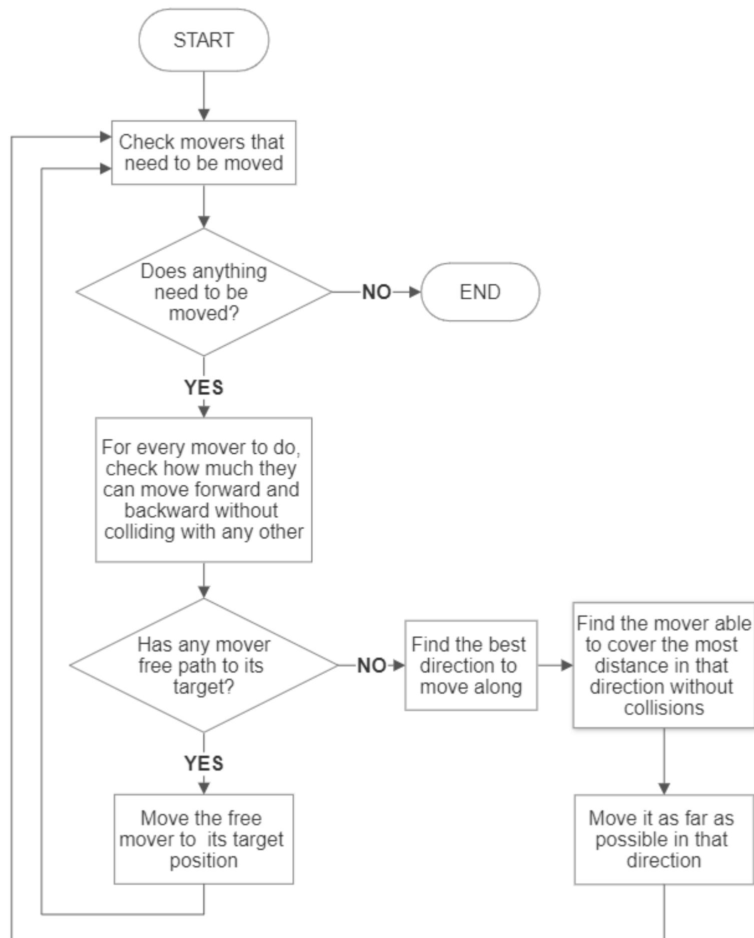


Figure 3.3: Flowchart illustrating the algorithm for Global Restore rephasing mode.

respect to the number of movers. Effectively, if a lot of operations are needed to obtain a free path for any mover, it takes a long time to complete the rephasing process.

3.4.2 Priority restore rephasing mode

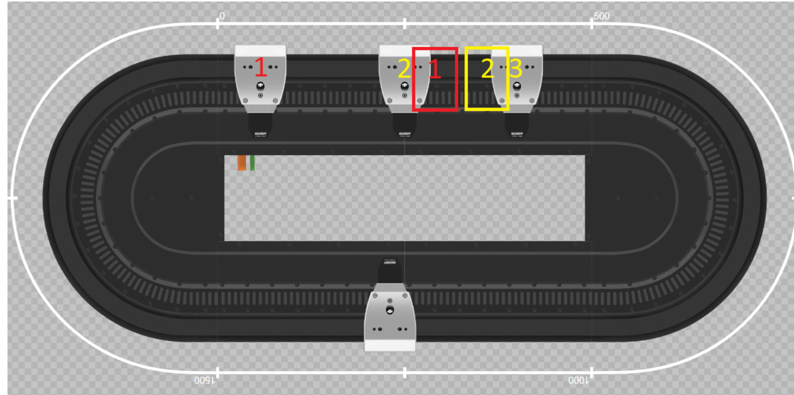
To optimize the rephasing process, an observation on practical cases has been made: usually, the XTS loop (or analogous system) is used as a transfer between workstations in a production line and during operations not all movers are populated with product units. The rephasing logic could take into account that only semi-finished products need to return to their original position to proceed into the next steps of the production line: all empty

movers don't need to go back to specific positions since they will be sent again to queue position.

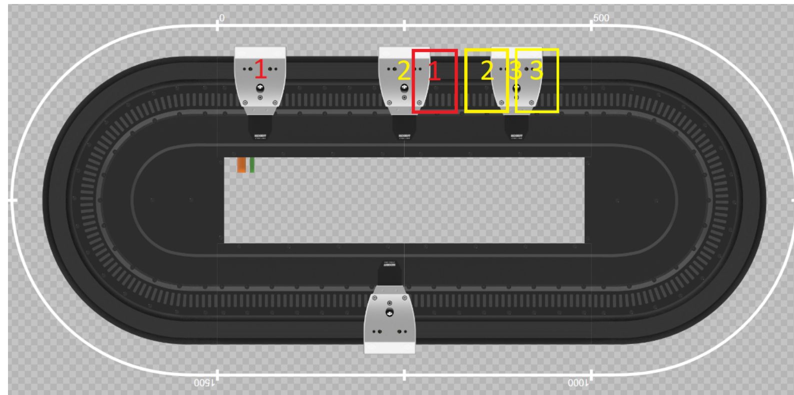
The *priority restore rephasing mode* works by defining some movers to be *prioritary* (for example with products on board, but any logic can be applied) and restoring only their positions. In case any prioritary mover is blocked on its path by other ones (prioritary or not), the movers in the way need to be sent beyond the priority target position to make space and allow the priority rephasing. Of course, it could be the case that all the priority movers have a free path to their restoring position, but in general this isn't true and the obstructions need to be managed properly: anytime there is no priority mover able to directly reach its target, the algorithm picks the one closest to its goal. A rephasing direction is determined for the procedure: it's based on the shortest path of the first mover at hand. Contrary to the *global restore*, it has been observed in simulation that the optimal direction for starting rephasing often isn't the shortest overall since, in this case, deadlock avoidance could require inverting the direction at any point of the procedure. For this reason, the first mover determines the direction to begin with. Once the priority mover is picked and the direction is set, which movers need to get out of its way must be evaluated: this doesn't include only the ones standing between it and its target, but also every other one past the target that needs to make room for them. An example of this case is shown in figure 3.4. The generation of target positions, to make space for the priority one, must be made in such a way that the movers are placed as close as possible: this is necessary to ensure that any configuration can be reached by all priority movers. If, by any chance, between the ones to get out of the way, there is a priority mover which is already in its target position, the direction must be inverted: otherwise, this will generate an endless sequence of movement commands. Once the priority has been free to reach its target, another one is handled until there are no more. This algorithm produces, on average, many fewer movement commands than the global restore and in figure 3.5 is illustrated its flowchart.

3.4.3 Implementation

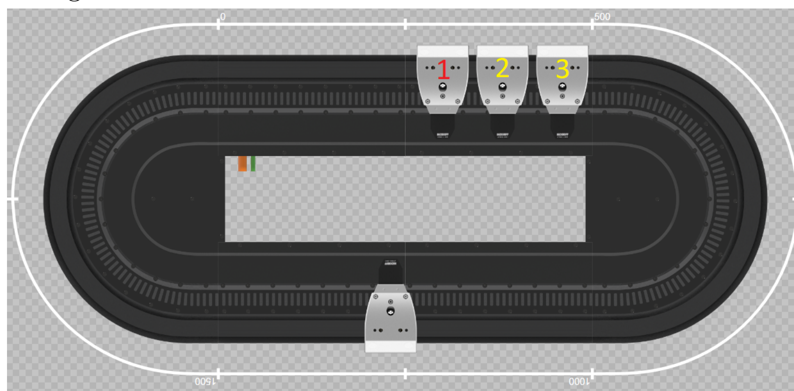
The previous solutions have been implemented in a Function Block coded in ST according to IEC 61131-3 standard: its I/O interface is shown in figure 3.6. The instance of the FB needs to communicate with the main program (PRG): after the initial call, in which all input parameters are loaded, with a rising edge of the `I_Enable` input the FB activates. When the `Q_Ready` output is `TRUE`, the block has calculated a mover index to be moved and how much it should travel in which direction. Once the mover has completed its



(a) Mover 1 is the only prioritary in this example: it needs to reach its target (red rectangle). Considering the shortest path, it's obstructed by mover 2. Therefore, a position target is generated (yellow) for mover 2 to get out of the way of 1.



(b) Since now also mover 2 is obstructed on its way by mover 3, a position target for 3 must be generated.



(c) In the end, for mover 1 to reach the target position, both movers 2 and 3 needed to make room for it.

Figure 3.4: Priority mode example in the XTS simulator tool.

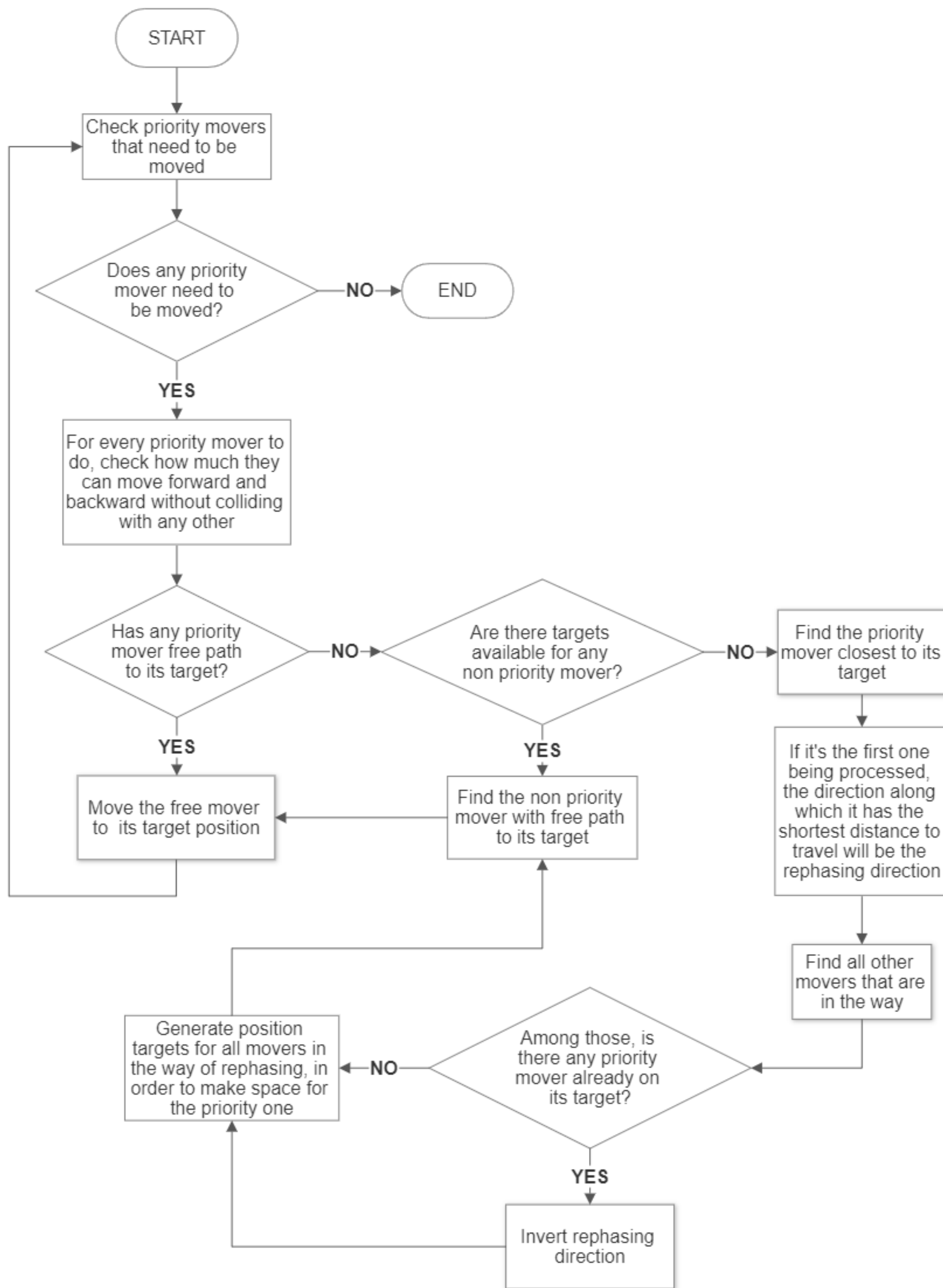


Figure 3.5: Flowchart illustrating the algorithm for Priority Restore rephasing mode.

Figure 3.6: Function block I/O interface.

task, a rising edge of `I_NextMover` will iterate the block again and produce the next movement command at the output. This process needs to be repeated until the `Q_Done` flag comes `TRUE`. Here follows a detailed description of the I/O interface: variables with the prefix `I_` are inputs, while `Q_` is for outputs.

I_Enable (BOOL) A rising edge of this input will activate the function block and output the first movement command.

I_NumberOfMovers (INT) The number of movers in the loop is a parameter essential to know how many axes to look for.

I_TrackLength (LREAL) Parameter needed to compute the modulo position at every step in which a calculation is involved. Since the system at hand produces periodic motion of the axis, the period is essential information to evaluate distances.

I_MoverPosition (ARRAY OF LREAL) This array contains the position of all the movers before the rephasing process. It's the starting configuration.

I_MoverTarget (ARRAY OF LREAL) This array contains all the positions of the movers that need to be restored. It's the objective to achieve.

I_ST_MoverWithPriority (ARRAY OF ST_MoverWithPriority) If the Priority mode is used, here are indicated which movers are prioritary.

I_MoverLength (LREAL) Mover size is necessary to avoid crashes.

I_NextMover (BOOL) A rising edge of this input, after an output is produced, will generate the next movement command.

I_OperatingMode (et_RestoreOperatingMode) This input allows to select the desired operating mode.

I_RestoreGap (LREAL) Tolerance to consider movers to be rephased. If the mover target is inside ± `I_RestoreGap` from the position, then doesn't need to be rephased.

I_EncoderGap (LREAL) Tolerance needed to correctly detect movers and avoid crashes when very close. `I_RestoreGap` ≥ `I_EncoderGap` must hold.

Q_MoverSetpointRelative (LREAL) When `Q_Ready` is `TRUE`, this is the signed distance that the specified mover needs to travel.

Q_MoverNumber (INT) When `Q_Ready` is `TRUE`, this is the mover to actuate.

Q_Ready (BOOL) Output ready if `TRUE`.

Q_Done (BOOL) Rephasing completed if `TRUE`.

Q_Error (BOOL) Debugging flag used to check for errors. If `TRUE` an input parameter is incorrect.

Q_ErrorID (STRING) If `Q_Error` is `TRUE`, this is the error description.

The internal structure of the Function Block is based on the two algorithms previously illustrated (figures 3.3 and 3.5) and the main logic is governed by a Finite State Machine (FSM). Several methods have been added to handle common tasks, such as checking the free space ahead and behind a selected mover: the interface of this specific one is represented in figure 3.7 as an example. The two modes (global and priority) are implemented in different states of the FSM but share the initialization, the output writing and a lot of methods.

3.5 Proposed solutions with collision avoidance

Starting with what has been built so far, another two modes have been designed: *Global restore Collision Avoidance* and *Priority restore Collision*

Figure 3.7: Method created to evaluate the distance between two movers (in both directions) that can be travelled before collision. Positive and negative directions are defined on the reference system centred in ActualPosition.

avoidance. The main goal has been to optimize the rephasing times obtainable with the non-CA versions by moving many (all) movers at the same time. As previously explained, giving the embedded CA functions the task of preventing crashes doesn't free the programmer from the necessity of checking the path of every mover: deadlocks are a constant threat during operations.

The main structural difference from this implementation to the previous one will be that, instead of generating single movement commands to be executed in sequence, the output of these algorithms will be a set of target positions and a set of rephasing directions to be executed all at the same time. In practice, every mover will get a target to reach and a direction to follow.

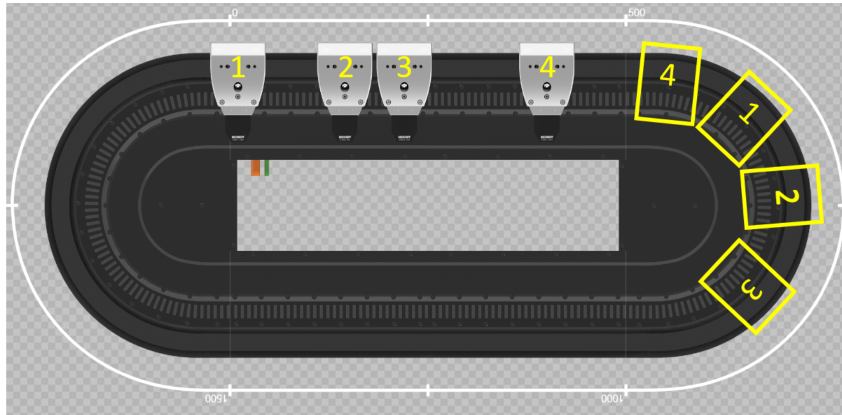
3.5.1 Global restore CA rephasing mode

Like for the homonymous non-CA modality, the goal is to restore the position of every mover to the configuration previous to the torque-off state: the algorithm is simpler overall (with respect to the first solution, shown in 3.4.1), but includes a necessary *deadlock avoidance* routine to function properly. After checking which movers need to be displaced, the rephasing direction is evaluated by looking for the shortest path overall. However, simply moving every axis in the same direction isn't enough to avoid deadlocks, as can be seen in the example in figure 3.8.

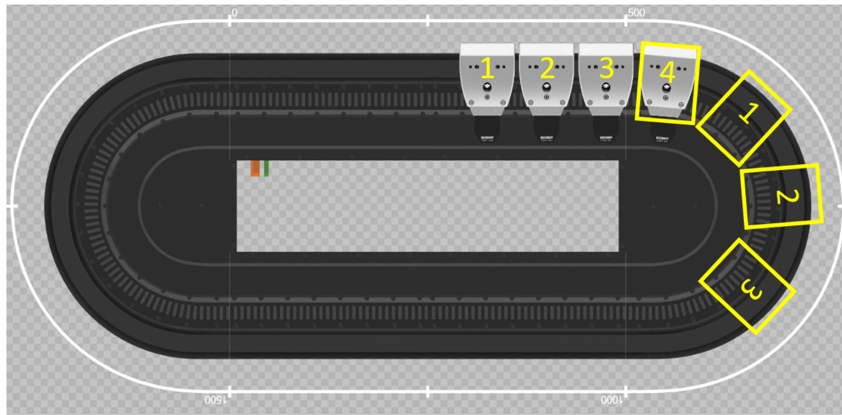
The *deadlock avoidance* works by checking three conditions for every mover and, if they are all satisfied, it means that a deadlock is found and the rephasing direction must be reversed for that case. Given a mover, its target position and rephasing direction, its *path* is defined as the track portion it's supposed to cross to reach the goal. The algorithm checks if:

1. there is any other mover along its path (will be referred to as interfering mover)

2. the target of an interfering mover is on the same path



(a) If every mover needs to reach the corresponding target (yellow rectangle), the shortest path overall, in this example, is along the positive direction (clockwise).



(b) However, since no one can move past mover 4, which has reached its position, the system is standstill due to a *deadlock*.

Figure 3.8: Deadlock example in the XTS simulator tool.

3. the path of the interfering mover does not contain the given mover

If all these conditions are verified, the direction of the movement must be reversed to avoid a deadlock. An example of an application can be seen in figure 3.9.

Once the set of directions is determined, all the movers can be safely sent to their target by using the CA functions. This solution is much faster than the ones without collision avoidance because all the movers can advance towards their goal simultaneously: the flowchart in figure 3.10 represents the overall behaviour.

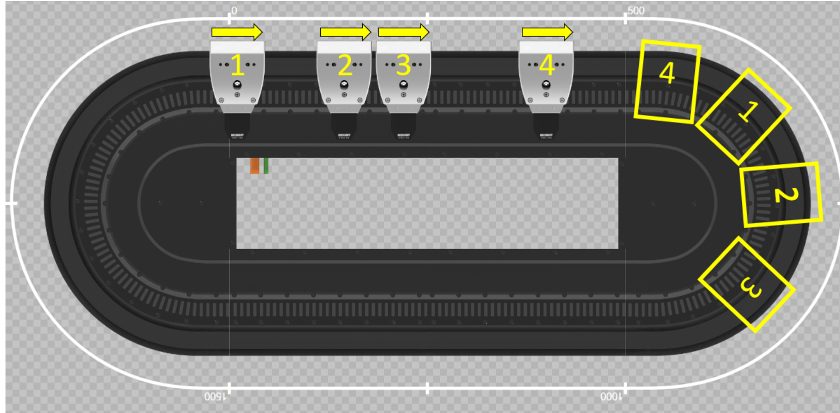


Figure 3.9: Deadlock avoidance example. If we consider the situation illustrated in figure 3.8a and apply the algorithm to mover 1, we can see that mover 4 and its target are on its path, while mover 1 is not on the path of mover 4. A deadlock is detected, as was the case in figure 3.8b, and mover 1 direction must be reversed. The same result is obtained on mover 2 and 3.

3.5.2 Priority restore CA rephasing mode

Analogously to the *Global restore mode*, also for the *Priority mode* has been coded a CA version. However, this time, the optimization with respect to *Global restore CA mode* hasn't been so drastic as the non-CA case, since moving every axis at the same time doesn't allow for much improvement. Nevertheless, in many cases could be beneficial to reduce the number of movers to rephase to the minimum necessary, as explained in section 3.4.2.

The overall goal of this modality is the same as the non-CA version but, as in the case of Global CA mode, the result will be a set of target positions and directions used to move all the necessary axes at the same time.

The algorithm begins as the one in 3.5.1 but it's only applied to the priority movers. Once that phase is completed, the logic generates targets for movers to get out of the way as was the case in 3.4.2. This time, however, there is no need to check for deadlocks since it has already been determined at the beginning of the process. The flowchart in figure 3.11 explains the proposed algorithm.

3.5.3 Implementation

These last two designs have been implemented in a Function Block built as the one shown in 3.4.3 but, as already stated, its output and communication with the main program are modified: the new I/O interface is shown in

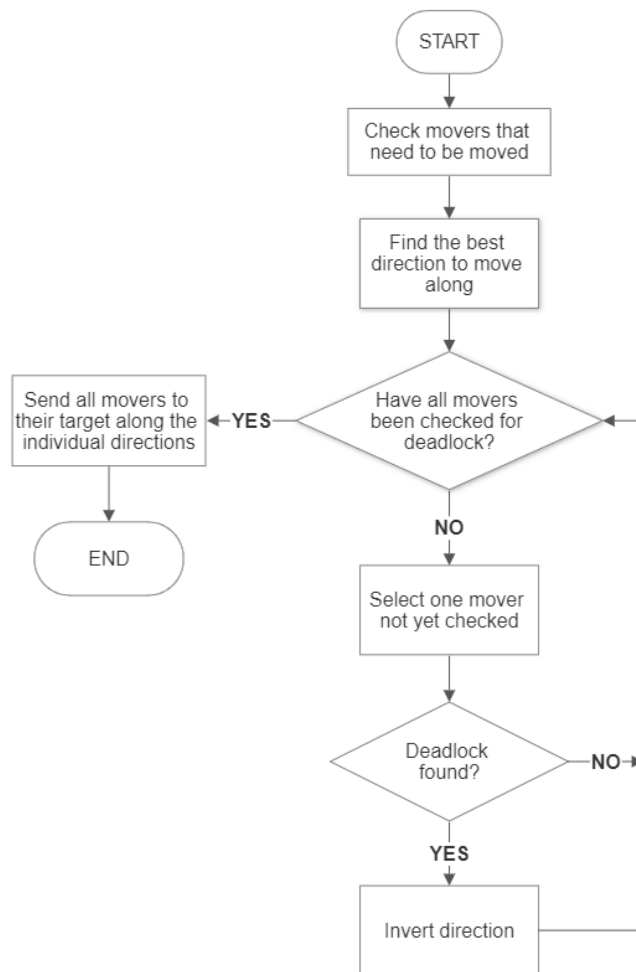


Figure 3.10: Flowchart illustrating the algorithm for Global Restore CA rephasing mode.

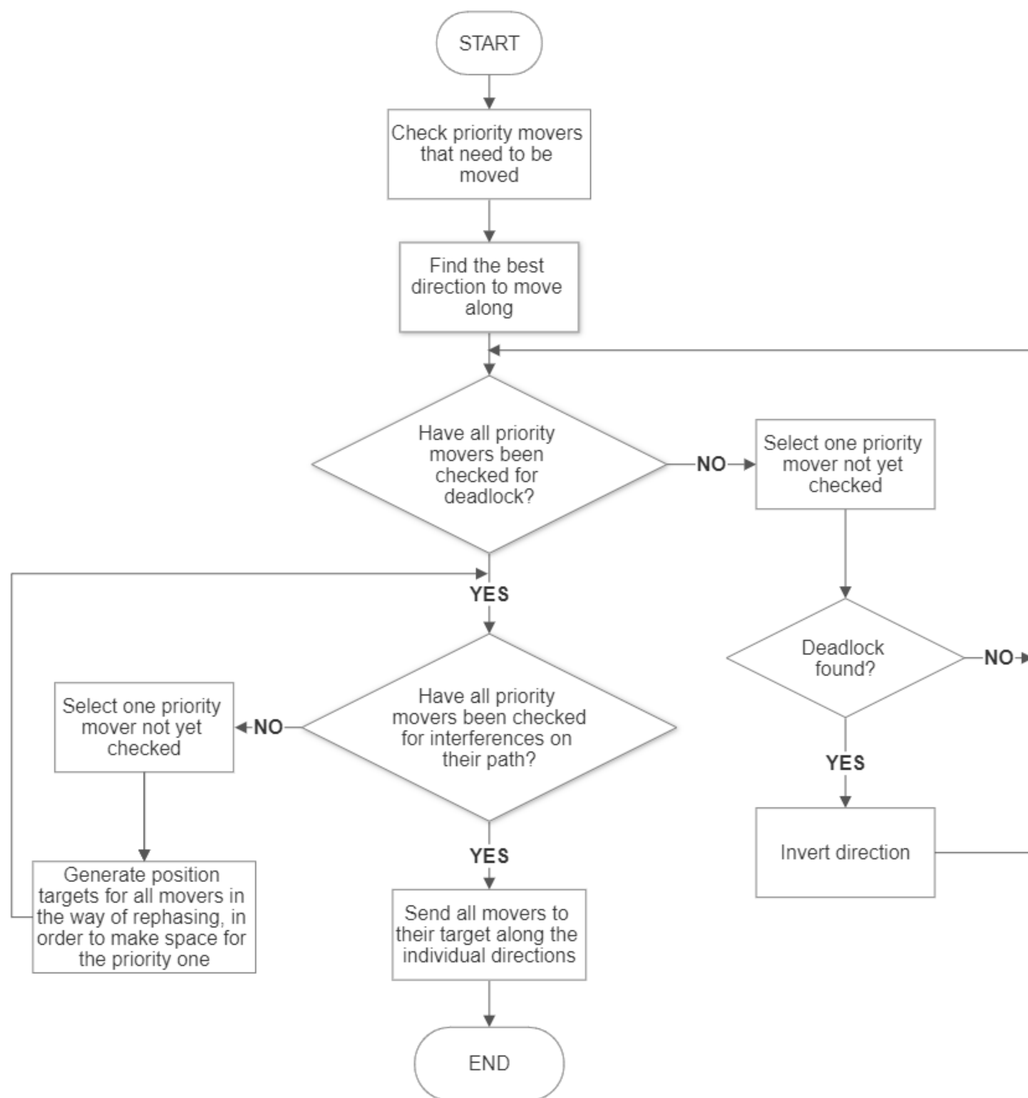


Figure 3.11: Flowchart illustrating the algorithm for Priority Restore CA rephasing mode.

Figure 3.12: Function block I/O interface.

figure 3.12. As before, in the main program (PRG), after the initial call in which all input parameters are loaded, with a rising edge of `I_Enable` the FB activates. After a few cycles, the block will output `Q_Done` and the output positions and directions will be available, respectively, at `Q_MoverTarget` and `Q_MoverDirection`: all necessary movers can now be moved together using CA functions. The I/O interface differs only for a few variables, which are:

I_CollisionAvoidanceGap (LREAL) Used to generate appropriate positions for movers to get out of the way in the priority mode. In the non-CA version, the mover length was used instead.

I_MoveSortOrder (BOOL) `TRUE` for ascending order. Again needed in priority mode for target generation, is needed to know the previous or next mover index given a direction. In the previous implementation, it was found by looking at distances.

Q_MoverTarget (ARRAY OF LREAL) It's an array of absolute modulo positions.

Q_MoverDirection (ARRAY OF MC_DIRECTION) Array of directions for every mover index.

`I_NextMover`, `I_EncoderGap` and `Q_Ready` are not needed in this implementation.

Also this time, the internal structure of the Function Block is based on a Finite State Machine (FSM) comprised of two different parts based on the logic shown in figures 3.10 and 3.11. Several methods have been re-used or slightly adapted from the previous case, such as the one shown as an example in figure 3.7.

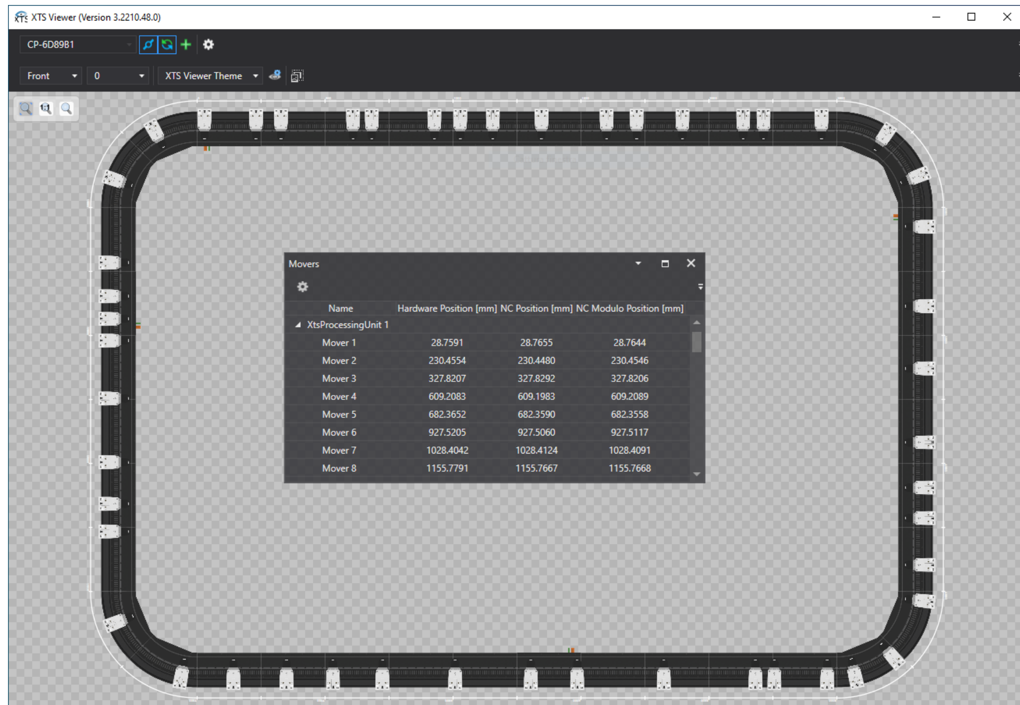


Figure 3.13: XTS Viewer from the XTS extension in TwinCAT 3. The loop used for testing in simulation is 10 meters long and contains 50 movers.

3.6 Testing

The proposed designs have been tested to check their effectiveness: first, they have been examined in a simple TwinCAT project using the XTS simulator tool, then they have been implemented on a real automatic machine project designed by IMA Automation and evaluated on its 3D model built for virtual commissioning.

3.6.1 TwinCAT setup

As explained in 1.3.2, the XTS extension for TwinCAT 3 includes a simulator that can be set up with an arbitrary configuration of motors and movers. It has been fundamental in the development of this work to test the progress step by step. In figure 3.13 is shown a screenshot of the tool and the used layout.

The testing methodology consisted of defining some arbitrary homologous configurations of movers and evaluating the ability of the rephasing tool in moving from one to another. Worst-case scenarios have been simulated in order to stress test the algorithms and their deadlock avoidance strategies.

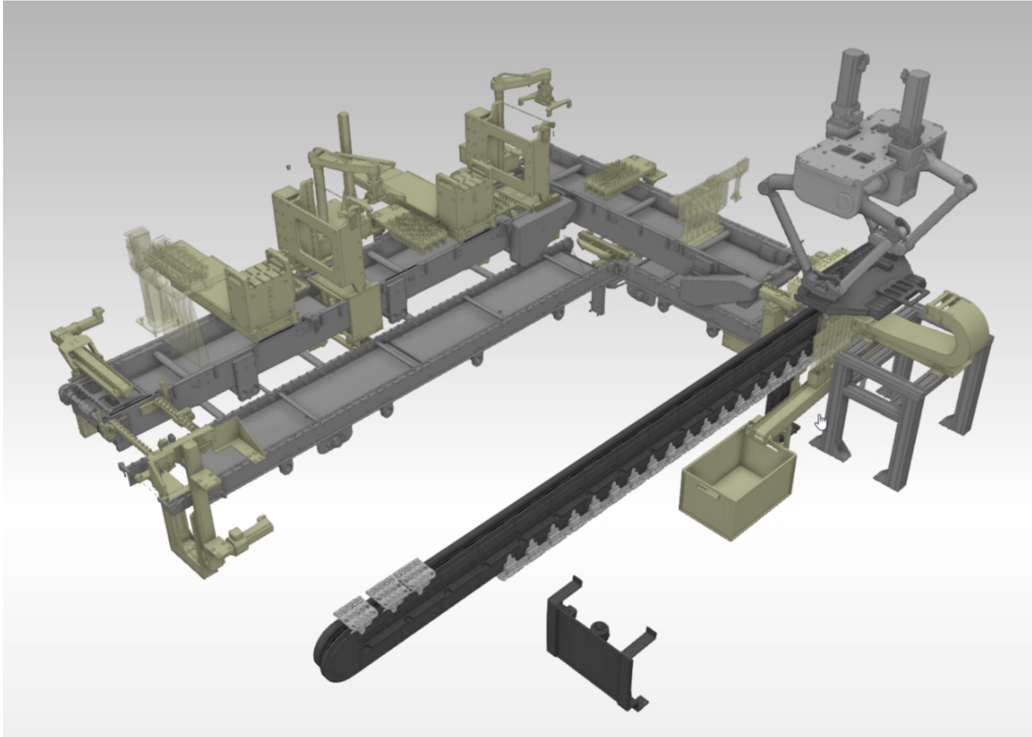


Figure 3.14: 3D model built in iPhysics of the automatic machine used for Virtual Commissioning. The XTS loop in this application is 9.5 meters long and contains 30 movers: in this case, it's used to transfer trays of products from one pick and place robot (on the right) to another (on the left, 3D model not present in the image). The production line continues after and before this machine.

3.6.2 Virtual Commissioning setup

To test the developed code on an existing project it has been implemented in its Starting program: as explained in section 2.2, the PackML Starting state is the necessary step to resume operations after the machine is stopped and correctly reset. The program is governed by a FSM in which has been added an instance of the Function Block to test, according to its I/O structure. To test the rephasing procedure, and simulate an operator's intervention, the movers positions have been modified by actuating the axes once the machine was stopped.

The Beckhoff IPC (see figure 1.2) has been responsible for running the XTS simulation and its PLC project while communicating with a Schneider PLC over Profinet, which was running the project of the rest of the automatic machine. A high-end desktop workstation running iPhysics was connected

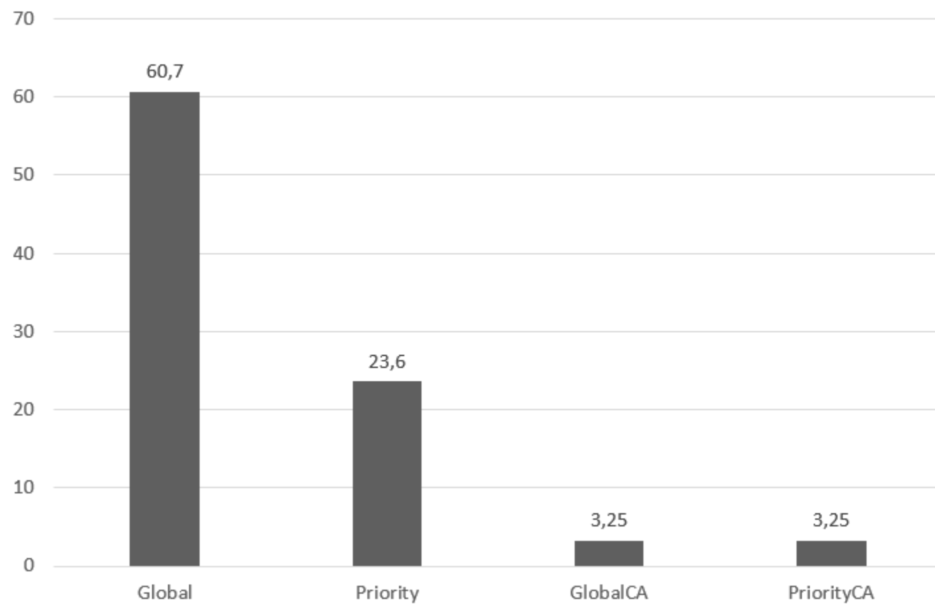


Figure 3.15: Average times (in seconds) required, by the different modes, to reach 10 arbitrary configurations in the setup shown in figure 3.13.

to the Schneider PLC: this has been used internally in IMA for the Virtual Commissioning of the represented automatic machine. The 3D model is shown in figure 3.14.

3.6.3 Results

All the proposed solutions have been able to solve the rephasing problems tested in the arbitrary loop and the real application's digital twin.

Even if the goal in designing this tool hasn't been the optimization of the rephasing time, some considerations on the different approaches efficiencies can be made. As previously explained, the performance of the CA modalities has been far superior to their counterparts: to complete the rephasing procedure of tens of movers, the average time has gone from a few minutes to a few seconds. Of course, the tested rephasing procedures have been complex beyond any probable real-life configurations: if only a few movers need to be displaced, every modality is able to succeed in a reasonable timeframe. In both approaches (CA and non-CA), the priority mode is marginally faster than the global mode, while there are orders of magnitude between the times if CA is being used or not.

In figure 3.15 is shown an example of the average rephasing times of the different modes.

Chapter 4

XTS workstation

In this chapter, the design for a generalized Function Block able to handle the queuing and positioning of XTS movers in every workstation on a track will be presented.

4.1 Preface

During the development of an XTS PLC project, a lot of time is dedicated to coding the movers' logic to transit from one workstation to the next: many times queues and buffers are present between stations and the number of movers needed differs at every step of the process. This flexibility is one of the strengths of these kinds of systems, but it requires significant effort to be correctly controlled.

Working with IMA Automation in the development of a new production line, in which several XTS loops are used as transfers between workstations, it has been clear that a generalized structure to handle the movers' logic (to be parameterized for every station) could save a lot of time. Being able to build simulations really fast in the XTS tool, even before virtual commissioning, could be beneficial, for example, in testing cycle times and productivity of the automatic machine, to correctly refine the mechanical design in the very early stages of the project.

The most common deployment of XTS in this industrial setting (almost always) has been to use it as a transfer between workstations where is needed grouping of product units. For example, in an automatic machine, there could be several workstations able to process the products transported on only one mover while many others, to optimize cycle time, perform parallel processing on multiple movers. In all these cases, the task that XTS needs to perform is usually pretty simple: move the required movers from one worksta-

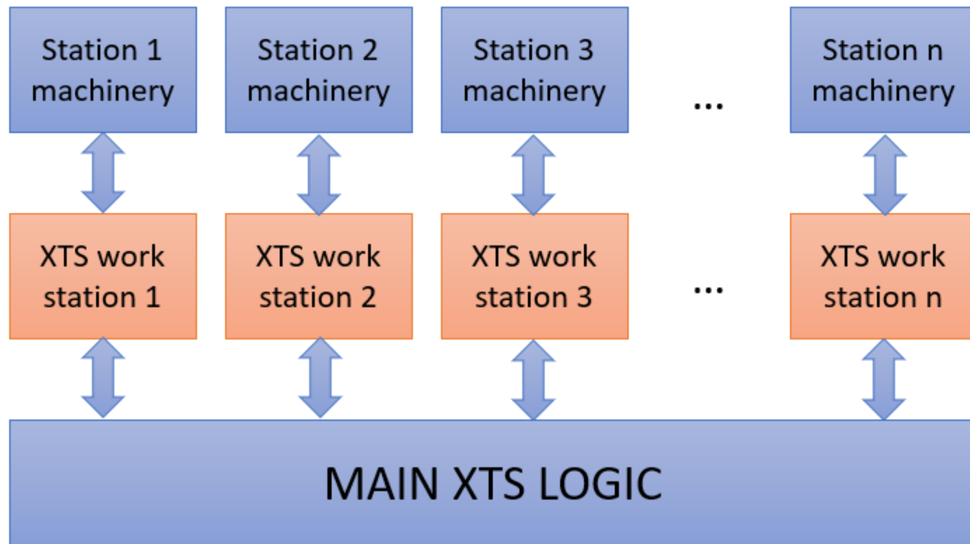


Figure 4.1: XTS workstation structure. In this picture is represented the logic architecture of the proposed solution. The bottom layer, indicated as "main XTS logic", handles axis (movers) movement and the overall machine operation: it communicates with n instances of the proposed "XTS workstation" structure and every one of them controls movers supply while synchronizing with its relative machinery's specific needs.

tion to the next and exchange data with the machinery actually performing the work at every station, while handling queues and accumulations. Almost every time, the workstation requires one (or a set of) mover(s) to reach some target positions on the track, wait standstill during the product processing, free the position once finished and repeat.

4.2 Proposed solution

Given the simple logic needed by most workstations along a track, the proposed solution consists of an architecture in which every workstation (machinery actually performing tasks on the product) communicates with its corresponding XTS workstation (handling movers coordination), while each of those communicates with the main XTS logic to handle movers motion. The structure is represented in figure 4.1.

The logic of *XTS workstation* comprises two distinct algorithms running simultaneously: the *Movers Adoption* and the *Movers Requests Handling*. Every XTS workstation is responsible for all movers in the queue before its station and for the ones sent to working positions: every mover along the

track can be assigned to at most one station at any given time. Therefore, the *Movers Adoption* routine will be responsible for periodically collecting movers who completed their task on the previous station and sending them to the current one's queue: it will be running all the time since it must free the previous station's work positions as soon as they are done. Conversely, the *Movers Requests Handling* must communicate with the main XTS logic and synchronize with the machinery's requests: it runs independently from *Movers Adoption* and is tasked to send movers to working positions and free them (for the next station to adopt) once processing is complete.

As represented in 4.1, the various instances of XTS workstation don't communicate directly with one another, but they share global variables as registers on the main XTS program.

4.2.1 Movers Adoption

As just explained, the main tasks of this asynchronous logic are two:

1. Look for movers from the previous station that have completed their tasks and need to proceed in the production line: adopt them.

2. Send those newly assigned movers to the current station's queue position, in order to free the previous platform as soon as possible.

These points can be achieved by constantly scanning the shared registers for movers that need to reach the specific station of an instance and are done in the previous one. Since more than one may be collected at once, they need to be sorted by distance before being sent to the queue position: always knowing the first of the line is essential to send the correct ones to work.

It may be the case, for some stations, that the product units need to stop there only if some specific conditions arise: for example, after quality control, all the good products must proceed while some of them need to be rejected. Hence, the reject station is not always needed. To handle this eventuality, the Movers Adoption must also check that the first mover in the queue (which means every mover for stations requiring only one at a time) actually needs to be worked in the station. If that's not the case, it is immediately freed and the next station will adopt it and move it to its queue. Checking only the first in line is essential for a station requiring multiple movers, because otherwise it would be unable to proceed and the whole loop would stop at a deadlock, as shown in the example of figure 4.4.

The flowchart of this algorithm is presented in figure 4.2 and an example of its operation in figure 4.3.

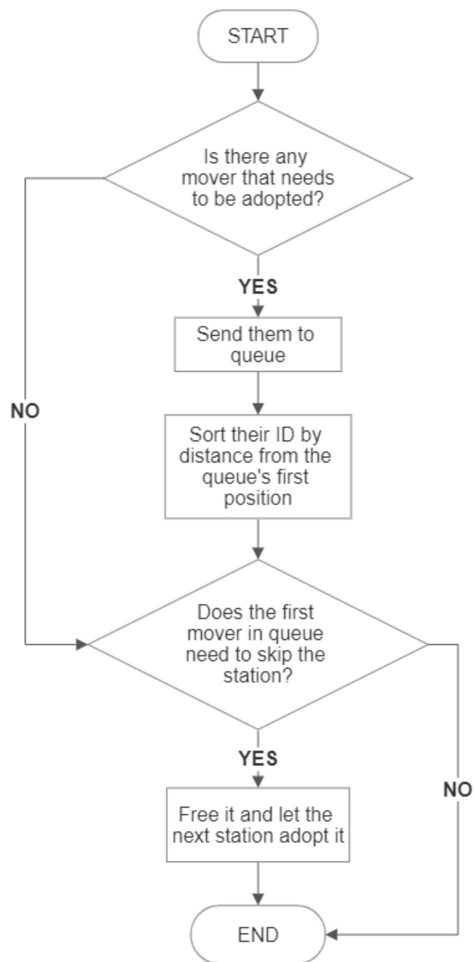


Figure 4.2: Flowchart illustrating the algorithm for the Movers Adoption routine.

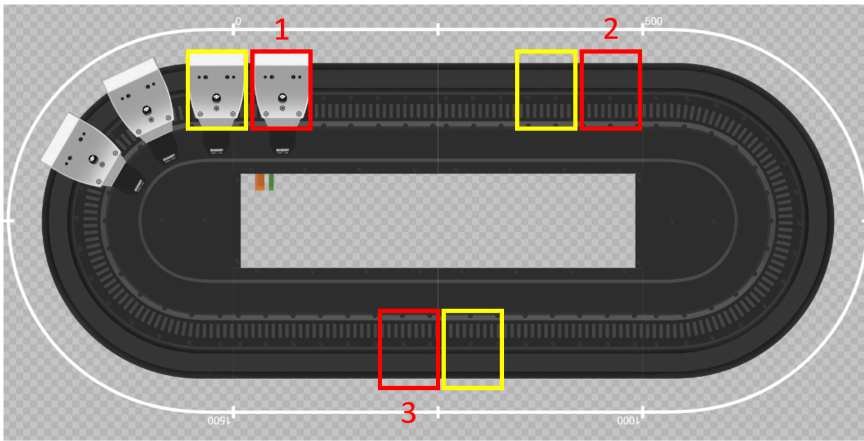
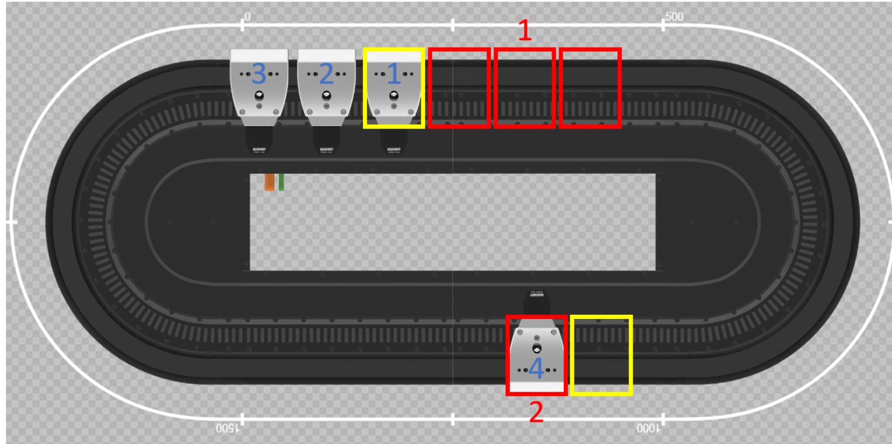
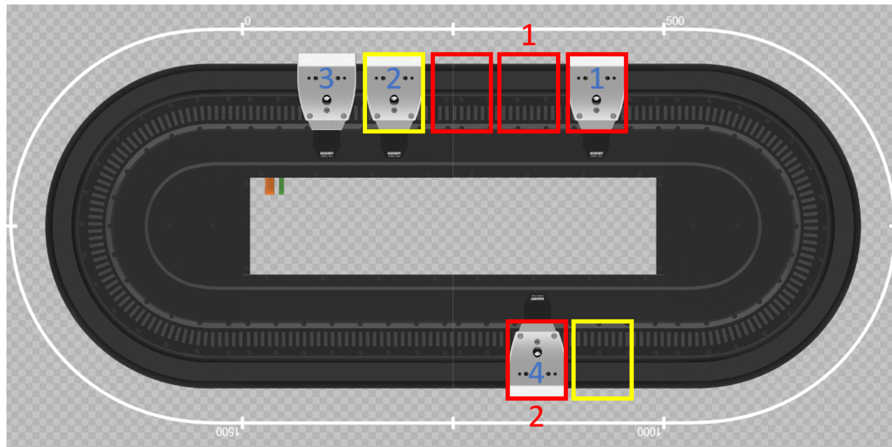


Figure 4.3: Mover adoption example. Let's consider a simple loop with 3 workstations: they can process only one mover at any given time. The work positions are indicated in red while the yellow positions indicate the first mover in the queue for that station. In this case, movers are advancing clockwise. Station 1 is working on one mover while other 3 are in queue behind it. If we consider the Mover Adoption routine of station number 2, once the mover in position 1 is done with its processing it will be adopted and sent to the queue position before station 2 (yellow rectangle). In case the specific mover doesn't need to be worked on station 2, it will be immediately freed upon adoption because of being the first in line. Then, the instance of station 3 will adopt him and send it to its queue position right away.



(a) Let's start from a condition similar to figure 4.3 but, this time, only two stations are present and the first one requires 3 movers to process products in parallel. At this stage, 3 movers are available in its queue (ID 1, 2 and 3). If all three of them need to be worked in station 1, they can just reach the required positions. If mover 1 needs to skip station 1, it can just move to station 2's queue while mover 2 goes to the first position of station 1 and mover 3 to the second: they will wait for mover 4 to come (once finished in station 2).



(b) If however, for example, mover 2 needs to skip the station, it cannot reach station 2 because mover 1 is already in place: the system reaches a *deadlock*. Many solutions have been evaluated to solve this issue, but the best one for the application at hand has been found to be to process mover 2 anyway in station 1, even if it needed to skip it. Of course, this behaviour could be customized for different solutions.

Figure 4.4: Example of handling station skipping in the XTS simulator tool.

4.2.2 Movers Requests Handling

The proposed design for this part of the software consists of the following steps: considering a specific station on the track, when from the machinery comes a request for movers, the ones found at the beginning of the queue (adopted by the previous logic) are sent to the specified work positions. Once all the required movers have reached their target location, the processing can be performed while they wait standstill. Then, another request could arise for the same movers to move in other positions or they could be freed for the next station to collect them. Not all of the movers present in the workstation could be allowed to move to the next one: the number of them that can proceed in the loop can be specified every time. Once they are freed, for each individual mover, the next station for them to be worked on is specified: while the next on the track is fixed, they could need to skip it, as in the examples of figure 4.3 and figure 4.4.

In figure 4.5 is shown the flowchart of the proposed algorithm: this and the previous one (section 4.2.1) are running concurrently.

4.2.3 Implementation

Similarly to what has been seen in sections 3.4.3 and 3.5.3, the previous solutions have been implemented in a Function Block coded in ST according to IEC 61131-3 standard: its I/O interface is shown in figure 4.6. The instance of the FB needs to communicate with a program (PRG) to handle the communication with its specific station machinery (actually performing processes on the transported products): with a rising edge of the `I_Enable` input all parameters are loaded and, if the FB station is correctly parameterized, it activates (`Q_Active` is `TRUE`).

In the program, when the machinery asks for movers to work on, the `I_MoverRequest` input of the FB is set to `TRUE`. The workstation sends the required movers in the specific positions and, when all are in place, rises the `Q_PositioningDone` flag while waiting for further commands. In the program, this gets redirected to the corresponding machinery. At this stage, two inputs are contemplated: another `I_MoverRequest`, with another set of positions for the same mover, or the input `I_JobDone`. If the latter becomes `TRUE`, the FB frees the specified amounts of movers and gets ready to receive another `I_MoverRequest`.

Here follows a detailed description of the I/O interface: variables with the prefix `I_` are inputs, while `Q_` is for outputs.

I_Enable (BOOL) A rising edge of this input will activate the function block.

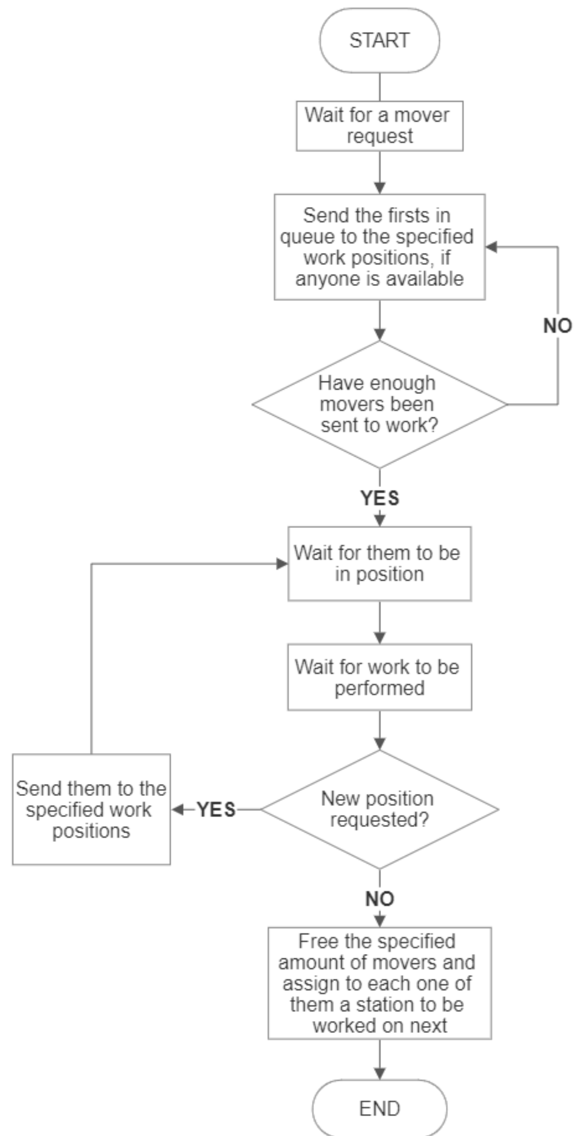


Figure 4.5: Flowchart illustrating the algorithm for the Movers Requests Handling.

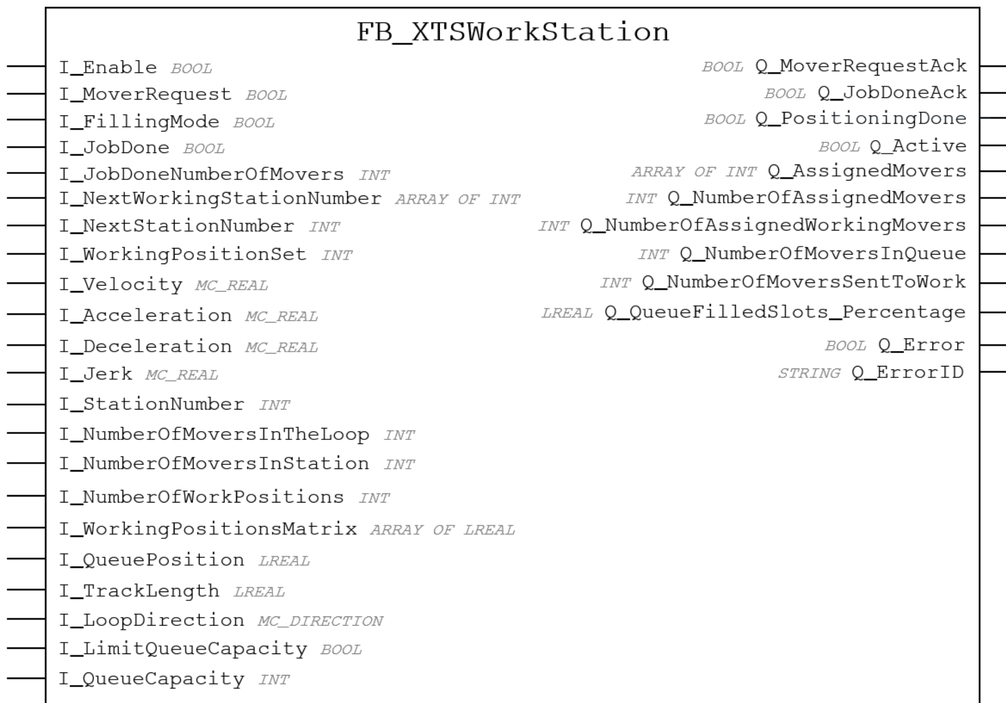


Figure 4.6: Function block I/O interface.

I_MoverRequest (BOOL) Trigger to send a specified number of movers into a predetermined set of positions.

I_FillingMode (BOOL) Special input used only in the program of the first station: is needed to enable the whole loop. If this is `TRUE`, everything works as expected, while if this input falls the first station of the loop goes in *emptying mode*: the FB will not process any other mover and the subsequent stations will work on the remaining ones until the whole production line is unloaded.

I_JobDone (BOOL) Trigger to free the current movers and get ready for another mover request.

I_JobDoneNumberOfMovers (INT) The number of movers that need to be freed when `I_JobDone`. By default, it's all of the ones in working positions, but can be fewer.

I_NextWorkingStationNumber (ARRAY OF INT) For each mover in `I_JobDoneNumberOfMovers` when `I_JobDone` rises, set their next station to be worked on. By default is the upcoming in the loop.

I_NextStationNumber (INT) Fixed parameter for each instance that depends on the physical layout of the machine. It's used in the Movers Adoption routine.

I_WorkingPositionSet (INT) Indicates a row of `I_WorkingPositions Matrix`: the elements are the positions for the movers requested by the machinery with `I_MoverRequest`.

I_Velocity (MC_REAL) Motion parameter for the axes. It's used both to reach the queue and to send movers to their work positions.

I_Acceleration (MC_REAL) Motion parameter for the axes. It's used both to reach the queue and to send movers to their work positions.

I_Deceleration (MC_REAL) Motion parameter for the axes. It's used both to reach the queue and to send movers to their work positions.

I_Jerk (MC_REAL) Motion parameter for the axes. It's used both to reach the queue and to send movers to their work positions.

I_StationNumber (INT) Number of the station represented by the instance.

I_NumberOfMoversInTheLoop (INT) Fixed parameter for each instance.

I_NumberOfMoversInStation (INT) Fixed parameter used to specify the number of movers required by each instance to process them.

I_NumberOfWorkPositions (INT) Fixed parameter for each instance that defines the number of different position sets that could be requested.

I_WorkingPositionsMatrix (ARRAY OF LREAL) Fixed matrix of parameters for each instance. The matrix is `I_NumberOfWork Positions` rows by `I_NumberOfMoversInStation` columns. Each time there is a `I_MoverRequest`, a row of positions is selected via `I_WorkingPositionSet`.

I_QueuePosition (LREAL) Fixed parameter for each instance. It's the position at which are sent all the movers in the queue. Collision avoidance handles the queue implicitly.

I_TrackLength (LREAL) Parameter needed to compute the modulo position at every step in which a calculation is involved. Since the system at hand produces periodic motion of the axis, the period is essential information to evaluate distances.

I_LoopDirection (MC_DIRECTION) Motion parameter for the axes. This is the default direction to advance movers.

I_LimitQueueCapacity (BOOL) Optional input that can be used to limit the number of movers that a station can have in queue. If `TRUE`, the station will not adopt movers beyond the limit imposed by `I_QueueCapacity`.

I_QueueCapacity (INT) Optional input, used only if the queue is limited by `I_LimitQueueCapacity`. This is the maximum number of movers allowed to be in the queue.

Q_MoverRequestAck (BOOL) Used to acknowledge an `I_MoverRequest` input.

Q_JobDoneAck (BOOL) Used to acknowledge an `I_JobDone` input.

Q_PositioningDone (BOOL) Output communicating that all the requested movers are in position and standstill.

Q_Active (BOOL) If `TRUE`, parameters have been correctly loaded and the FB is working.

Q_AssignedMovers (ARRAY OF INT) Array of indices of the movers that have been adopted. Here are all the movers being handled by the instance (in the queue or sent to work) sorted according to their order. The first element is the leading one in the track.

Q_NumberOfAssignedMovers (INT) Number of elements of the array `Q_AssignedMovers` populated. Used for debugging.

Q_NumberOfAssignedWorkingMovers (INT) Number of elements of the array `Q_AssignedMovers` which need to work in this instance station. Used for debugging.

Q_NumberOfMoversInQueue (INT) Number of elements of the array `Q_AssignedMovers` which are in queue. Used for debugging.

Q_NumberOfMoversSentToWork (INT) Number of elements of the array `Q_AssignedMovers` which are sent to work. Used for debugging.

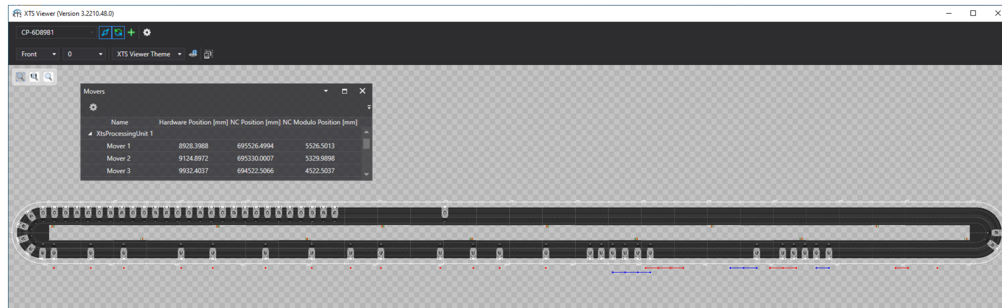


Figure 4.7: XTS Viewer from the XTS extension in TwinCAT 3. The loop used for testing in this simulation is 15 meters long and contains 60 movers.

Q_QueueFilledSlots_Percentage (LREAL) If the queue is limited, the filling percentage.

Q_Error (BOOL) Debugging flag used to check for errors. If `TRUE` an input parameter is incorrect.

Q_ErrorID (STRING) If `Q_Error` is `TRUE`, this is the error description.

The internal structure of the Function Block is based on the two algorithms previously illustrated: the request handling logic is governed by a Finite State Machine (FSM) while movers adoption by an independent routine inside the same FB. A few methods have been added to handle common tasks, such as sorting mover IDs by position.

Every instance of this FB needs to be implemented in a separate program (PRG) in the PLC project: the same structure can be reused with minimal changes and correct parameterization of FBs is all that's needed.

4.3 Testing

The proposed design has been tested to check its usefulness: using the XTS viewer in TwinCAT, a simple simulation has been set up. The goal was to test the cycle times of a real automatic machine in development at IMA Automation: its mechanical design still needed to be finalized.

4.3.1 TwinCAT setup

As can be seen in figure 4.7, the setup has been configured according to the project needs. The placement of 18 stations along the track needed to be verified. In particular: it was necessary to test the cycle time of the whole

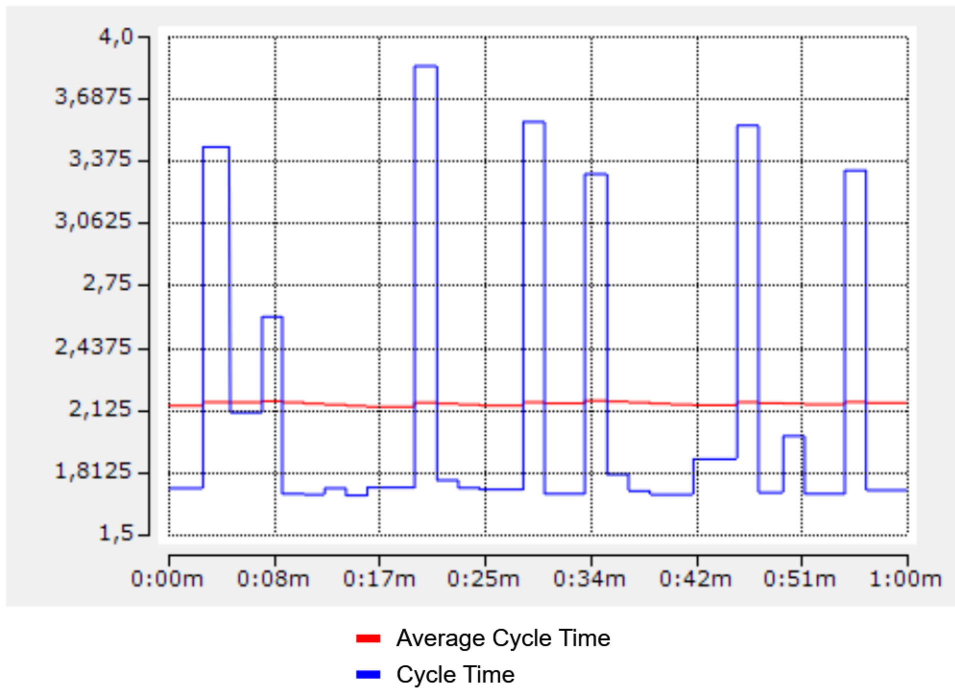


Figure 4.8: Cycle Time Trace of the last station. On the horizontal axis is shown the simulation time of the trace (1 minute), while on the y-axis the cycle time is expressed in seconds.

loop while verifying the correct distance between stations to allow for queues to form freely. To test the productivity of the machine, simple timers have been used to simulate the processing time of every station.

4.3.2 Results

Concerning cycle times, in this application every mover will be expected to transport one product unit. To test the productivity of the machine, the output of the last station of the loop has been studied. As can be seen in figure 4.8, while a fluctuation is present, the average cycle time appears to be close to 2.1 seconds. This value has been computed by measuring the time elapsed between two consecutive processed movers. Since the required productivity of the final design will need to be 30 product units per minute, this initial layout has been found satisfying.

Regarding the spacing needed for queues, 3 stations on this track need to work on multiple movers: station 14 needs 4 movers, 15 needs 3 and 16 needs 2. By monitoring the movers present in their queues in simulation,

thanks to the output `Q_NumberOfMoversInQueue` illustrated in 4.2.3, it has been possible to estimate the minimum distance required before each station to ensure smooth operations. The trace is shown in figure 4.9.

The proposed solution for a generalized workstation has been very useful to deploy these simulations very quickly. In the very early stages of design, the possibility of changing the logic of all movers by simply tuning some parameters in the appropriate instances has been extremely convenient.

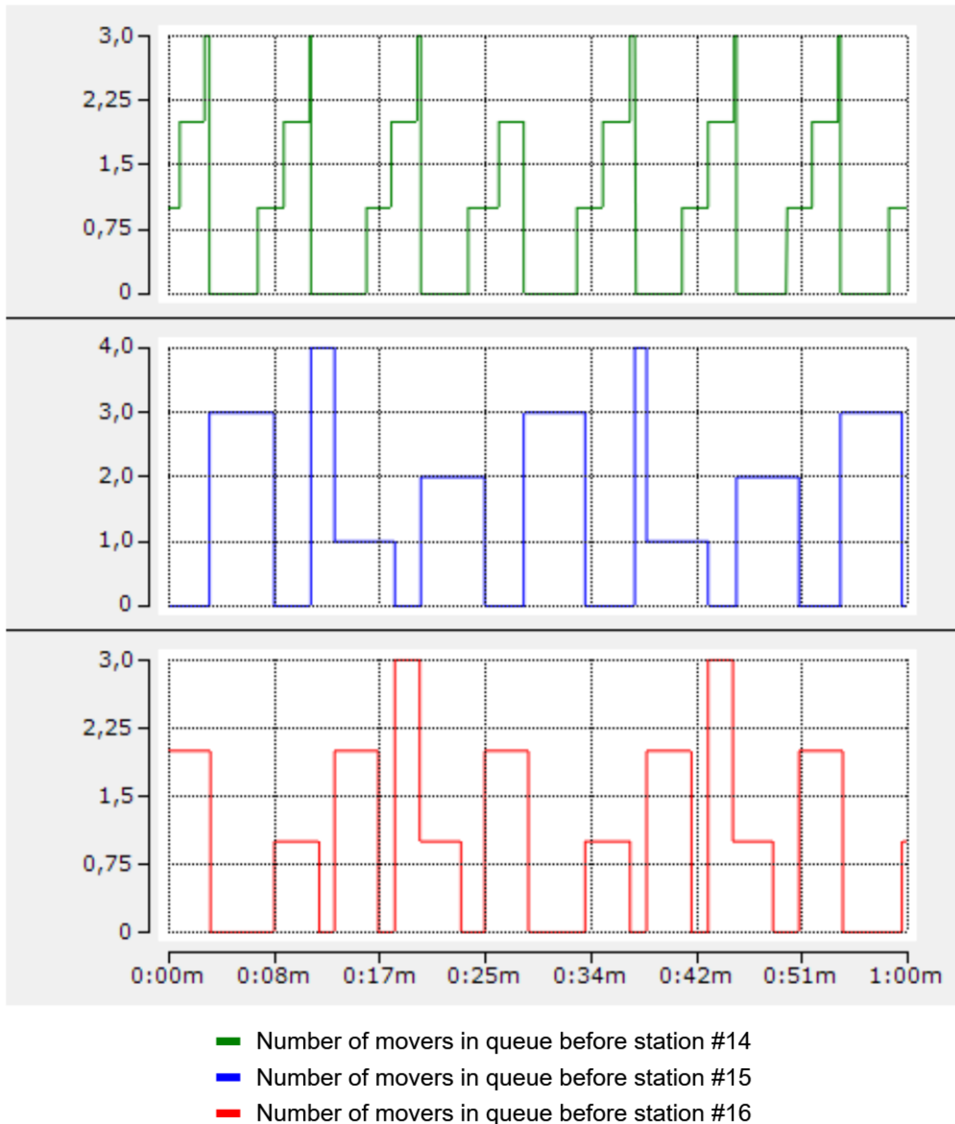


Figure 4.9: Trace of the numbers of movers in queue. On the horizontal axis is shown the simulation time of the trace (1 minute), while on the y-axis is indicated the number of movers in the queue at any given time. Thanks to these findings, it has been possible to move the stations closer than what was originally planned.

Chapter 5

Conclusions

5.1 Achievements

During the internship that led to this master's thesis, different software solutions to motion applications on the Beckhoff XTS system have been designed and tested. To achieve it, in IMA Automation has been possible to familiarize with the source code of automatic machines and the Beckoff development environment. While the developed code has been proven effective using the Beckhoff simulation tool and the virtual commissioning 3D model, a test on a real system is still required.

5.2 Future developments

As just stated, a test on a real system will be performed in the near future to complete the development.

Concerning the *rephasing tool*, a proper optimization study is still required. Even if the developed work is able to complete the presented task, it may be the case that a different algorithm would perform better than the proposed solutions. Due to the heuristic nature of the current logic, a proper mathematical formulation could lead to an optimal solution to the problem.

Regarding the *XTS workstation*, the amount of generalized features that could be added depends greatly on the task at hand. The scope of this work, in the industrial environment of IMA Automation, has been to reduce development time for future XTS tracks' logic. In this framework, it's clear that has been avoided the design of features too specific that would require more time to be generalized with respect to the actual saving that they would bring. That said, an example of a feature that could be useful to introduce is the possibility to parameterize dynamic stations: sometimes the movers

need to advance at constant speed to allow the station to work. Until now, only the case of standstill processing has been contemplated.

Bibliography

[1] Beckhoff Automation. *Slides from the course on TwinCAT 3*. 2023.

[2] Beckhoff Automation. *Slides from the course on XTS*. 2023.

[3] Beckhoff Automation. *Tc3_McCollisionAvoidance library documentation.* URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tf5410_tc3_collision_avoidance/1419730315.html&id=6949934259970075206.

[4] Beckhoff Automation. *Tc3_McCoordinatedMotion library documentation.* URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tf5410_tc3_collision_avoidance/8892648715.html&id=3846970227931590176.

[5] Beckhoff Automation. *TF5410 — TwinCAT 3 Motion Collision Avoidance extension documentation.* URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tf5410_tc3_collision_avoidance/index.html&id=.

[6] International Society of Automation. *ANSI/ISA-88.00.01-2010 Batch Control Part 1: Models and Terminology.* 2010. ISBN: 978-1-936007-75-2.

[7] International Society of Automation. *ANSI/ISA-TR88.00.02-2015, Machine and Unit States: An implementation example of ANSI/ISA-88.00.01.* 2015, pp. 11–23. ISBN: 978-1-941546-65-9.

[8] *IEC 61131-3.* URL: https://plcopen.org/iec-61131-3.