GridHub: a grid-based, high-density material handling system.

Gang Hao
University of Louisville

GRIDHUB: A GRID-BASED, HIGH-DENSITY MATERIAL HANDLING SYSTEM

By

Gang Hao
M.S., University of Nebraska-Lincoln, 2011
B.Eng., Nanjing Forestry University, 2008

A Dissertation
Submitted to the Faculty of the
J. B. Speed School of Engineering of the University of Louisville
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy in Industrial Engineering

Department of Industrial Engineering
University of Louisville
Louisville, Kentucky

May 2020

GRIDHUB: A GRID-BASED, HIGH-DENSITY MATERIAL HANDLING SYSTEM

By

Gang Hao
M.S., University of Nebraska-Lincoln, 2011
B.Eng., Nanjing Forestry University, 2008

A Dissertation Approved On

August 15th, 2019

by the following Dissertation Committee:

Professor Kevin R. Gue, Dissertation Director

Professor Lihui Bai

Professor Monica Gentili

Professor Dan Popa

ACKNOWLEDGEMENTS

This dissertation is dedicated to my family.

I would like to express my appreciation to my advisor Dr. Kevin Gue. Thank you for your guides and generous supports in my research. I like working on this topic, and these five years are the happiest time in my life. I also thank my committee members, Dr. Lihui Bai, Dr. Monica Gentili, and Dr. Dan Popa. Thank you for serving in my committee and your valuable advices.

I really appreciate helps from Benedikt Fuß, Bin Weng, Donghuang Li, Indika Wijayasinghe, and Josh Christian on AnyLogic, statistics, simulation, control theory, and writing review. I would like to thank Emily Burks and all of my fellow students in LoDI. I am really appreciate your helps, and I have very happy time in our office.

Completing my PhD study in a big milestone in my life. I have finished all of my formal school education. I would like to thank for Dr. Jeonghan Ko, Dr. Yongxin Ji, and all my teachers in my life. Without your education, I cannot have today's achievements.

ABSTRACT

GRIDHUB: A GRID-BASED, HIGH-DENSITY MATERIAL HANDLING SYSTEM

Gang Hao

August 15, 2019

In the past twenty years, the share of e-commerce has increased (FRED, 2019). Since more and more activities, such as picking and sorting customers' orders, are done in warehouses, high efficiency warehouses are in demand. Furthermore, the efficiency of warehouses is related to customer satisfaction (Colla and Lapoule, 2012). Storage systems are key components in warehouses, which are related to the efficiency of warehouse operations. In this dissertation, we address an automatic puzzle-based storage system under decentralized control. We call this system GridHub.

GridHub meets standards of Industry 4.0 (Lasi et al., 2014), and it features high throughput with parallel order processing.

In the first part of this research, we describe a GridHub which can handle unit-sized items; that is, one box only occupies one conveyor module. The GridHub is capable of moving boxes in all cardinal directions. It can complete multiple material handling tasks, such as sorting, sequencing, retrieving, and storing without changing the control algorithms. To move the active boxes to their targets, we developed a decentralized control algorithm to arrange box movements. The algorithms are executed by conveyor modules cyclically, and all actions in the execution process are one iteration of the algorithm. There are three phases in one iteration (assess, negotiation and convey), and several steps consist of one phase.

The conveyor modules execute the algorithm simultaneously and synchronize at every step. The goal of the control algorithms is to move active boxes into their immediate destinations, and the key idea of the algorithm is to move away other boxes for the active boxes through message passing process.

Negotiation behaviors are patterns of action generated by the conveyor modules while executing the algorithms. We describe these behaviors and explain how they affect the transfer process of active boxes. Some of those behaviors and other actions, which can prevent the transferring processes of boxes, are listed and discussed. These actions are related to deadlock and livelock in the GridHub. We prove that GridHub is deadlock free, and it is also livelock free under certain conditions.

In the second part of this research, we extend the unit-sized GridHub by enabling it to handle non-unit-sized boxes meaning every box can occupy more than one conveyor module. We name the new GridHub the *NU GridHub*. The control algorithms of the NU GridHub are developed based on the unit-sized GridHub's algorithms by adding new rules. Performance of the NU GridHub is also measured and discussed.

GridHub is the first grid-based material handling system to offer four-way movement of stored items with a rich set of material handling task – storage, retrieval, sorting, and sequencing. GridHub is also the first grid-based system to implement a decentralized control algorithm based on "nested attempts," a feature the guarantee deadlock free operation. Finally, the NU GridHub is the first grid-based solution to handle bigger boxes, which have not been done for a grid-based system under the virtual aisle method.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

In the past twenty years, the share of e-commerce retailers in the total retail market sale has increased (see Figure 1).

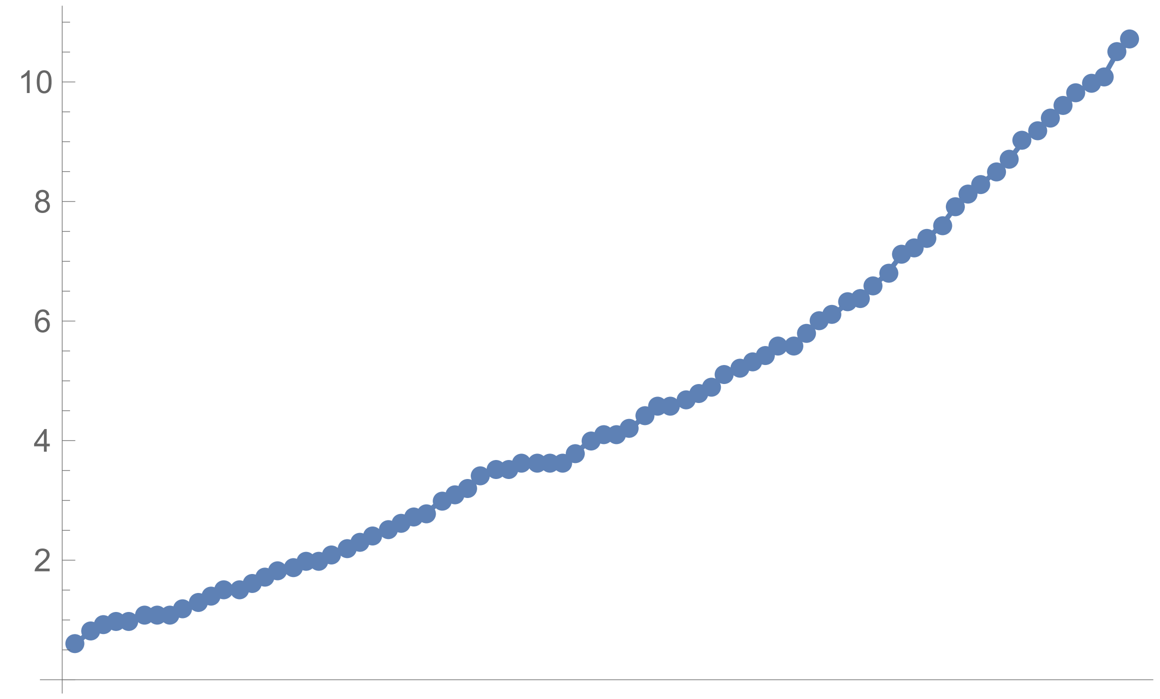


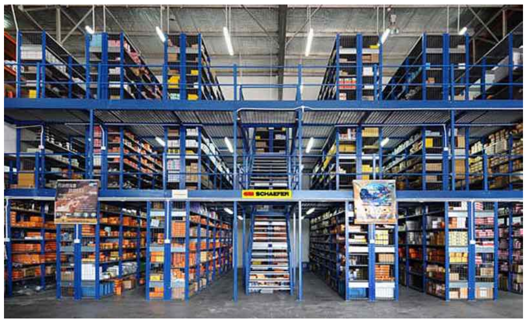
Figure 1: Share of sales of e-commerce in the total sales (Data source: FRED). The y-axis indicates the market share of e-commerce in the total retail market sales, and the unit is %. The x-axis is the date (first date of every month from 1999-10-01 to 2019-04-01).

Because goods are usually shipped to customers from warehouses or distribution centers directly, and customers expect to have short preparation time for their orders (Colla and Lapoule, 2012), high efficient warehouses are necessary for customer satisfaction. Hence, highly efficient warehouses are related to the retailer's success.

A storage system is a material handling system used to store goods in a warehouse. Most of the operations of a warehouse are completed by, or inside, a storage system. Hence, well designed storage systems are necessary for efficient warehousing. The conventional storage systems, which are are well developed and optimized, have been used by the industry for decades (Figure 2).

(a) Racks (Massey Rack, 2017).

(b) Flow rack (CLARK Associate MH. Inc., 2017).


(c) Mezzanine storage (mezzaninestoragesystems101, 2014).

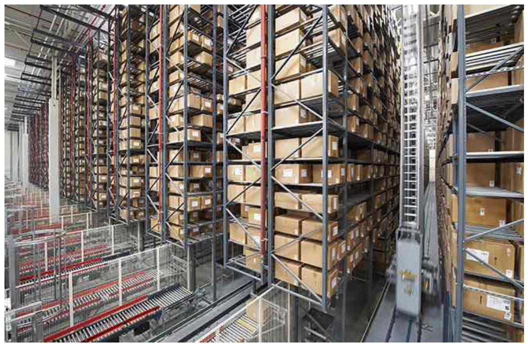

(d) Automated storage and retrieval system (ASRS) (Vanderlande, 2017).

Figure 2: Examples of conventional storage systems.

However, these systems face multiple challenges in the era of e-commerce, such as:

- Picking and sorting orders in pallet racks or mezzanines needing a long travel time; Processing customer orders in a flow rack requires more workers, and high picking accuracy is hard to achieve.
- If a warehouse is required to have a high utilization rate, the upper bound of a pallet rack's utilization rate is around $\frac{2}{3}$; Flow racks have to store a single SKU in each slot, and the re-configuration for a flow rack is difficult.

Industry 4.0 (Lasi et al., 2014) is new concept set that emphasizes building connections among all entities and decentralized decision making in a industrial system, such as a manufacturing system or a material handling system. The benefits of connections and decentralization are flexibility and efficiency. For example, a storage system can be expanded with small changes over a short time period, or a system is able to process multiple orders with different requirements.

A grid-based system is an automated storage system that consists of multiple pieces of homogeneous transportation equipment or multiple modules, such as conveyors or Autonomous Guided Vehicles (AGVs). Modules of a grid-based system have to meet the following requirements:

- The modules are identical, which means that they both have the same hardware and software.
- They can either consist of a grid-like layout, or the items they are handling can consist of a grid-like layout.
- Every module can connect with some other modules electronically.
- Instead of a single controller to order every module where to move the stored items, each of the modules can decide its movement individually by communicating with other modules.

The grid-based systems consists of identical modules. These modules are called FlexConveyors (see Figure 3). Multiple conveyors can comprise a grid(Figure 4). A top-view abstract representation of this system is in Figure 5.

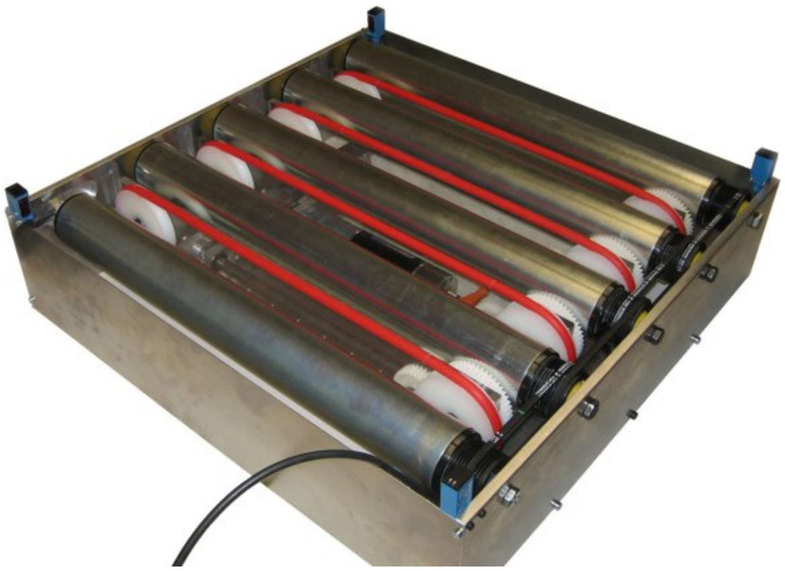


Figure 3: FlexConveyor (an individual module) (Uludağ, 2012). A FlexConveyor is able to receive an item from any side, and it can also move the item to any side. The rollers of a FlexConveyor move an item in one pair of opposite directions, and the belts move the item in the other pair of directions.

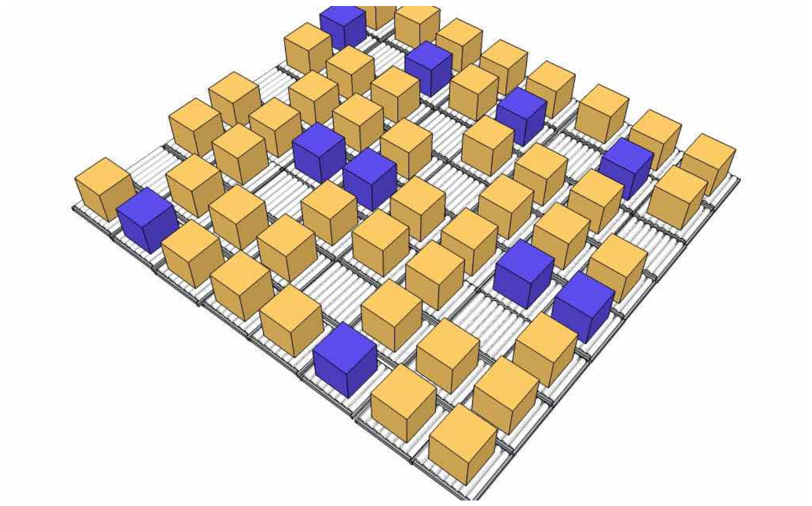


Figure 4: Grid-based system (Gue, 2014). Items on these conveyors are in two categories: requested items, which need to be moved out of the system or transferred to certain locations in the system, are displayed in blue; stored items are in yellow.

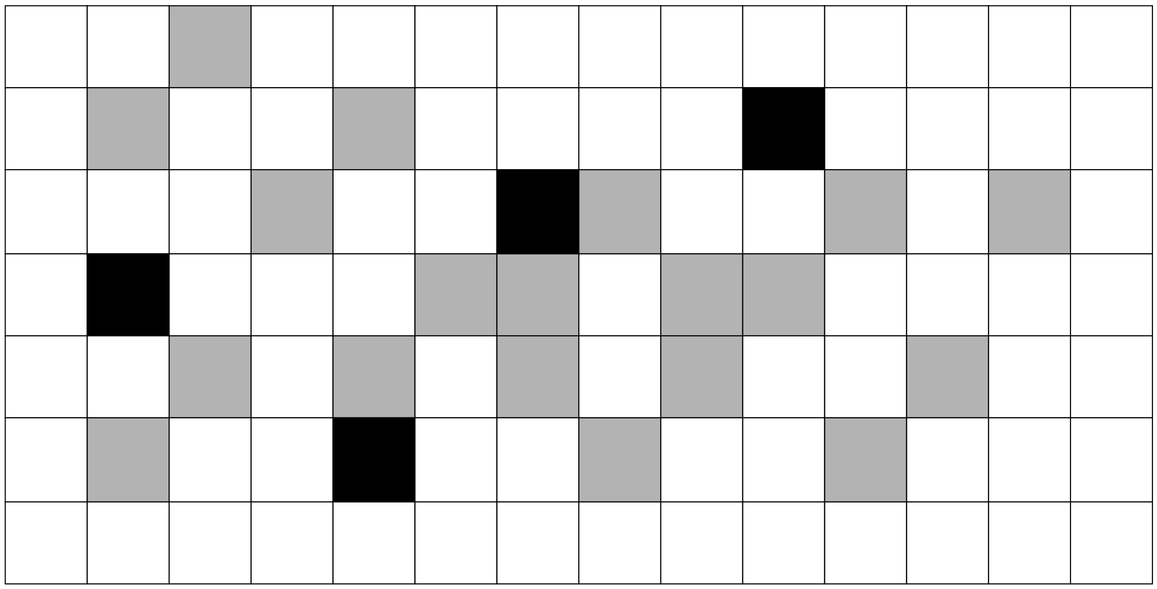


Figure 5: Grid-based system schematic diagram. The black squares represent the requested items; the grey squares are used to represent stored items; the white squares indicate empty conveyors.

Based on previous works, grid-based systems are capable of doing one or multiple of the following tasks.

- Storage: systems receive items and store them for long periods.
- Retrieving: systems move requested items out of any locations that are used for releasing items.
- Sortation: systems move requested items out of the system through specific locations.
- Sequencing: systems move requested items out of the system in certain sequences.

The grid-based system can process multiple parallel orders, and it is easy to setup these systems. To control these systems, two concepts, which are path reservation and virtual aisle, are developed (detailed explanations are in Chapter 2). However, some questions arise:

1. In these systems, requested items can be moved only in one direction or a pair of opposite directions. Is there a method to move requested items in four

directions?

2. If a group of items is requested to be released at a certain location with certain sequences, how do we form this problem and how do we solve it?
3. If a bigger item is stored in a grid-based system, how do we move it? For example, is this possible to introduce a *2* $\times$ *1* item or a *3* $\times$ *3* item into the system and move it to required locations with decentralized control?
4. How do we verify the control algorithms of a new grid-based system?

1.2 Objects and structure of this research

To answer these questions, we develop an improved grid-based system called "GridHub" (Figure 6), which consists of a grid of identical square conveyors (Figure 3). GridHub is capable of doing all the material handling tasks we mentioned – storage, retrieval, sorting, and sequencing. It also allows items to move in and out a all four sides. Like other grid-based algorithms, GridHub implements decentralized control, but using a novel "nested negotiation rule" that guarantees deadlock free operation.

We divide the research into two parts, based on whether a GridHub can handle non-unit-sized items. In the first part, we design a GridHub that is able to receive and release unit-sized items on all four edges. Items can be received released at any specific locations around the GridHub. In the second part, we modify GridHub for transferring non-unit-sized items, and we also explore system performance.

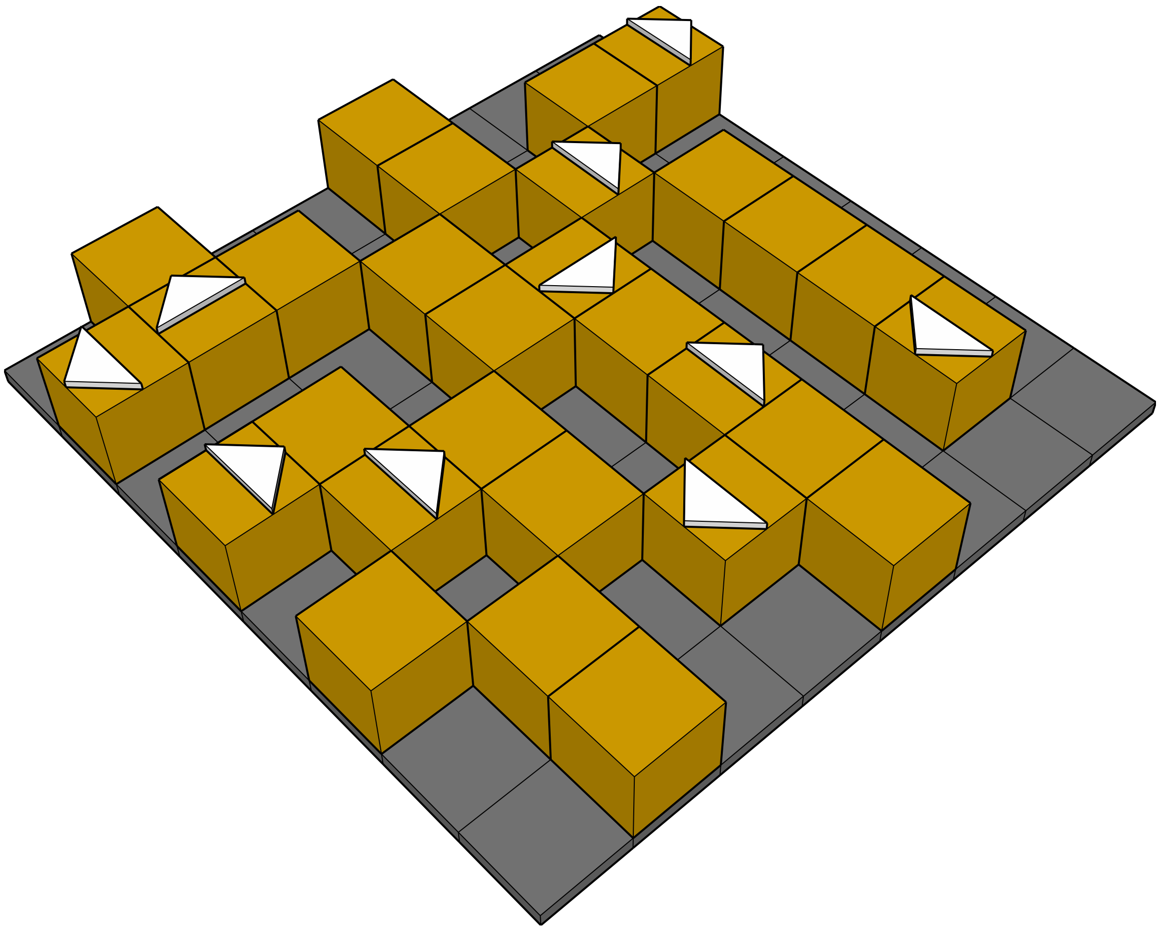


Figure 6: An analog of the GridHub with requested item moving in four directions. The grey tiles are conveyors, and boxes are the yellow cubes. A white triangle on the boxes indicates the direction to move the requested boxes.

1.3 Dissertation organization

The rest of the dissertation is organized as follows: Chapter 2 reviews literature and summarizes the related background. We explain terminologies of the grid-based system first, then existing grid-based systems are reviewed. In the second part of this chapter, related background, such as controlling deadlock solutions, are summarized. In Chapter 3, the system descriptions of unit-sized GridHub are presented first. Then, we explain the control algorithms. In Chapter 4, system behaviors, deadlock, and livelock in the unit-sized GridHub are discussed. We prove that unit-sized GridHub is deadlock free. It is also livelock free under specific conditions. In Chapter 5, we show the performance of the unit-sized GridHub. In this chapter, we explain how the simulation model is built and how to conduct experiments. The experiment results are then displayed and discussed. In Chapter 6, we extend the unit-sized GridHub to handle non-unit-sized items. The performance of the extended system is also measured and discussed. In Chapter 7 we list contributions and make conclusions.

CHAPTER 2

LITERATURE REVIEW

2.1 Terminology

Conveyors and the items carried by them were usually considered a whole "conveyor module" in a grid-based system's description (Gue et al., 2014; Uludağ, 2014; Seibold, 2015). A module's state describes whether the conveyor module holds a box, or what kind (requested or nor) of box it holds. For example, a conveyor carries a box that is requested to be moved out, which also means that the conveyor module's state is "requested." In Figure 7, every module is in one of the three states: requested (in black), stored (in grey) or empty (in white).

Directions are used to describe stored item movements or locations in grid-based systems. Previous research (Gue et al., 2014; Uludağ, 2014; Seibold, 2015) used map directions (see Figure 7). In these systems, the North-south (NS) directions were the directions to move the requested items. The East-west (EW) directions were the directions to move stored items for the requested items. A schematic diagram of a grid-based system, such as Figure 7, is similar to a representation of a cellular automaton. Hence, grid-based systems can be modeled as cellular automatons. Von Neumann and Moore introduced two types of neighborhoods for a cellular automaton, respectively (Figure 8). Arbitrary neighborhoods are also defined to build the relationships among cells (Krühn et al., 2010) (Figure 8c).

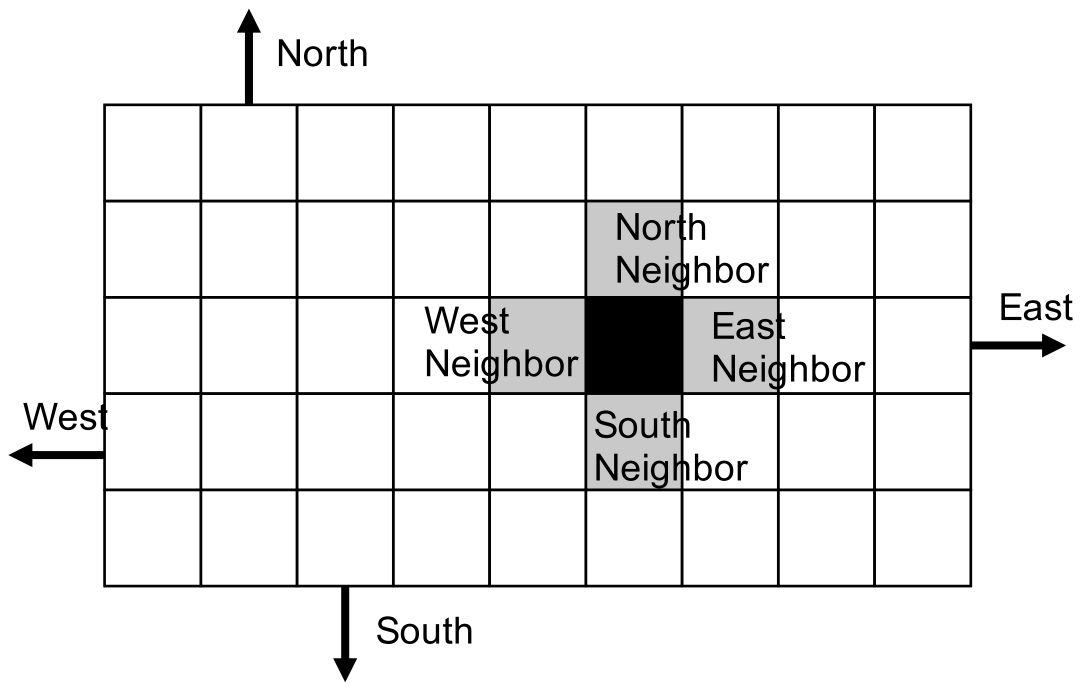
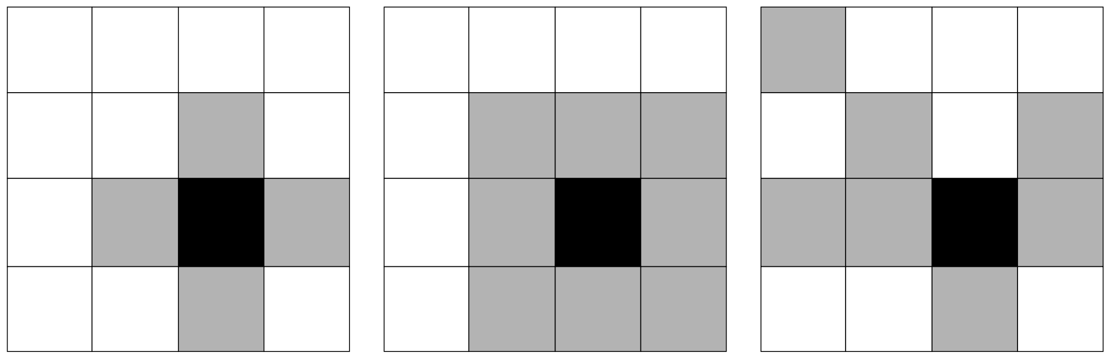


Figure 7: Schematic diagram of a grid-based system. The grey tiles are conveyors, and boxes are the yellow cubes. A white triangle on the boxes indicates the direction to move the requested boxes.



(a) Von Neumann. (b) Moore. (c) Arbitrary.

Figure 8: Neighborhood types of cellular automaton (Grey cells are the black cells' neighbors according to the neighborhood indicated).

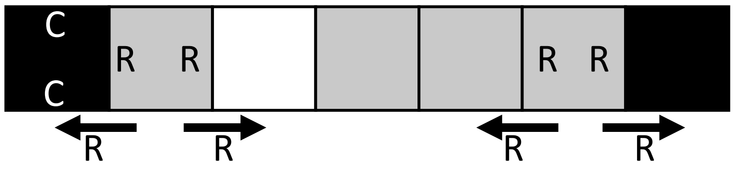
Existing grid-based systems used the Von Neumann neighborhood to establish connections among conveyor modules (Gue et al., 2014; Uludağ, 2014;

Seibold, 2015), and the names of a conveyor's neighbors are: the north neighbor, the south neighbor, the east neighbor and the west neighbor. The example of these neighbors are displayed for the black conveyor in Figure 7.

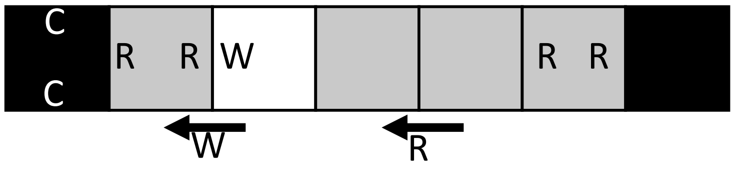
Communication activities among a conveyor and its neighbors are called messages passing or negotiation. All of the conveyor modules in a certain grid-based system have the same negotiation protocol. The conveyor modules follow the protocol to make decisions in the negotiation process.

We call the process of executing an entire negotiation protocol an "iteration" or a "cycle." For any stored item in a grid based system, it can move from a conveyor to the neighbor of this conveyor in one iteration. Hence, the number of iterations required to complete some task is usually used to measure time spent in a grid-based system. For example, if a requested item is moved during *3* iterations and stays on a conveyor for *2* iterations before it has been moved out of a system, we can use *5* iterations to measure the time spent moving this item out.

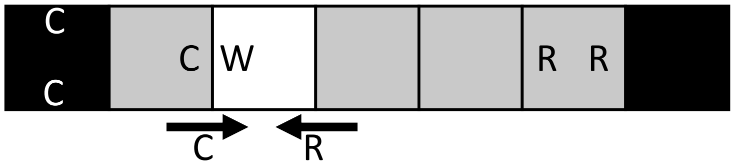
An iteration is divided into multiple phases or steps. For example, Gue et al. (2014) and Uludağ (2014) divided an iteration into three phases: assess, negotiate and convey. In the assess phase, an item's information is updated and evaluated; in the negotiate phase, conveyors communicate with their neighbors to generate the transportation instruction. in the convey phase, conveyors execute the transportation instructions. Furthermore, every phase can be divided into several steps. For example, the negotiate phase of the GridStore system was divided into a "north-south" negotiation step and an "east-west" negotiation step (Gue et al., 2014). To illustrate the negotiation process, an "east-west" negotiation example is displayed from top to bottom in Figure 9.



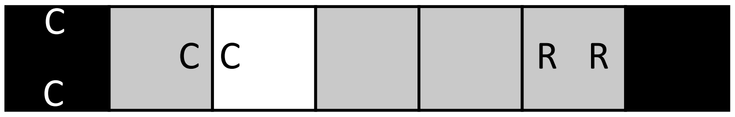
(a)



(b)



(c)



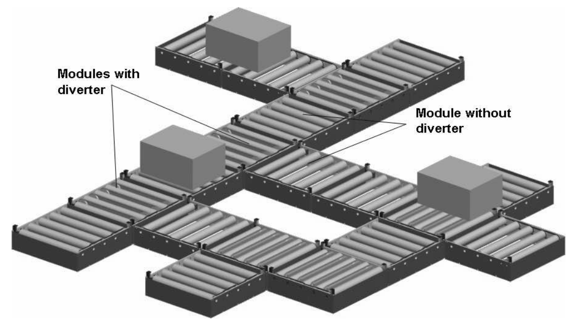
(d)

Figure 9: Example of negotiation among conveyors (Gue et al., 2014). In this case, the most left conveyor has committed to move its box to south. Its east neighbor, and the other conveyor are both marked with "R," which indicating they have "east-west" requests. In Figure 9a, the grey conveyors with "R R" marks are sending "request" messages to their east and west neighbors. In Figure 9b, while one request message is being passed, the other message responds with a "willing" message that is sent to the source of the request messages. In Figure 9c, a commit message is sent by replying to the will message. In Figure 9d, as a result of negotiation, the 2nd and 3rd conveyor (from left) commit to move their boxes to the right.

2.1.1 Existing grid-based systems

FlexConveyor FlexConveyor consists of identical square conveyors similar to the conventional conveyor systems (see Figure 10a). Every conveyor unit communicates with the components attached to it. Mayer (2009) presented the technical details of the FlexConveyor module and developed the control algorithms. The system layout is easy to change, and no modification to the control algorithms is required during the changes. Furthermore, most of the grid-based systems used or proposed the use of similar conveyors developed by Mayer (2009).

GridStore The GridStore (Gue et al., 2014) system was designed to store and retrieve unit-sized items. The items enter from the north side of the system, and depart on the south side (see Figure 10b). Both of the requested and replenishing items are only being moved to the south direction. The locations and sequences of releasing requested items are not specified. A study on the effects of failed conveyors in a GridStore system was conducted by Furmans et al. (2013), and the result confirmed the system's robustness.



(a) FlexConveyor system (Mayer, 2009).

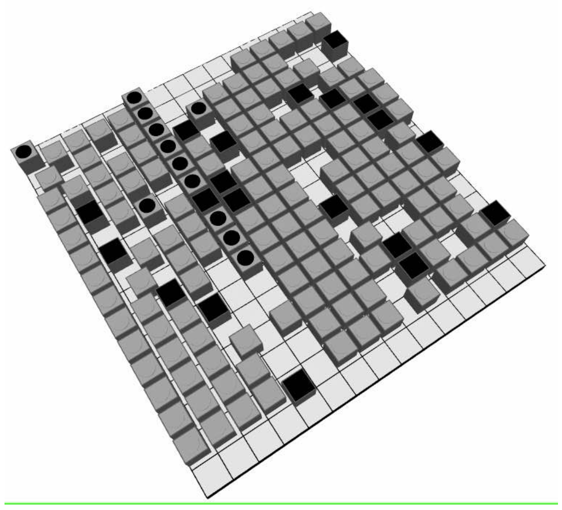
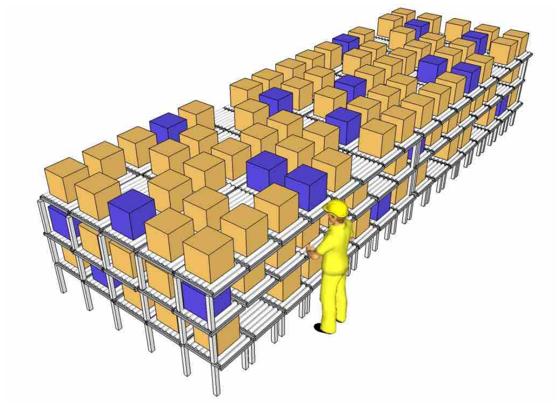

(b) GridStore (Gue, 2014).

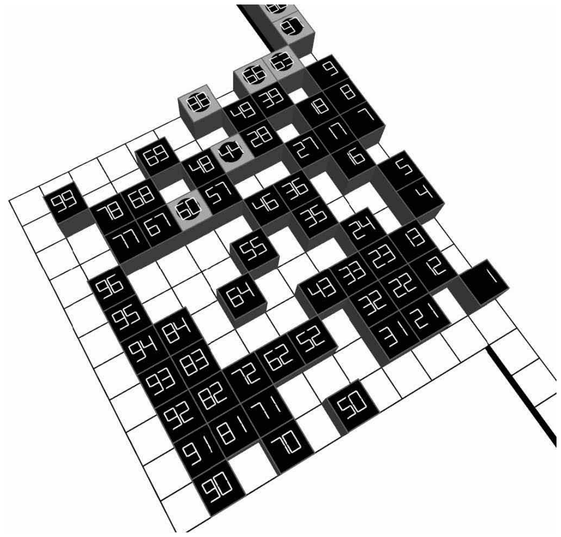
Figure 10: Existing grid-based systems (part 1).

GridSequence The GridSequence (Gue et al., 2012) system was developed based on the GridStore system to sequence cartons. When the system is running, a group of cartons assigned sequences entered the system from the north side in a random order. Cartons are only requested toward the south direction to the preset locations on the grid. After sequencing operations are finished, all cartons leave the system on the east edge in a preset sequence.

GridPick GridPick (Uludağ, 2014), which was designed as an order picking system, was another extension of the GridStore. The south side or both the south and north side can be used as picking faces. Boxes in this system are moved to the picking faces and are returned to the system after picking is finished. The orders could be picked at any locations on the picking faces. While the system is running, only two batches of boxes' picking sequences are set. For example, in Figure 11a, the boxes in blue are the requested boxes, and the worker can pick orders from these boxes when they are moved to the edges.



(a) GridPick (Uludağ, 2014).



(b) GridSequence (Gue et al., 2012).

Figure 11: Existing grid-based systems (part 2).

GridSorter GridSorter (Seibold et al., 2013) was designed for sorting boxes. It is not necessary for the system's shape to be rectangular. The entrances and exits of boxes are set at different locations on the edges of the grid. The GridSorter moves boxes to the preset departure locations along "reserved" paths that are found by negotiations among conveyors. (Seibold et al., 2013) adopted FlexConveyor controlling for GridSorter. Dominik et al. (2016) extended GridSorter to move non-unit-sized items. Seibold (2015) implemented "logical time," which is a concept related to distributed computer systems, in the decentralized control algorithms of the GridSorter. As a result, the absence of deadlock and livelock could be proven.

GridFlow AGV This system was designed to control AGVs in a storage space (Schwab, 2015). Hence, unlike other grid-based systems, the stored items are moved by AGVs instead of conveyors. In this system, goods are placed on pallets, and AGVs run under them (see Figure 12a).

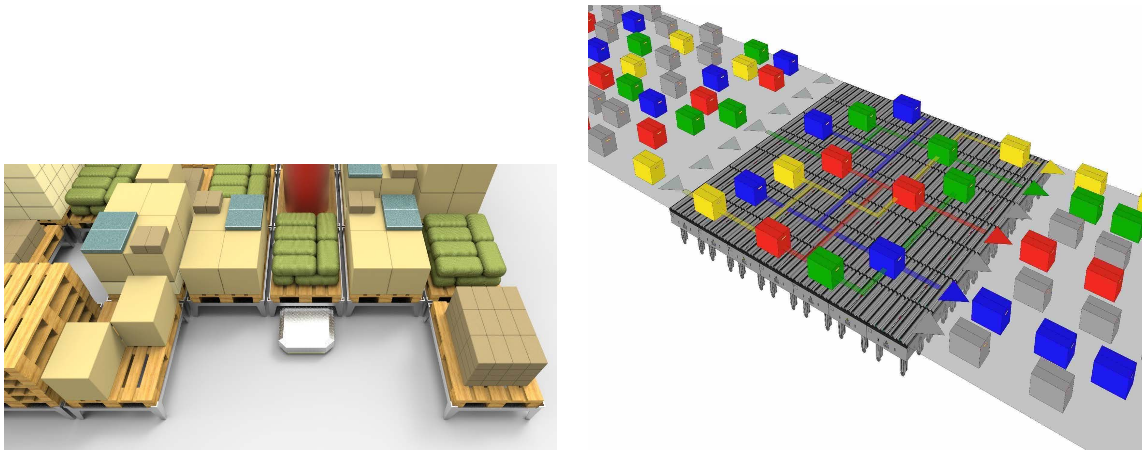

(a) GridFlow AGV (Schwab, 2015). (b) GridSorter (Seibold, 2015).

Figure 12: Existing grid-based systems (part 3).

2.1.2 Other material handling systems with similar features

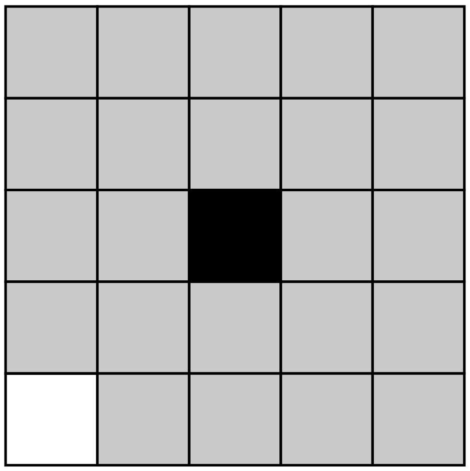
Some other storage systems have features similar to grid-based systems, such as high storage density or decentralized control.

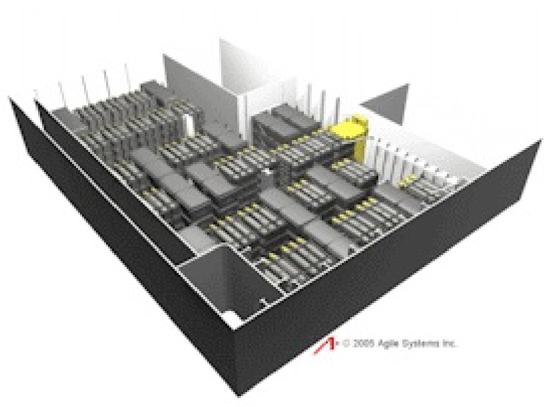
Puzzle-based storage systems A puzzle-based storage system is a type of high-density storage system. The layout of a puzzle-based system is similar to the layout of a N-puzzle game (see Figure 13a). Gue and Kim (2007) described an order retrieval method for a puzzle-based storage system. In the puzzle-based system, an empty space is called an "escort." In the retrieval process, the escort must be moved to the location that is adjacent to the requested item in order for the item to be moved.

Kota et al. (2015) conducted an analytical study about the retrieval time of a puzzle-based storage system. In the research, the authors assumed the locations of escorts are uncertain, and accounted the time to “move” escorts to desired locations into the retrieval time. Mirzaei et al. (2014) modeled the retrieval process of puzzle-based storage systems. They considered a square, puzzle-based storage area with different dimensions and calculated the average time for moving one or two loads out of the system. Zaerpour et al. (2010) presented an optimal configuration for a puzzle-based storage system, which was later explored in the fresh food industry (Zaerpour et al., 2015). The policy for operating this system was divided into “dedicated lane” and “shared lane.” The dedicated lane system was similar to flow racks because the same SKUs were stored in the same lanes. The shared lane policy did not require the same SKU in the same lanes, so space utilization was increased. However, the authors pointed out that to move the requested items out, the process of moving the items in front of the requested item increased the complexity of the operation.

Yalcin et al. (2019) developed methods to retrieve requested items from a puzzle based storage system with multiple escorts. The method organized escorts to different locations in order to perform the requested item’s non-stop movement.

Shirazi (2018) implemented decentralized control in a puzzle-based storage system. The author claims that the system can move up to three requested items simultaneously and release them to any specific locations on any system border. Shirazi also mentions deadlock, and provides a method to solve deadlock when deadlock occurs.


(a) Schematic of a puzzle-based storage system.



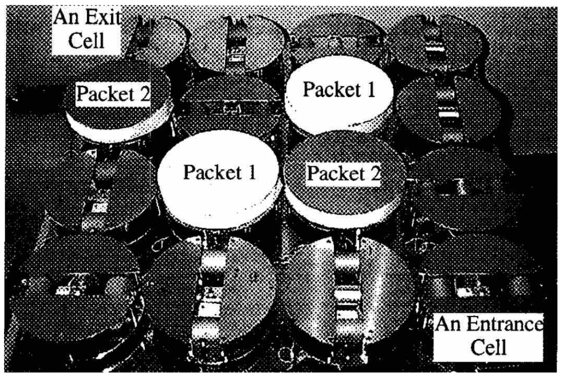
(b) An example of proposed puzzle-based system (Agile System Inc., 2017).

Figure 13: Puzzle based storage systems. In the left figure, the black square indicates a requested item, and lower left corner is the I/O place.

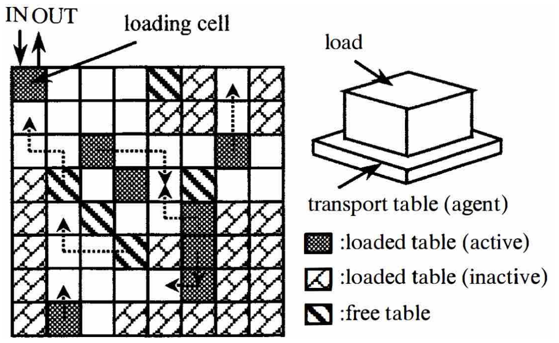
Cellular warehouse Sakao et al. (1996) developed an automatic material handling system, the earliest grid-like system known so far (see Figure 14a). The system consists of a group of homogeneous "cells." The cell is a machine with an individual CPU that can communicate with and move items to its "adjoining" cells. The exits and entrances are assigned to some cells that can be at any border of the system. The cells communicate through message passing. In every cell's controller, a message buffer is used to store messages received. The controller of a cell selects the most important messages from the buffer to read and initiates actions based on pre-loaded rules. Tests of the system were conducted by simulation and prototypes. The system features easy re-configuration, and possibilities of handling broken cells were confirmed via these tests.

Hama et al. (2002) developed a distributed control method to coordinate autonomous tables (see Figure 14b) in a cellular-like field for desired configurations. These tables can communicate with their neighbors and enter the state of active or inactive. Each of these tables are controlled by its own behavior functions, and an Artificial Neural Nework (ANN) is employed to generate the behavior functions. The training of the tables starts with solving smaller N-puzzle problems to bigger

N-puzzle problems.



(a) Cellular warehouse proposed by Sakao et al. (1996).



(b) Cellular Warehouse proposed by Hama et al. (2002).

Figure 14: Cellular warehouses.

Flexible transportation system Fukuda et al. (2000) and Takagawa et al. (2003) worked on systems called "flexible transportation system," which also features decentralized control. The systems' layouts are similar to grid-based systems, and Takagawa et al. stated that some of the modules locations could be changed autonomously according to tasks completed. The modules in these systems can communicate with each other through messages. A major difference is that the authors applied methods of machine learning to guide the units in routing.

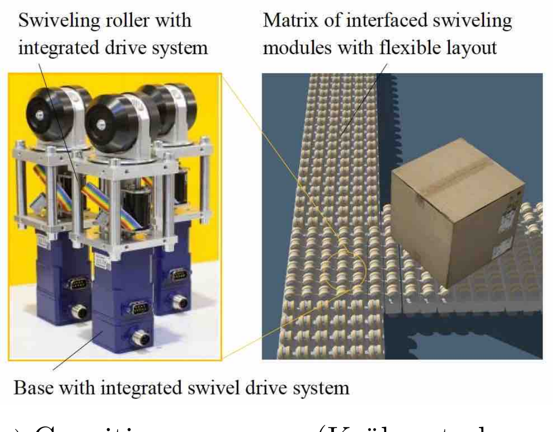
Modular warehouse Sittivijan (2015) developed a storage system called "modular warehouse" under hybrid controlling. Both centralized and decentralized methods were used by the author. The inside layout of a modular warehouse is similar to a grid-based system. There are several entrances or exits to the rectangle warehouse, and the stored items are moved by results of negotiations.

Cognitive conveyor A cognitive conveyor system was designed to move and sort items in different dimensions (Overmeyer et al., 2010). The size of a cognitive conveyor is small, and these conveyors are omnidirectional (see Figure 15a). The design of cognitive conveyor was completed by Overmeyer et al. (2010). Krühn et al. (2013) described a control method by modeling the system as a Cellular

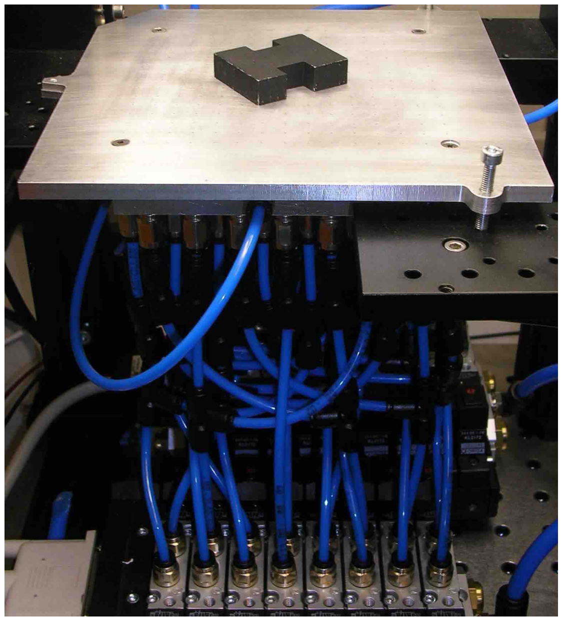
Automaton. Routes of the items are usually planned prior to actual movements through a decentralized negotiation process (Krühn et al., 2013).

Firvida et al. (2018) introduces an omnidirectional route planning method in a conveyor system, similar to the cognitive conveyor. The author also states that more work such as deadlock avoid methods were under development.

Smart surface Another study called smart surface was presented by Boutoustous et al. (2010) to sort items based on their shapes. This system consists of a rectangle board and multiple small scaled control units (see Figure 15b). Each of these units has its own sensor and actuator. The sensor can detect whether the control unit is pressed by the parts. The system is modeled as a Cellular Automaton, and the shape of the part is recognized by the control unit under a decentralized method. After shape recognition, some of the actuators lower one edge of the board, so the item can slide to this edge. For example, to move the "H" shaped item to the left side in Figure 15b, actuators on the left side open valves and lowers the board on the left side, then the item is slid to the left, and this completes the sorting operation.


Swiveling roller with integrated drive system
Matrix of interfaced swiveling modules with flexible layout
Base with integrated swivel drive system

(a) Cognitive conveyors (Krühn et al., 2010).



(b) Smart surface (Boutoustous et al., 2010).

Figure 15: Other material handling systems that are similar to grid-based systems.

The existing Grid-Based systems are also compared (see Table 1), and we summarized according to their functions.

TABLE 1: Comparison of current Grid-Based systems.

System	Functions	Release at Specific location	Release in specific sequence	Item size
GridStore	Store, retrieve	no	no	unit
GridSequence	Sequence, sort	yes	yes	unit
GridPick	Store, retrieve	no	yes	unit
GridSorter	Sort	yes	no	multiple
GridFlow AGV	Store, sort	yes	no	unit
Cognitive conveyors	Sort	yes	no	multiple

2.2 The control methods of grid-based systems

The grid-based systems are under decentralized control. First, we introduce the concept of control and compare the differences between the centralized and decentralized control. Second, since there are two distinct method of controlling a grid-based system, we explain these methods. Then, because deadlock and livelock must be considered when designing a grid-based system, we explain methods to solve deadlock and livelock for grid-based systems. Finally, we review the existing methods to handle non-unit-sized items in grid-based or similar systems.

The state of a system is the information used to describe the system at a fixed time point. For a grid-based system, the system state is the configuration of the system. For example, Figure 7 shows a system configuration that includes items' locations and conveyor module states (empty, store or requested), and this configuration is one of the system's states.

Control is a process where a system takes inputs and generates outputs as desired behaviors according to some rules (Cassandras and Lafortune, 2008). In the control process, several states transition may occur, and these transitions are called events. A system's states are usually used as the inputs and outputs (Cassandras and Lafortune, 2008). A state transition diagram is a useful tool to represent the control process (Figure 16). Controlling a grid-based system is taking a state (configuration or layout) as an input, and generating transportation orders based on a conveyors rule set to move some items to their targets.

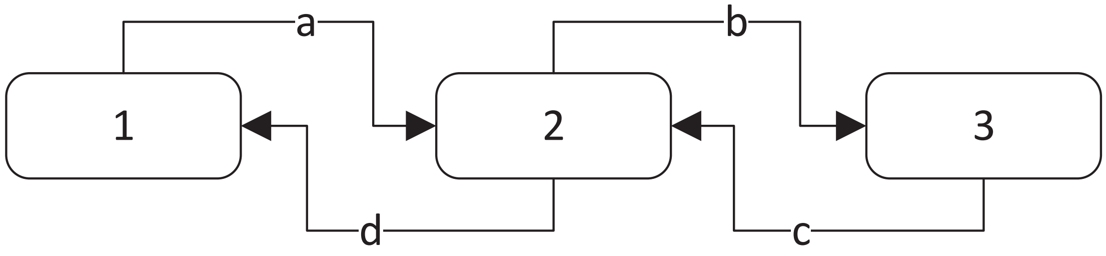


Figure 16: Example of state transition diagram. The nodes represent states; arches or edges indicate the state transition paths; the letters on the arches are the events which trigger the transitions.

2.2.1 Control architecture and system modeling

The control architecture was divided into three classes: centralized, hierarchical and decentralized. Mayer (2009) gave an intuitive summary of these control architectures. Centralized control has one supervisor or controller; hence all of the inputs for the control process are collected by this controller, and outputs are generated from this controller or supervisor. However, decentralized control architecture has more than one supervisor or controller (Cassandras and Lafortune, 2008).

Another architecture, the "distributed control," was referred to by many researchers, such as Scattolini (2009). Scattolini (2009) also gave the differences of distributed and decentralized control. Based on the description, the grid-based systems are under distributed control. Because these two terms are always mixed up by the community today, we use “decentralized control” in our research to maintain consistency.

For a centralized controlled system, if the control rules are designed properly, the output would be desired. As the input gets more and more complex, the controller of a centralized system takes longer to generate output. For the individual controller of a decentralized system, it does not need to collect all of the input information; hence, its computation load is reduced. However, in the case of lacking necessary information, desired output is hard to obtain. Furthermore, a decentralized system is more susceptible to deadlock or livelock states.

The control architectures described above are in Figure 17

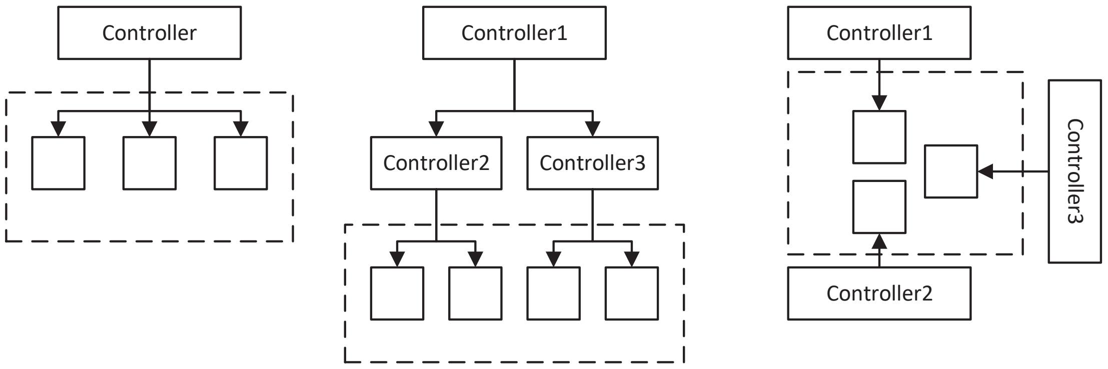


Figure 17: Control architectures (The figures are drawn by descriptions of (Mayer, 2009), (Cassandras and Lafortune, 2008)), and Scattolini (2009). From left to right: centralized control architecture, hierarchical control architecture, and decentralized control architecture.

Agent-based modeling is a system modeling tool. Jennings et al. (1998) defined an agent as a computer system featured “situatedness,” “autonomy” and “flexibility,” meaning that an agent can automatically interact with both the environment and other agents (Jennings et al., 1998). The environment was the other object in the system in addition to an agent, such as, other agents, physical environment, etc. (Wooldridge and Jennings, 1995). If a system is modeled as an agent, the system can be called agent-based system, or a multi-agent system, when it consists of more than one agent (Jennings et al., 1998).

The agent-based system is an intuitive tool to model the grid-based system. Hence, Uludağ (2014) and Seibold (2015) modeled the grid-based system as a multi-agent system and obtained the estimated system performance. In these models, every conveyor module was modeled as an agent. the other objects such as the neighbors of a conveyor were considered the environment.

2.2.2 Methods of designing control algorithm

All of the existing grid-based systems are distinguished according to the method of designing control algorithms. Schwab (2015) gave a taxonomy of the grid-based systems on system control perspective; Seibold (2015) divided two categories of grid-based systems based the routing methods. We update the summary of the design methods as follows.

The path reservation method The objective of the path reservation is to find moving paths for items by communicating with conveyors. FlexConveyor (Mayer, 2009), GridSorter (Seibold, 2015) and Cognitive Conveyor (Krühn et al., 2013) used this method. Routing algorithms, such as the IDA* algorithm, were adapted to the decentralized system (Seibold, 2015). Messages are categorized as "path request messages" and "confirmation messages." The path request messages are used to find a moving path from start conveyors to destination conveyors; confirmation messages are sent from the destination conveyors to reserve transportation paths. The route planned by these methods is always toward the items' targets.

Seibold (2015) summarized works which used the path reservation method into two categories, "time independent" and "time window based." In works which used the first category, modules were reserved for items to pass through. When an item had been passed, the module was available again. For the second category, the future time of each conveyor was divided into several windows, and each item only occupied one time window.

The virtual aisle method This method is to find an empty conveyor or make an aisle and move the requested items forward. Gue and Furmans (2011) introduced this method. This work was extended to the GridStore (Gue et al., 2014) system later. This method is similar to the method of moving "escort" in a puzzle-based storage system (Gue and Kim, 2007). In a grid-based system, to move a requested item forward, an empty conveyor (escort) has to be in front of the requested item. There are two cases to move a box forward (Figure 18 to 19).

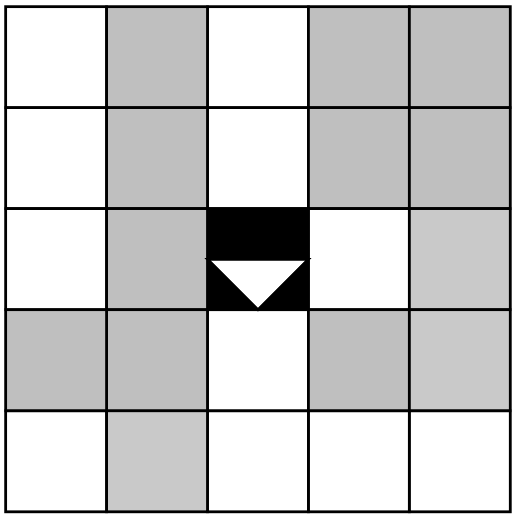


Figure 18: Case 1 of moving the requested items using the virtual aisle method: an empty conveyor is in front of a requested conveyor module. The movement can be completed in one iteration via north-south negotiation.

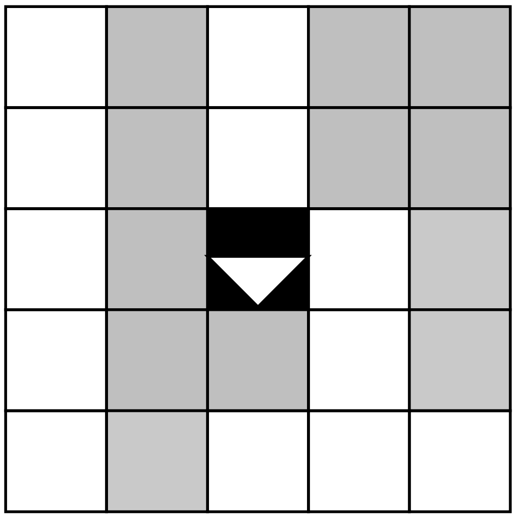


Figure 19: Case 2 of moving the requested items using the virtual aisle method: the empty conveyor is not directly face the requested module. East-west negotiations are used to "exchange" the empty conveyor's location in order to transform this case into the first case. Hence, at least two iterations are needed to complete the movements, and it is harder to finish these movements than the first one.

Routes selection is simple in the virtual aisle method. The shortest distance to move an item in a grid is the Manhattan Distance (Figure 20).

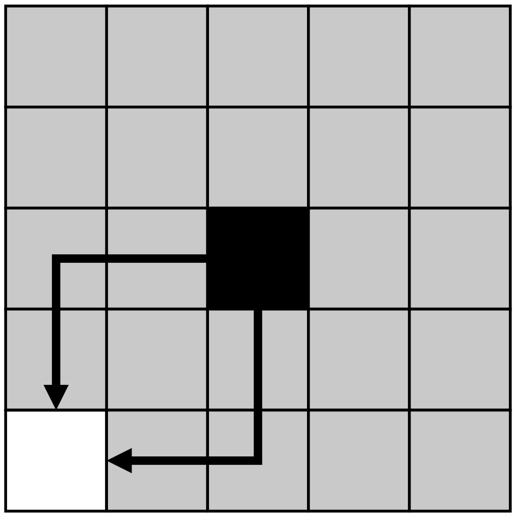


Figure 20: The shortest routes of moving requested items. Move a requested item along the Manhattan route. Both of the paths marked are the shortest moving paths. Suppose the target of the requested item is the lower left corner of the grid.

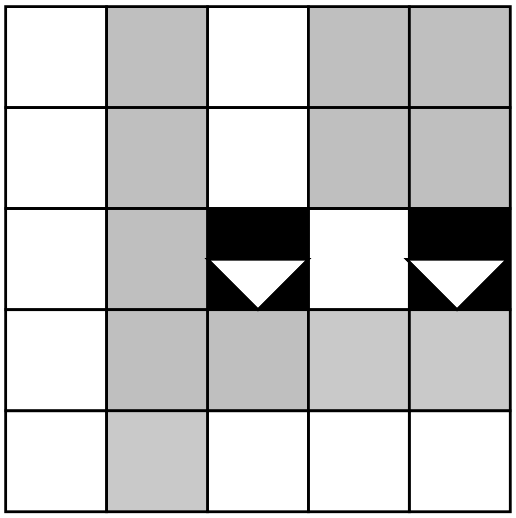


Figure 21: Scenario of waiting. One of the above requested items has to wait for the other one to move away. Suppose the target of the requested item is the lower left corner of the grid.

In summary, the path reservation method is offline path planning (Shiller, 2015) because the path is fixed before physical movements start. In contrast, the virtual aisle method is on-line path planning (Shiller, 2015) because there is no transportation path planned before the item starts to move. These methods both have advantages.

- In the system applying the path reservation method, the transfer time of an item is known as soon as its route has been fixed. The length or transportation time of this route is optimal or near optimal. Furthermore, Seibold (2015) pointed out that for a grid-based system with narrowed shape, such as the FlexConveyor, the path reservation method could find actual routes in a resource limited scenario. The other advantage of this method is that the shape of the system does not need to be rectangular. In other words, the routing method fits in multiple layouts.
- The virtual aisle method allows non-requested items to stay in a system for a long period; hence more space is utilized than in the path reservation method. The other advantage of the virtual aisle method is that the system can process external requests instantly. There is no waiting time for planning paths.

2.2.3 Deadlock and livelock in grid-based systems

Deadlock and livelock are crucial issues that have to be avoided in grid-based systems. Deadlock occurs when a system can not transition to other states (Cassandras and Lafortune, 2008) (see Figure 22 as an example).

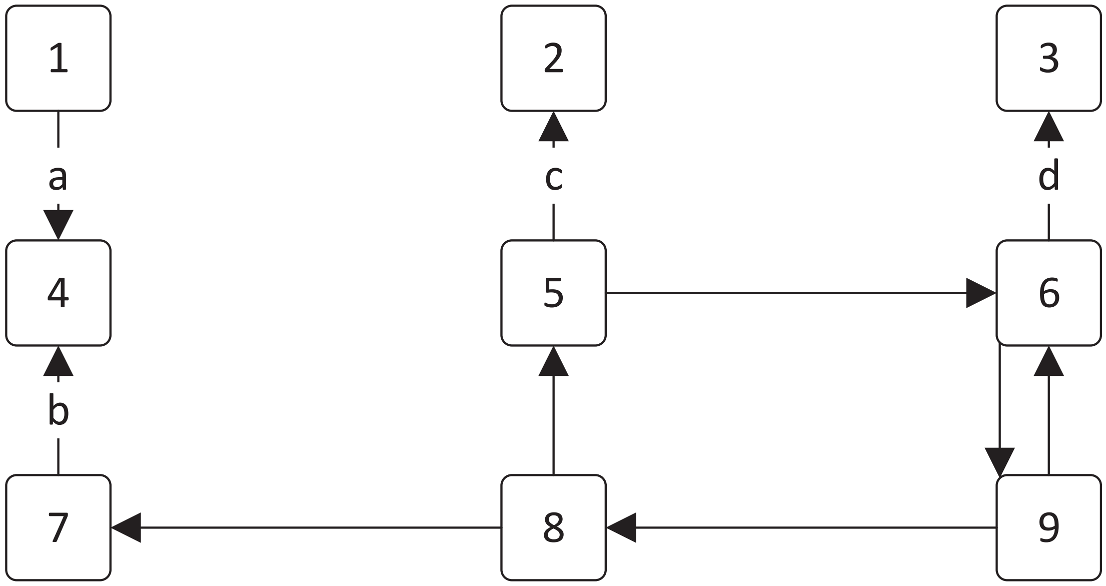


Figure 22: Deadlock: a system cannot change to any other states. State *2*, *3* and *4* are deadlock states.

In a deadlocked grid-based system, it is impossible to move items, so the transfer task can not be finished (Figure 23).

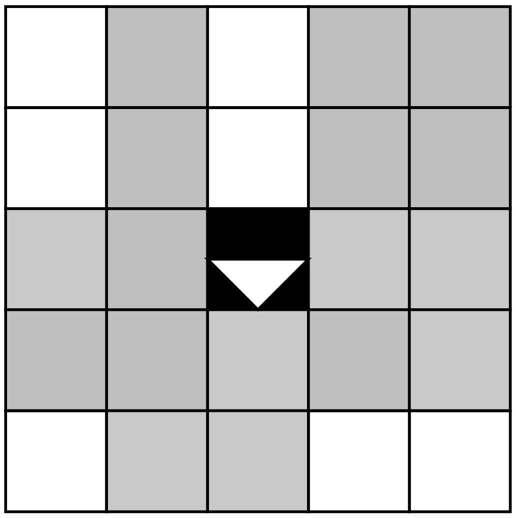


Figure 23: Example of deadlock in a grid-based system. Suppose the black module's box is requested to move south. According to existing methods of negotiation that follow the virtual aisle method, there is no way to move the requested box forward. Hence, this is a deadlock case in a grid-based system.

Livelock occurs when a system continually changes among a set of fixed states in a fixed pattern, and it is impossible for the system to transition to any other state (Cassandras and Lafortune, 2008) (see Figure 24).

$$\begin{tikzcd}
\boxed{1} \arrow[d] & \boxed{2} \arrow[l] & \boxed{3} \arrow[d, shift right] \\
\boxed{4} \arrow[d] & \boxed{5} \arrow[u] & \boxed{6} \arrow[u, shift right] \arrow[d, shift right] \\
\boxed{7} \arrow[r] & \boxed{8} \arrow[u] & \boxed{9} \arrow[l] \arrow[u, shift right]
\end{tikzcd}$$

Figure 24: Livelock: a system changes among certain states forever. The system in this figure keeps transitioning among states $8-5-2-1-4-7$ and never transition to any other states.

When a grid-based system is in a livelock state, items are moving among certain conveyors in a sequence, and these movements never progress the transfer process (Figure 25).

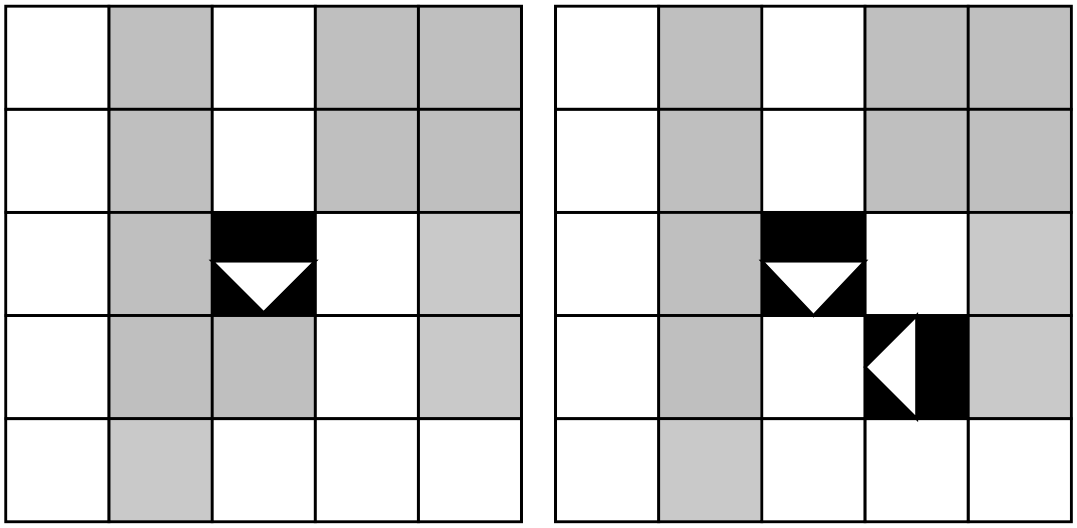

(a) Layout 1 (b) Layout 2

Figure 25: Examples of livelock in a grid-based system. The system switches layout between the above two, and the requested items never make progress.

From a computer science perspective, Coffman et al. (1971) concluded that four conditions, which are (1) mutually exclusive, (2) hold and wait, (3) no preemption, (4)circular wait, have to be met together to cause deadlock in computer systems; Mayer (2009) pointed out that in a grid-based system:

- Each conveyor can only hold at most one item (the "mutually exclusive" condition is met).
- Each conveyor has to wait for another conveyor to accept the holding item (the "hold and wait," and the "no preemption" conditions are met).

The only way to prevent deadlock is to prevent the system from meeting the last condition.

Seibold (2015) defined system liveness as all requested items being moved to their target locations, which can be either inside or outside of the system. In other words, a grid-based system's liveness is absence of deadlock and livelock.

2.2.3.1 Review of methods to handle deadlock and livelock in grid-based systems

Seibold (2015) summarized four strategies to handle deadlock and livelock: ignore, detect-solve, avoid and prevent. According to the state transition diagram in Figure 22 and 24, the ideas to address deadlock and livelock in grid-based systems can also be divided into two categories. The first category is to prevent the system entering the deadlock or livelock states; the second category is to detect and solve these problems when encountered.

Prevention If explicit conditions for deadlock and livelock free are known, control methods can be designed to make the system meet these conditions (Figure 26). Mayer (2009), Seibold (2015), Gue et al. (2014), and Uludağ (2014) used this idea to solve deadlock and livelock.

Mayer (2009) used a method called deadlock tokens to prevent overlapped reservations. In a circle layout of the FlexConveyor, when a conveyor is available to receive an item, it sends a “deadlock token” to the next conveyor in the planned transportation route. Then, the available conveyor becomes blocked and ignored any other request for entering. This process is repeated by all conveyors up to the destination of an item in order to locate an available path. In complex layouts of the FlexConveyor, such as overlapped circles, checking for transfer requests from the opposite or perpendicular directions on the overlapped conveyors, while reserving routes, is necessary to avoid deadlock.

In the GridSorter system, the process of moving an item among multiple conveyors requires multiple resources in a distributed computing system. Seibold (2015) implemented the concept of logical time in the GridSorter’s routing. When conveyors are reserving paths for moving items, different items on the same conveyor have a unique “logical time stamp,” and the logical clock of each conveyor only advances when a transfer task is completed. The author proved that the system is deadlock free. Furthermore, since the path reservation method never generates backtrack paths, livelock is impossible in GridSorter.

In the GridStore system, all items are requested in a fixed direction (the south side of the system), and task assignments that requested any item to the north did not exist; hence, livelock never occurred in this system. The authors also proved that if there was at least one empty conveyor in every row of conveyors, the system is deadlock free (Gue et al., 2014). This is because the requested conveyor modules can always find empty space in every row to move items forward, even if they need to wait for other requested modules. Hence, to prevent a deadlock state, the system has to leave at least one empty module in each row. To accomplish this objective, the system assigns every replenishing item a target row, which equals a departed item's start row. For example, if a newly departed item's start row is 4, then a newly arrived item assigns its target row to 4, and this item has to move to row 4. As a result, the number of items assigned to the row is constant.

By the same principle, the GridSequence system also keeps an empty conveyor assigned to each row to avoid deadlock (see Figure 11b). There is no opposite request direction, and the items' routes are always moving toward their targets directly, so livelock is impossible.

The GridPick system used a method called "balancing" to reach the states that satisfied the deadlock free conditions defined for GridStore. The balancing idea is to move an item between two adjacent rows, while a requested item is moving between these two rows. The purpose is to keep the number of items in each row constant, which is the requirement of remaining deadlock free in GridStore. The north requested items are also defined by having higher priorities than the south requested items; hence, no south moving items cannot block the north moving items. The author modeled the system by Petri Nets, and proved that the GridPick system was deadlock free with dimensions (in unit of conveyors) of: 3×3, 3×4, 3×5, 4×3, 5×3 and 4×4. The additional condition is that the number of requested items in the system had to be less than or equal to $2c - 1$ (c is the number of columns in the GridPick system) (Uludağ, 2014). For preventing livelock, though replenishing items move in the opposite direction of requested items, the routes of the items always move to their targets directly. Hence, livelock is impossible.

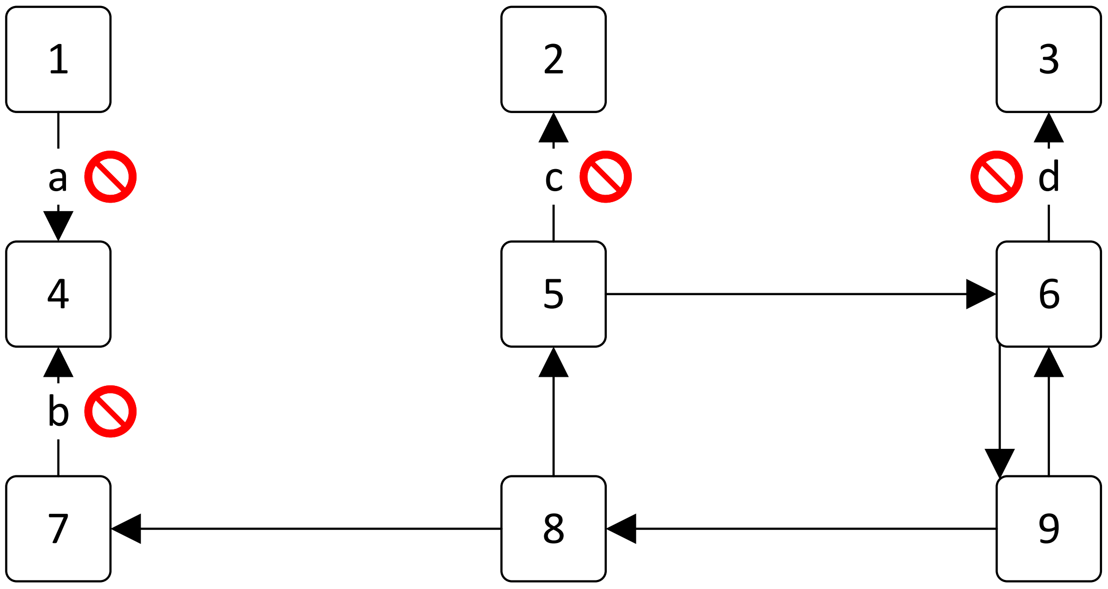


Figure 26: Deadlock and livelock solution 1: prevent entering some states. If events a, b, c, and d are known to cause deadlock, deadlock can be solved by preventing these events from occurring.

Detect and solve If the explicit conditions of deadlock or livelock free are not known, methods to detect them are necessary, and the methods to solve them are also required to keep the system alive (Figure 27).

GridFlow AGV used this idea to handle deadlock and livelock cases. Deadlock and livelock are detected, and then they are solved based on the rankings of AGVs. The deadlock cases are defined as cases where the AGVs blocked each other. To solve the deadlock, these cases are detected first. The highest ranking AGV in a blocked area sends messages to lower ranking AGVs to ask them wait in their positions or backup to create space for it. Livelock had two cases. The first case is an AGV carrying a load between two locations; the second case is an empty AGV moving between two locations. Both of these cases also need to be detected first, then the AGVs with lower ranks are asked to wait until the highest ranking AGV move away from the livelock area.

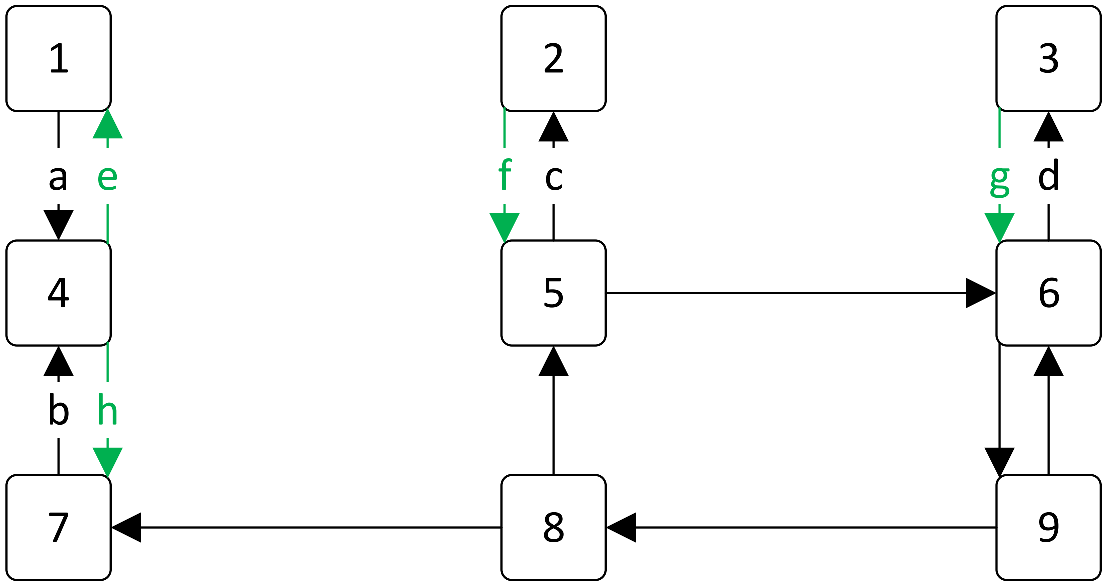


Figure 27: Deadlock and livelock solution 2: exit the deadlock or livelock states after detect. When the system is in deadlock states *2*, *3* and *4*, events, such as *e*, *f*, *g* and *h* are triggered to transition the system out of these states.

2.2.3.2 Deadlock and livelock in systems using on-line path planning

All of the existing on-line path planning grid-based systems require empty space to move requested item to their targets. All of these systems use message passing as the method to arrange movements for all items. Based on Coffman et al.'s definitions and the explanations of (Mayer, 2009) and Seibold (2015), deadlock in a grid-based system that uses on-line path planning, is caused by failure of find empty modules. Livelock in these systems is not encountered thus far.

In GridHub, requested items can be moved all four directions, which means that the system has more complex states, and message passing is more difficult. There are also more possibilities for livelock. Algorithms of the unit-sized GridHub make deadlock impossible, and these algorithms are described in Chapter 3. Proof that the system is deadlock free, and the discussion of livelock are detailed in Chapter 4.

2.2.4 Handling non unit-sized items in grid-based systems

Handling non-unit-sized items is one of our research objectives. The key is to develop a method of organizing conveyors handling an individual item. Methods of grouping conveyors or organizing the conveyors' communication activities have been developed in the above systems.

Organizing conveyors by tree structure Dominik et al. (2016) developed a non-unit-sized GridSorter. A tree structure was formed to build relationships among conveyors that were occupied by the same item (Figure 28). In the path reserving process, which inherited the path reserve method of Seibold et al., communication among the coordinator conveyor and helper conveyors was based on the tree structure.

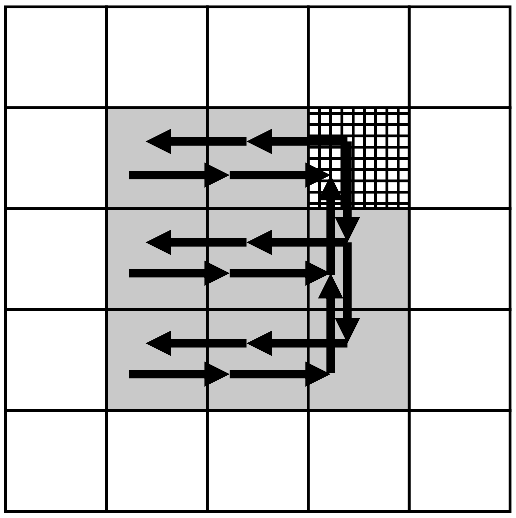


Figure 28: Grouping conveyors by building a direct tree ((Dominik et al., 2016)). The upper right conveyor (the shaded one) is set as a "coordinator module;" the other conveyors with the same item are set to "helper" modules. When the conveyors are looking for paths, request messages are sent from the coordinator conveyor to helper conveyor (right to left, or up to down). The replied messages are sent in the opposite direction.

Recognize part by add matrix Boutoustous et al. (2010) described a method to recognize an item's shape for smart surface (Figure 29).

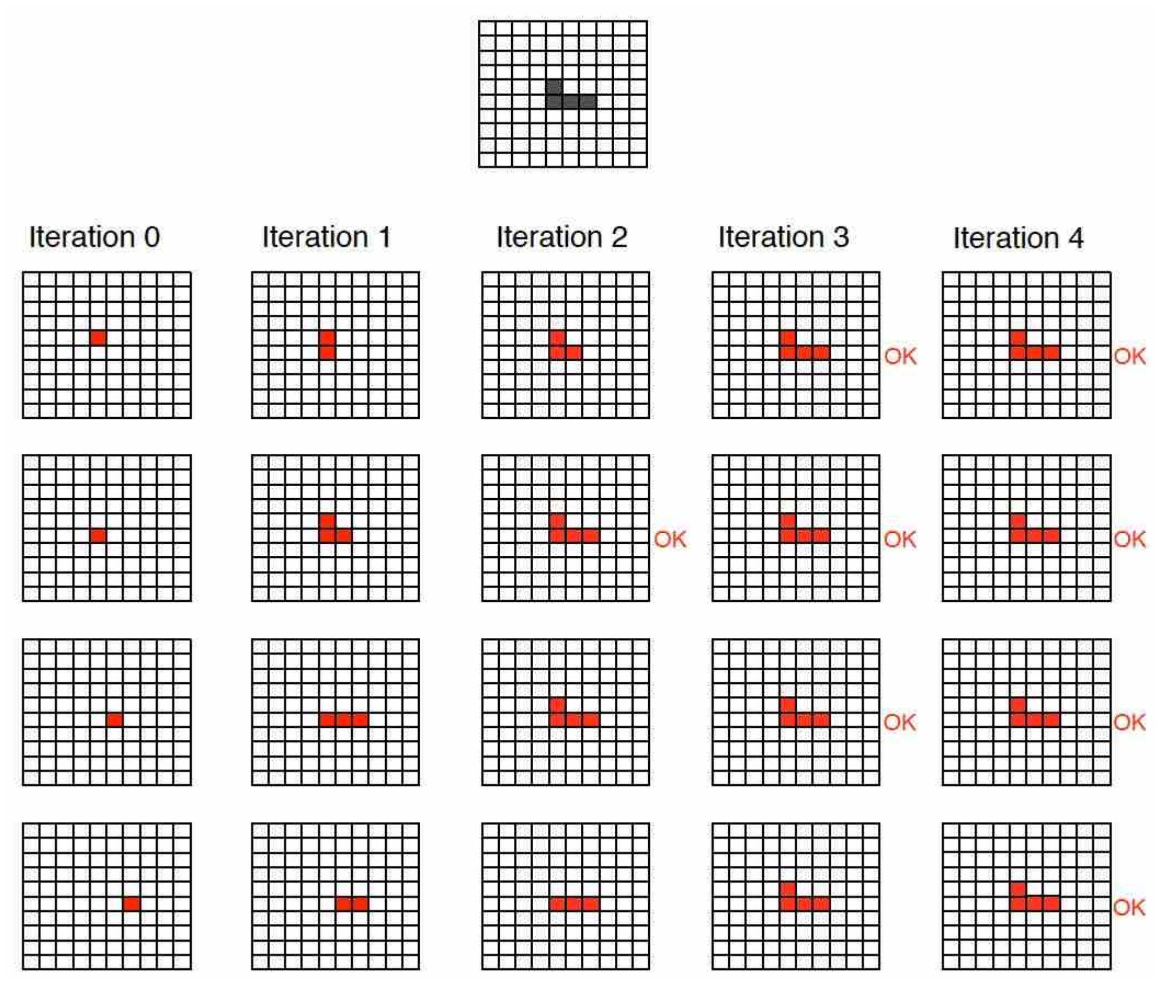


Figure 29: Process of shape detected by matrix adding (Boutoustous et al., 2010). First, each of the units in the smart surface generate a zero matrix, which has the same dimension as the smart surface. If a sensor detects pressure, it changes the element of the zero matrix, which matches its actual location, from 0 to *1* (The red square represents that the elements in the matrices are equal to *1*). Second, units of the smart surface send the matrices to each other. Every unit adds the received matrices to obtain a binary matrix. After all of the matrices are shared, every conveyor knows the shape of the item on the board.

Group conveyors according to neighborhoods Krühn et al. (2010) modeled the system by cellular automaton to group the omnidirectional conveyors. Every conveyor in the system was considered a cell, and it was set to have both the Von Neumann and Moore neighborhoods. Every conveyor has sensors to detect whether it is occupied; if yes, the item's ID is known. Each of the occupied conveyors uses the item's ID to communicate with its neighbors. First, Von Neumann neighbors explore. If the Von Neumann neighbors have no items detected during the exploration process, then the Moore neighbors communicate to build relationships.

In summary, though Dominik et al. (2016) and Krühn et al. (2010) developed a method to organize negotiations of conveyors, this method was based on the path reservation method and few details were described. The other work only provided a method to recognize an item. In a GridHub, the non-unit-sized items must be moved in all four directions. More detailed and sophisticated methods are demanded in order to organize conveyors.

2.3 Conclude the research gap

The research activities of grid-based systems are intensive. Several grid-based systems with different functions have been developed in the last decade. However, the following gaps exist the research.

- The existing grid-based systems were designed for specific tasks, and no system has the capability to finish storing, sorting, sequencing and retrieving operations together.
- No method has been developed to transfer requested items in all four cardinal directions for the virtual aisle method (on-line).
- Little to no literature exists about handling non-unit-sized items in grid-based systems.
- Deadlock and livelock conditions for the more complex grid-based systems, controlled by the online path planning method, have not been explored.

We address these gaps in the following chapters.

CHAPTER 3

DESCRIPTION OF UNIT-SIZED GRIDHUB

3.1 GridHub system description

Consider a grid of unit-sized conveyors, each capable of conveying in four cardinal directions (Figure 30). We assume each conveyor c_i ($i \in \mathbb{N}$, such as c_1 in Figure 30) knows its location in the grid.

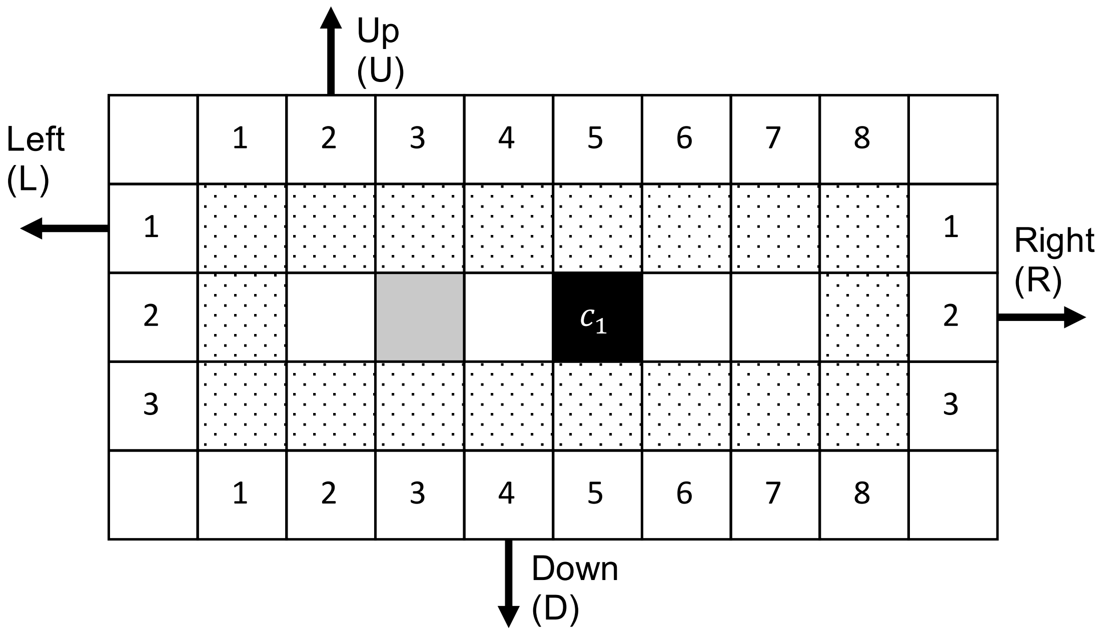


Figure 30: GridHub system layout. Directions of movements or locations in GridHub are "up," "down," "left," and "right" from the reader's point of view. Conveyors with numbers are *gates*, which are used to receive and release boxes, but not to store them. The system *edges* consist of these gates. The shaded conveyors, which are inside the gates, make up the system *border*.

Departure information of a box indicates when and where to release a box by setting the box's *departure edge*, *departure gate*, and *departure sequence*. Departure sequence refers to the order of departure, not to the time. For example, a group of boxes are set leave at a specific gate in a sequence. In this group, a box with departure sequence 3 may depart only after the box with departure sequence 2, but it may depart at any time thereafter. When departure information is assigned to a box, it becomes a *working box*. The conveyor associated with the final destination is the *target*. If it is not possible to move a working box to its target in a straight path, we define one or more *intermediate targets* at the corners of a path.

An *expected path* of a working box is the shortest path to move the box from its current conveyor through its intermediate targets to the final destination. In a grid, there can be many shortest paths, so we choose the two having one turn, unless there is a direct path without turns.

Moving a box to its target or intermediate target requires a *transfer task* with the box's target or intermediate targets and the direction of movement toward the next target. If other boxes need to be moved to their neighbors in order to move a working box, a *temporary transfer task* is assigned to each interfering box. A temporary transfer task includes the moving direction and the neighbor to which the box will be moved. The temporary transfer task is complete after the box is moved. An *active box* or *temporary active box* has a transfer or temporary transfer task assigned. The next conveyor to which an active box will be moved is the *immediate destination*. To summarize, only a working box has a target or intermediate target(s); and only an active box or temporary active box has an immediate destination. As we will see, a working box can be active or inactive (e.g., it might go inactive to avoid a deadlock). Figure 31 uses an example to illustrate the above concepts.

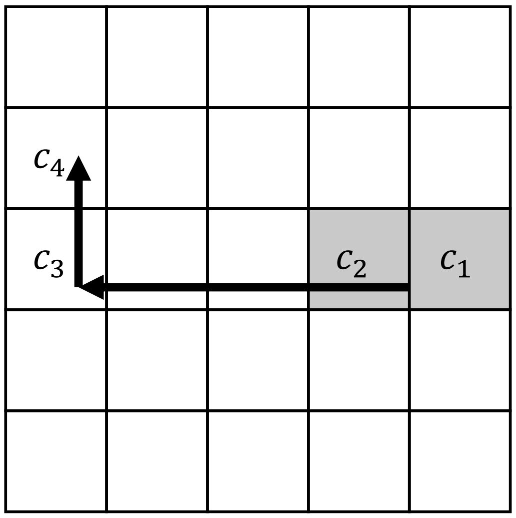


Figure 31: Target, intermediate targets and immediate destination of a box. Let the departure edge of c_1's box be the left edge. Its departure gate is the left neighbor of c_4, so the target of c_1 is c_4.
Because c_1 cannot move its box to the target in a straight path, we have to assign two transfer tasks sequentially. Suppose one of the expected paths of this box is along the two arrows in the figure.
Then the first transfer task is to move the box to the left border, with intermediate target c_3. The immediate destination of the box is c_2, but because c_2 is occupied, a temporary transfer task is assigned to c_2's up neighbor to receive c_2's box. The immediate destination of this task is c_2's up neighbor, and the moving direction is up.

For presentation purposes, we consider the conveyor and the box as an entire "conveyor module," which is also an individual agent from the system modeling perspective (Gue et al., 2014; Uludağ, 2014; Seibold, 2015). Hence, every conveyor with a box takes on the attribute of the box: active conveyor module, temporary active conveyor module, and working conveyor module. An active conveyor or empty module are expressed as c_i^a or c_i^e respectively.

3.1.1 Conveyor states and representations of states

The state space of a conveyor module is defined by its box's category and movement confirmations (the term "movement confirmation" will be explained in Section 3.2.2). The names, transitions of these states, are displayed in Figure 32. Conveyor state representations are in Table 2.

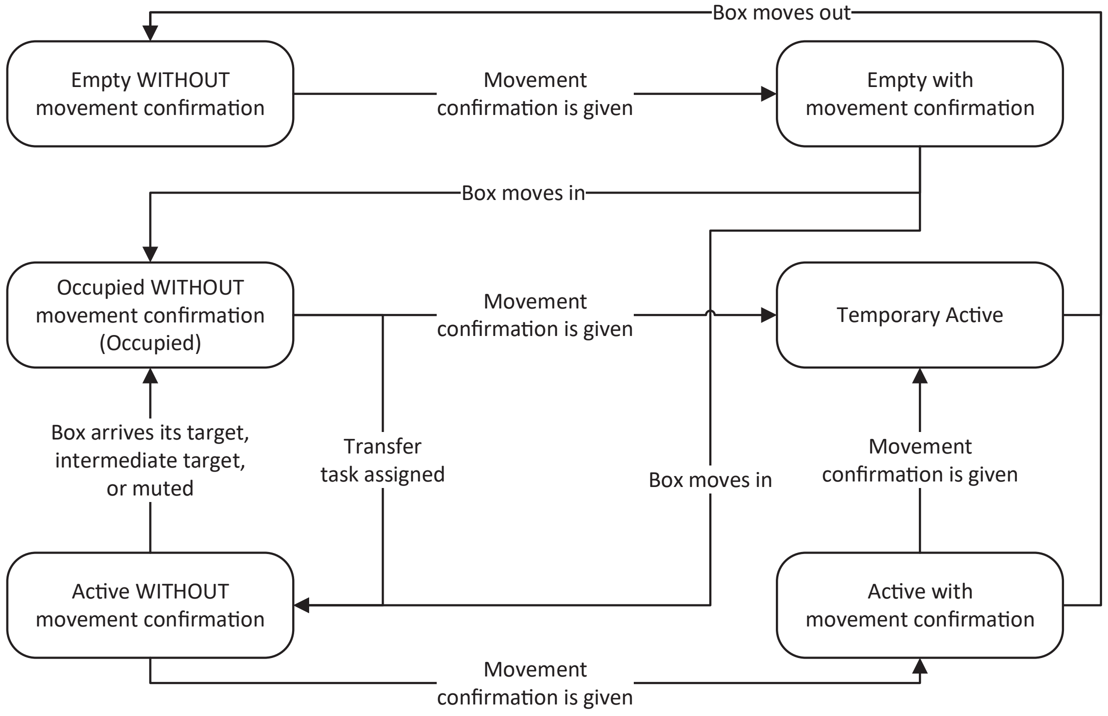
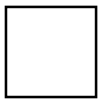
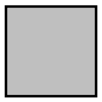
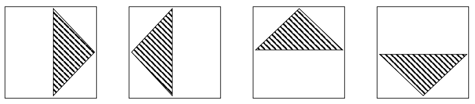
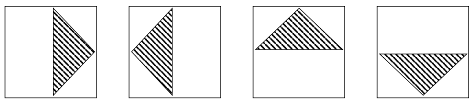
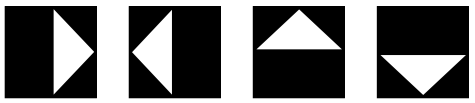
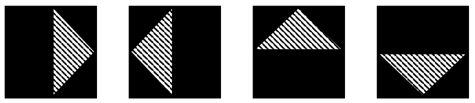

Box moves out
Empty WITHOUT movement confirmation
Movement confirmation is given
Empty with movement confirmation
Box moves in
Occupied WITHOUT movement confirmation (Occupied)
Movement confirmation is given
Temporary Active
Box arrives its target, intermediate target, or muted
Transfer task assigned
Box moves in
Movement confirmation is given
Active WITHOUT movement confirmation
Active with movement confirmation
Movement confirmation is given

Figure 32: States of a conveyor module in GridHub. The arrows show the events that trigger these transitions;

TABLE 2: Conveyor module and state representation (the pointing directions of the triangles indicate the active directions).

Empty	
Occupied	
Empty with movement confirmation	
Temporary active	
Active without movement confirmation	
Active with movement confirmation	

3.1.2 GridHub software architecture

We divide GridHub software into multiple layers, each performing certain functions or activities, and the information that is obtained externally is processed sequentially among these layers (Figure 33). When a conveyor module executes programs in the movement negotiation layer, it can only communicate with its physical neighbors. When a conveyor module runs programs in other layers, it can communicate with all other conveyor modules and with external entities. For example, when a conveyor module receives an external request, it can broadcast to all conveyors in the shared management layer. The methods of communication and sharing information among other conveyors is beyond the scope of this dissertation.

The structure in Figure 33 separates the methods of moving active boxes from the material handling task at hand (sort, sequence, retrieve, etc.), which allows developers to design new material handling methods without having to rethink "how conveyors get things done."

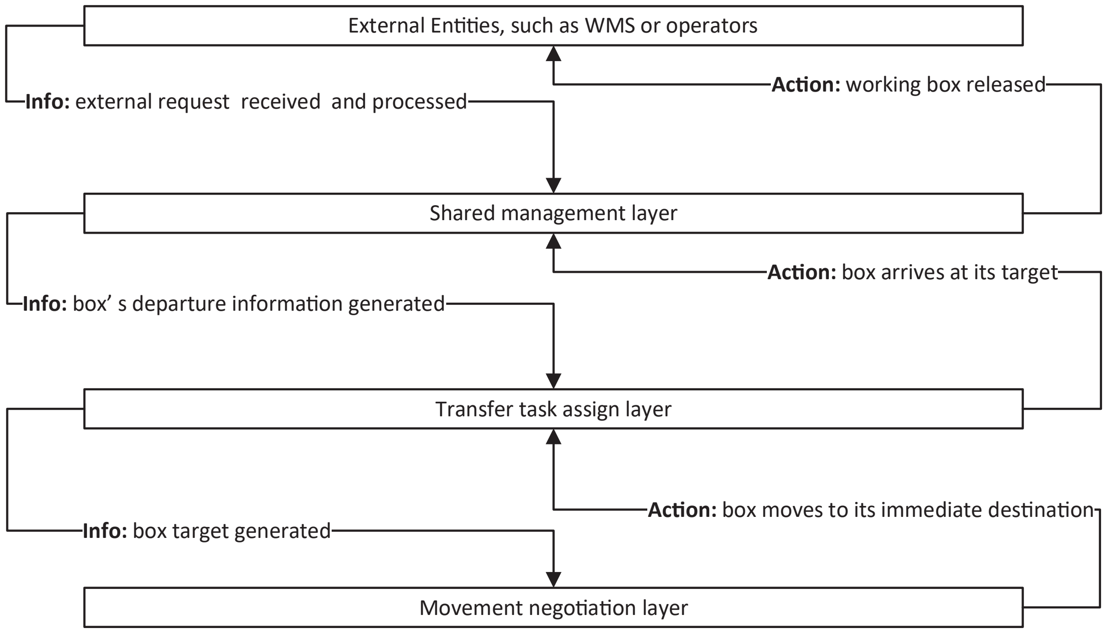

External Entities, such as WMS or operators
Info: external request received and processed
Action: working box released
Shared management layer
Info: box' s departure information generated
Action: box arrives at its target
Transfer task assign layer
Info: box target generated
Action: box moves to its immediate destination
Movement negotiation layer

Figure 33: GridHub system architecture. The arrows on the left indicate information flows; arrows on the right show actions performed as responses to the information. For example, the external requests are processed by the "shared management" layer and translated into transfer tasks. As responses to these external requests, some boxes are moved to their targets.

3.2 GridHub control algorithms

Like the existing grid-based system, such as GridStore (Gue et al., 2014) or GridPick (Uludağ, 2014), an *iteration* in GridHub is comprised of three phases: assess, negotiate, and convey. During each phase, every conveyor module executes several steps of the algorithms sequentially and synchronizes at each step; in other words, only after all modules complete their actions in one step, can they proceed to the next one all together.

3.2.1 Assess phase

In the assess phase, conveyor modules communicate with external entities and execute functions not related to moving boxes. Activities in this phase are completed by the shared management layer and the transfer task assignment layer (Figure 33).

3.2.1.1 Terminologies of assess phase

Before explaining the algorithm, we introduce two important concepts.

Higher priority directions GridHub's ability to move active boxes in all directions means active boxes might compete for space (Figure 34).

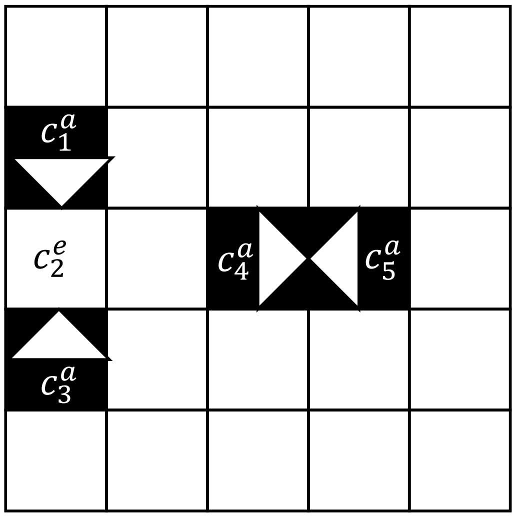


Figure 34: Example of conflicts. c_1^a and c_3^a conflict to move into c_2^e; c_4^a and c_5^a need one of the other modules to move their active box out of the way.

The solution is: all conveyor modules randomly choose one direction from a pair of opposite directions in every iteration, and every module has the same knowledge of the priority directions. Thus, in every iteration, all movements in the selected directions have higher priority. For example, if the higher priority directions are left and up in the case shown by Figure 34, c_3^a and c_5^a have priority. Since there are two pairs of opposite directions in GridHub, the higher priority directions include two directions, for instance, left and down. Thus, every active module has to choose a *matched higher priority direction* based on their own active directions.

For example, the matched higher priority direction for an active conveyor module with the left active direction is either right or left.

Limitation of task re-assignment Before an active box arrives at its target or intermediate target, the action of removing its transfer task is called *muting.* If its box is moved away after muting, with a temporary transfer task, we call the box *muted and moved.* Because departure information is not removed from muted boxes, the same or different transfer tasks can be re-assigned in a future iteration, which we call *re-activation.* When a box is muted or muted and moved, we avoid livelock by restricting the conveyor to re-assign the transfer task, but this can only occur if there are no *restrictions of task re-assignment* in this direction and the directions perpendicular to it (An example of task assignment restriction is in Figure 35).

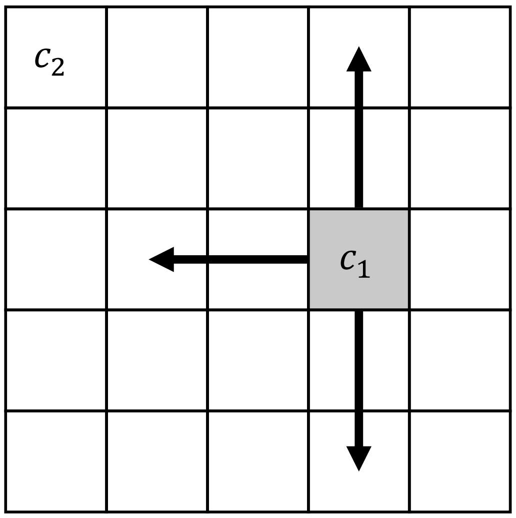


Figure 35: Example of task re-assignment (to left, up, and down) restriction. The box's target is c_2, so a transfer task toward left or up needs be to assigned in order to move the box into its target. If the box of c_1 has been moved from c_1's left neighbor in the previous iteration, before the limitation of re-assignment tasks has been removed, c_1 cannot re-assign a transfer task toward the left, up, or down.

3.2.1.2 Steps in the assess phase

Step 1: Evaluate box location First, every conveyor module receiving an active box from its neighbor compares the box's target or intermediate target to its own location. If they match, the module removes the transfer task from the box.

Second, all conveyor modules choose and update the higher priority directions in the current iteration simultaneously. There are several methods the conveyor modules use to communicate with each other and decide the higher priority directions. In this dissertation, we do not describe the details of these methods. We assume the conveyor modules can accomplish these methods. One method used during the simulation models is described in Chapter 5.

Step 2: Update information among all conveyor modules Conveyor modules share information with each other and communicate with external entities, such as the WMS.

Step 3: Assign transfer tasks For a working conveyor module located inside the grid, a transfer task (output task) is assigned based on one of its expected paths (see Figure 36); for a conveyor module holds newly arrived box, a transfer task (input task) is to move the box into the system (Figure 37). Both of the input and output tasks can be assigned when there is no restriction of task re-assignment in the required direction.

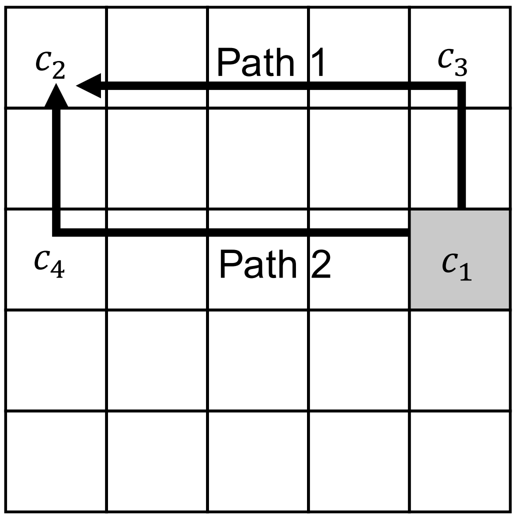


Figure 36: Expected paths of a working box from its original locations to its target.

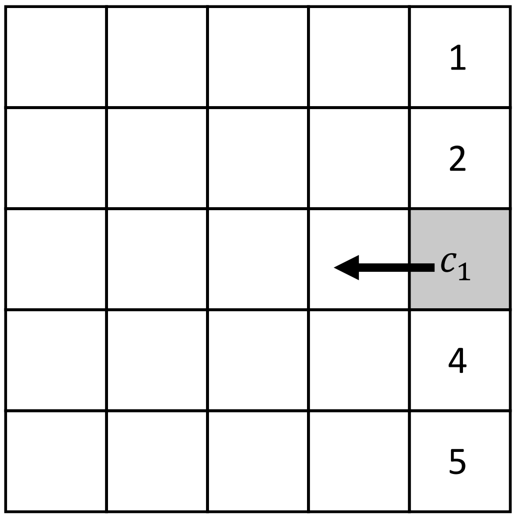


Figure 37: Transfer tasks to move a box into the system. Because c_1 is located on the right edge, it needs to assign a transfer task (input task) to its box, which is "to move its box to its left neighbor."

3.2.2 Negotiate phase

In this phase, the goal of the control algorithms is to move every active box to its immediate destination by arranging box movements via message passing. The conveyor modules' communication activities, which are related to arranging box movements, are *negotiation.* Negotiations are completed in the bottom layer, as shown in Figure 33.

3.2.2.1 Terminologies of negotiate phase

Box movements have two types, which are similar to box movements in GridStore (Gue et al., 2014) and GridPick (Uludağ, 2014). The first type is single movement of a box, which means a conveyor module moves its box to one of its neighbors (c_4^a in Figure 38). The second type of box movement is tandem or group movement, which means a group of consecutive conveyor modules moving their boxes in one direction (c_1^a and c_2^a in Figure 38). A group of movement is not necessarily formed by active modules only, for example, c_6 and c_7 in Figure 38).

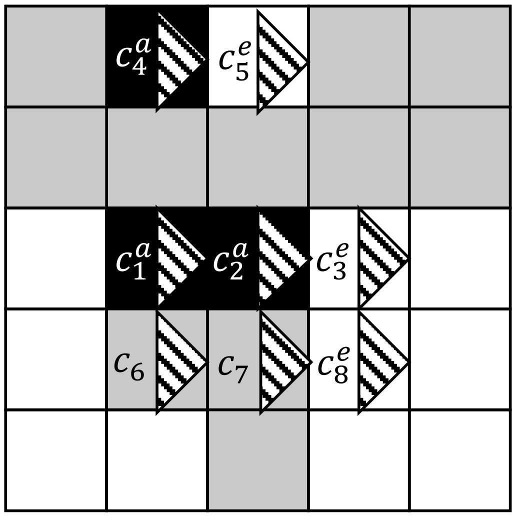


Figure 38: Examples of single and tandem box movements. c_4^a's box movement in Figure 38 is single movement. c_1^a moves its box to c_2^a, and c_2^a moves its box to c_3^e, so c_1^a and c_2^a form a tandem or group movement. c_6 and c_7 also form tandem movement.

Furthermore, every conveyor module is restricted to move its box to one of its neighbors in one iteration, which is similar to other grid-based systems. GridHub has a puzzle-like layout, so either single or tandem movements of boxes must occur in straight lines, in one iteration.

Negotiation messages are pieces of information generated and passed by conveyor modules. There are three types of messages: seek, confirm, and fail. Message content includes *passing directions* and the id of the conveyor module that initiated the message. In Table 3, different colored arrows are used to indicate the types of messages.

TABLE 3: Message operation symbols. The directions of the arrows are the message passing directions.

Type of message	Seek	Confirm	Fail
Symbol	←	←	←

When a conveyor module receives a message, it buffers the message first. The conveyor module *processes* every message in its buffer according to the sequence it receives them (earliest first). A conveyor module processes different messages following specific negotiation rules, which are described later in this section. The above process is similar to the “mailbox” concept in operating systems (Tanenbaum and Bos, 2015), and it is also similar to the process of dealing with received messages described by Sakao et al.. However, the communications of “ask states” are not done in the way described above, meaning every conveyor can obtain its neighbors’ states and related information instantly.

Actions are made by the conveyor module after processing every message, and these actions are performed before processing the next message. For example:

- After a conveyor module processes a seek message, it either (1) passes the message in the passing direction, (2) replies “confirm” to the sender, or (3) replies “fail” to the sender.
- After a conveyor module processes a confirm or fail message, it passes the message in the direction opposite the seek message’s passing direction.

If a seek message is confirmed, it is successful; otherwise, it fails.

3.2.2.2 The algorithm design ideas

If the immediate destination of an active module is available, the active module can move its box to the immediate destination directly. If the immediate destination of that module is unavailable, the active conveyor module attempts to find empty modules and to arrange one, or more box movements in order to make the immediate destination available in a future iterations. We divide the possible locations of empty conveyor modules into four categories based on the active module’s location (see Figure 39 and 40 for examples).

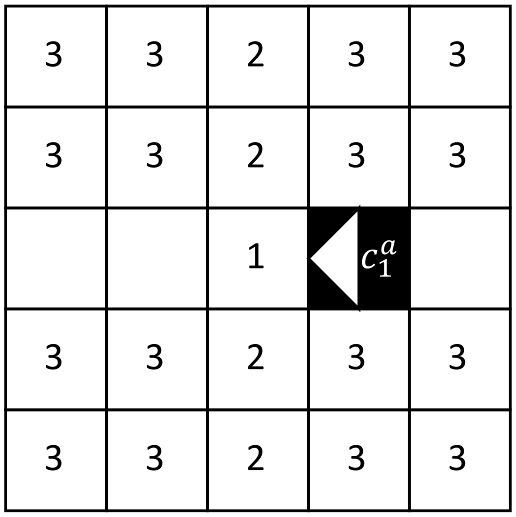


Figure 39: Empty module location categories 1 to 3. Category 1: the immediate destination of the active conveyor module. Category 2: the column or row the immediate destination is located. Category 3: the conveyor modules that are not located in the same column and row of this active module.

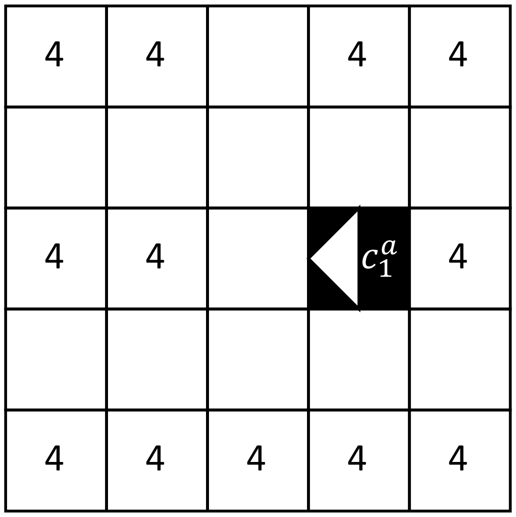


Figure 40: Empty module location categories 4.
Category 4: the conveyor modules that cannot be accessed by the paths designed for the first three categories in Figure 39.

If empty cell c_1^e's location is in regions 2 to 4, the possible paths of seek messages passed from c_1^a to c_1^e are displayed in Figure 41. When the active module can only find empty cells located from category 2 to 4, several iterations are needed to move an active box into its immediate destination. When an empty module's location belongs to category 1, its active box can be directly moved to its immediate destination (the least number of iterations). However, when an empty module's location belongs to category 4, the highest number of iterations is needed, because three extra iterations have to be completed to "switch" the location of an empty module from category 4 to category 1. In order to reduce the time required to move active boxes, we set every active module to attempt to move its box in the easiest way first, and perform the harder ones later only if the easier one is not successful. There are four attempts for every active module, and we name and put them into a set $\{N_1, N_2, N_3, N_4\}$. An arbitrary attempt can be written as N_i ($i \in \{1, 2, 3, 4\}$). The target of each attempt is to move one or a group of boxes to an empty module located in one category. So N_1 to N_4 are set to search for empty modules in categories 1 to 4 respectively. In these attempts, N_1 is expected to take the least number of iterations to move an active box to its immediate destination, and N_4 consumes the highest number of iterations.

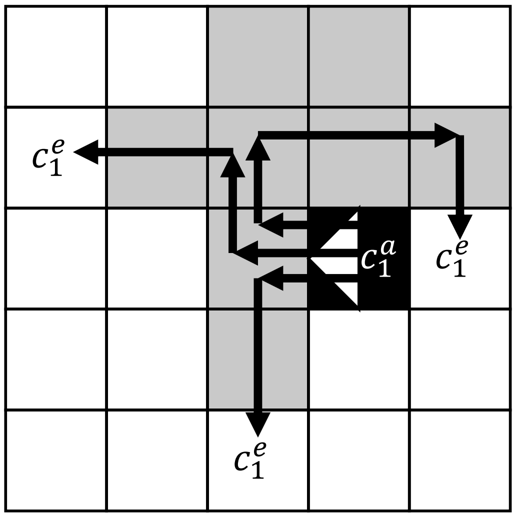


Figure 41: Possible seek paths from an active module to empty modules that belong to different categories.

When N_1 and N_4 act, they search empty modules and arrange box movements. These actions are similar, so they could be guided by similar algorithms. Therefore, we may not need to develop two stand alone algorithms for the actions in these attempts. To further explore this observation, we deconstruct N_2, N_3, and N_4 into components, finding that more actions can be guided by similar algorithms. Then we introduce the concept of a "nested attempt;" where each component of an attempt is called a *nested attempt*. The nest relationship is written as $N_i(N_j)$ $(i,j \in \{1,2,3,4\},\ j > i)$, which means that N_i is nested in N_j. If we describe one attempt nested in different attempts simultaneously, we use the notation $N_i([N_j, N_k])$ $(i,j,k \in \{1,2,3,4\},\ j > i, k > i, j \neq k)$. For example, $N_1([N_2, N_3])$ are used to describe a N_1 nested in a N_2 or a N_3.

The attempts after deconstruction are shown in Figure 42 and 45. All non-nested attempts are initiated by active conveyor modules, for example c_1^a in these figures. An N_1 does not have a nested attempt (Figure 42).

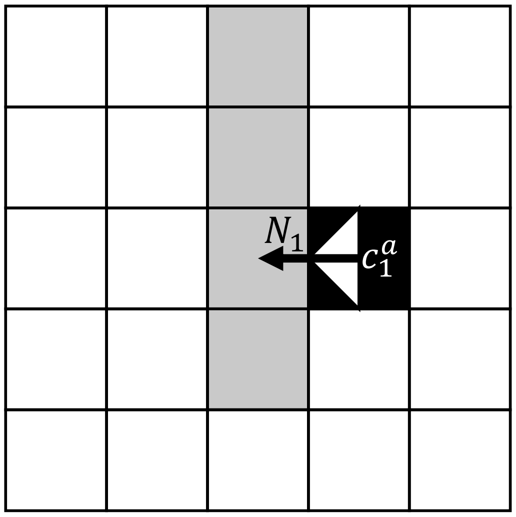


Figure 42: Seek message sending paths of N_1. c_1^a send seek message directly to its left neighbor.

An N_2 has a pair of nested $N_1(N_2)$ that have opposite seek message passing directions (Figure 43). One of these $N_1(N_2)$ is initiated first, and the other $N_1(N_2)$ is initiated after a fail message of the first $N_1(N_2)$ arrives at the original conveyor module. Furthermore, an active conveyor module initiates an attempt and randomly selects the sequences of message passing directions for all nested attempts, storing them in the seek message of the attempt. Other modules that initiate nested attempts obtain the message passing directions from the message, because the message's content is copied when passed.

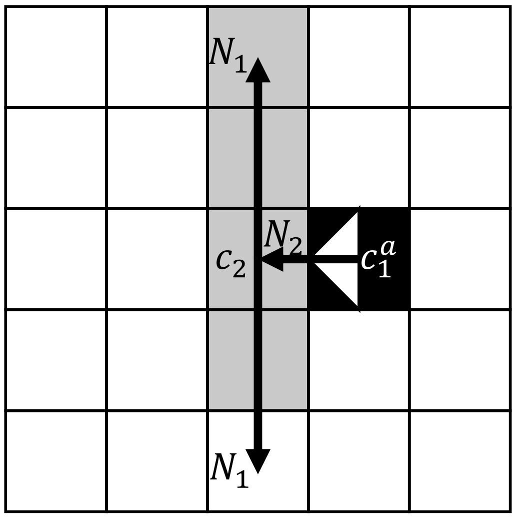


Figure 43: Seek messages sending paths of N_2 and $N_1(N_2)$. c_1^a selects the up direction as the first message passing direction that initiates $N_1(N_2)$. Then, c_2 initiates the first $N_1(N_2)$ to the up direction. If $N_1(N_2)$ fails, c_2 initiates another $N_1(N_2)$ to the down direction.

An N_3 has several nested $N_2(N_3)$ (Figure 44). Some of the $N_2(N_3)$ pass their seek messages in one direction first. If conveyor modules reply with fail messages to these seek messages, the other set of $N_2(N_3)$ are initiated by the same conveyor module (for example c_2 in Figure 44) in the opposite direction. Each of these $N_2(N_3)$ also starts a pair of $N_1(N_3)$. When some conveyor modules reply with fail messages to each pair of these $N_1(N_3)$, the conveyor module that initiates them to pass a seek message of $N_2(N_3)$ to the next conveyor module to initiate another pair of $N_1(N_3)$.

A N_4 has two nested $N_3(N_4)$ (Figure 45). One $N_3(N_4)$ proceeds in one direction first. If this $N_3(N_4)$ fails, the other $N_3(N_4)$ is initiated by the same conveyor module in the opposite direction. The message passing pattern of the $N_3(N_4)$ is the same as N_3's.

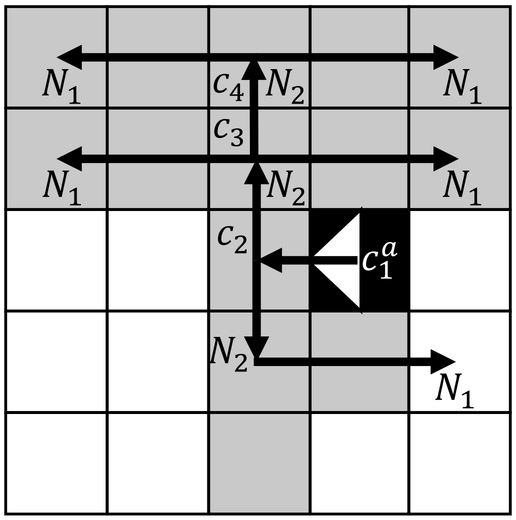


Figure 44: Seek message sending paths of N_3, $N_1(N_3)$, and $N_2(N_3)$. c_1^a randomly selects left as the message passing direction for the first $N_1(N_3)$. Then c_3 or c_4 initiates $N_1(N_3)$ to the left direction, first. If the pair of $N_1(N_3)$ initiated by c_3 fail, then c_3 passes a seek message of $N_2(N_3)$ to its up neighbor to initiate another pair of $N_1(N_3)$.

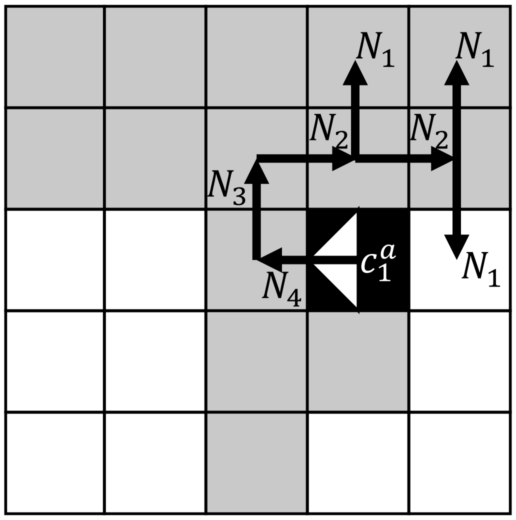


Figure 45: Seek messages sending paths of N_4, $N_1(N_4)$, $N_2(N_4)$ and $N_3(N_4)$.

From the Figure 42 to 45, we find that all box movements are arranged by N_1 or $N_1([N_2, N_3, N_4])$. Additionally, we discover: the directions of the movements arranged by N_1 or $N_1(N_3)$ are in or opposite to the active direction of c_1^a; the directions of the movements arranged by $N_1(N_2)$ and $N_1(N_4)$, which are also initiated by c_1^a, are perpendicular to the active direction of c_1^a. If active conveyors that have perpendicular active directions perform their attempts together, a conflict can result (see Figure 46). To solve this problem, we set the active conveyor modules with left and right active directions initiated every N_i prior to the active conveyor modules that have up and down active directions. In other words, in every step, only one type of attempt can be initiated by active conveyor modules that have opposite active directions, and only the movements that have opposite moving directions attempt to be arranged simultaneously. When a conveyor module is asked to move its box to a pair of opposite directions simultaneously, we apply the higher priority directions rule to solve this conflict. In summary, there are *8* steps for locating empty conveyor modules in the negotiate phase. They proceed in the sequence displayed by Table 4.

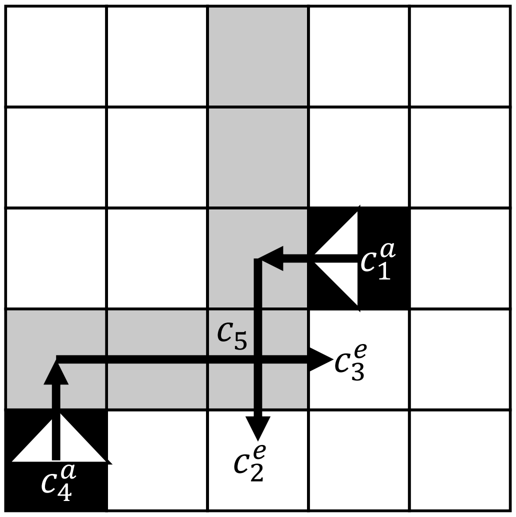
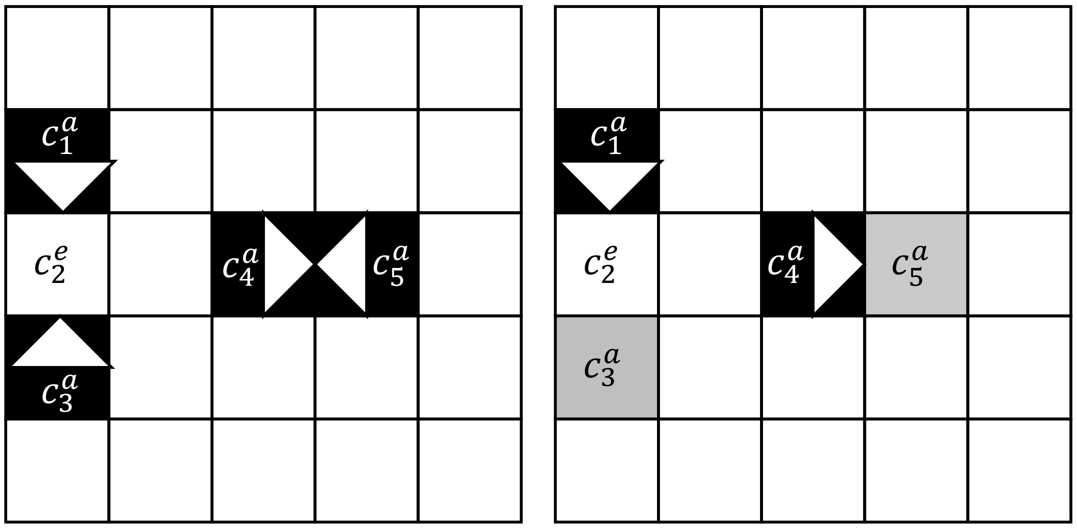


Figure 46: Example of perpendicular movements confusion. Suppose c_1^a and c_2^a initiate N_2 together. When the both N_1(N_2) find empty modules (c_2^e and c_3^e respectively), the conveyor module (c_5) that passes both seek messages must resolve its moving direction.

3.2.2.3 Steps of the negotiate phase

In step 0 of the negotiate phase, every empty conveyor module that has conflicts with other modules sends a signal to mute its neighbor on the matched higher priority direction of the current iteration. Consequently, the active module which active direction has lower priority is muted (Figure 47). For every active conveyor module that has conflicts with other modules, if its active direction is the same as the matched higher priority direction, it mutes its neighbor that is on its active direction. When an active conveyor is muted in this step, a restriction of task re-assignment on its active direction will be added in the convey phase.



(a) Conflicts (b) After conflicts are soloved.

Figure 47: Conflicts and after solving conflict. c_1^a and c_3^a conflict to move into c_2^e; c_4^a and c_5^a need one of the other modules to move their active box out of the way. Let the higher priority direction are down and right. c_3^a and c_5^a are muted, because their active directions do not have higher priority.

The remaining algorithms of the negotiate phase are executed in the sequence defined in Table 4. The message passing patterns for every attempt and nested attempt are illustrated in Figure 42 and 45. When an active module that has initiated an attempt receives a fail or confirm message it has completed one step of the control algorithms.

TABLE 4: Sequence of initiating attempts.

Step Number	Attempt	Active direction of the conveyor module that initiates the attempt
1	N_1	left and right
2	N_1	up and down
3	N_2	left and right
4	N_2	up and down
5	N_3	left and right
6	N_3	up and down
7	N_4	left and right
8	N_4	up and down

3.2.2.4 Glossary of negotiation rules

The major content of these algorithms in the negotiate phase is summarized in the negotiation rules. In the remaining parts of this dissertation, the rules are named in the format of $at(at).stage.number$. The "stage" describes messages' process stage, which includes the message types: g (initialization of messages); s (processing seek messages); c (processing confirm messages); f (processing fail messages). The numbers are used to distinguish the rules for the same attempt. For example, $N_1.s.01$ is the first (01) rule of attempt N_1 when processing seek messages (s). Some of the negotiation rules may be applied to different attempts. We will write the attempts together, such as, $[N_2, N_3].s.01$ for both of the N_2 and N_3 attempts applied to the same rule.

We describe the negotiation rules according to attempts and stages, and we use a flowchart to describe most of the negotiation rules. When a conveyor module starts processing a message, it begins checking its states and additional information at one of these nodes (left side of a flow chart). Then, it follows the arrows and checks the message content, which is shown at the top of the chart. A decision or process result is in one of the rectangular nodes on the right hand side.

TABLE 5: Notations for additional conditions of a conveyor module in flowcharts.

Notation	Explanation
BO(+), BO(-)	The conveyor module is at (+), or is not at (-) the system border
EA(+), EA(-)	The module's active direction (if applicable) has(+), or not have (-) higher priority
N1C(+), N1C(+)	The module's movement confirmation is (+), or is not (-) made by N_1
WNR(+), WNR(-)	The module is (+), or is not (-) waiting for response of a nested attempt
ANC(+), ANC(-)	All of the nested attempts initiated by this module are (+) completed, or are not (-) completed
EC(+), EC(-)	The module's movement confirmation's direction has (+), or does not have (-) higher priority
OC(+), OC(-)	The module is (-), or is not (-) the module initiates the attempt

TABLE 6: Actions or decisions of a module for processing a message.

Types of messages	Actions or decisions
Seek	0: do nothing 1: pass the message 2: reply a confirm message 3: reply a fail message 4: initiate nested attempts
Confirm	0: do nothing 1: pass the message 2: mark confirmations 3: change the modules states to temporary active (for N_1 and $miN_1([N_2, N_3, N_4])$ only) 4: pass the confirm message of the nested attempt
Fail	0: do nothing 1: pass the message 4: restart the same attempt in the opposite direction

3.2.2.5 Rules of initiating attempts

To initiate N_1 to N_4 at every active conveyor module, the active module has to meet one condition: there are no active confirmations from the previous attempts. For example, an active module cannot initiate N_3 if it has confirmations of N_1 or N_2.

$[N_2, N_3, N_4].g.01$: The rules to initiate N_2 to N_4 are listed as follows.

1. A conveyor module in the "active without movement confirmation" state can initiate N_2 to N_4, if it has no other confirmations from all previous attempts (Figure 48).
2. A conveyor module in the "empty with movement confirmation" state can initiate N_2 to N_4, if it meets the following conditions: It has confirmations of N_1 and it is not the target or intermediate target of the box moving to it. And,

except the confirmation of N_1, it has no other confirmations from all of the previous attempts (Figure 48).

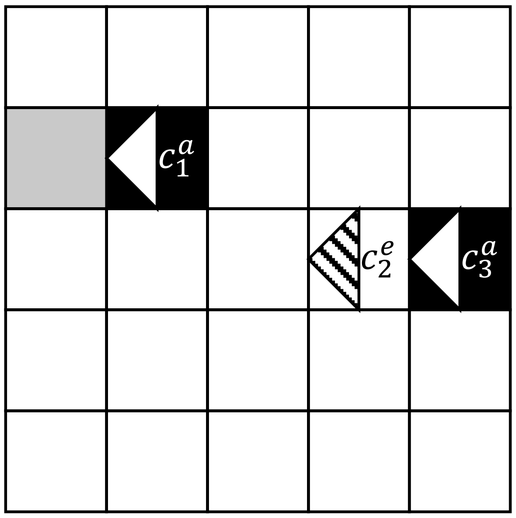
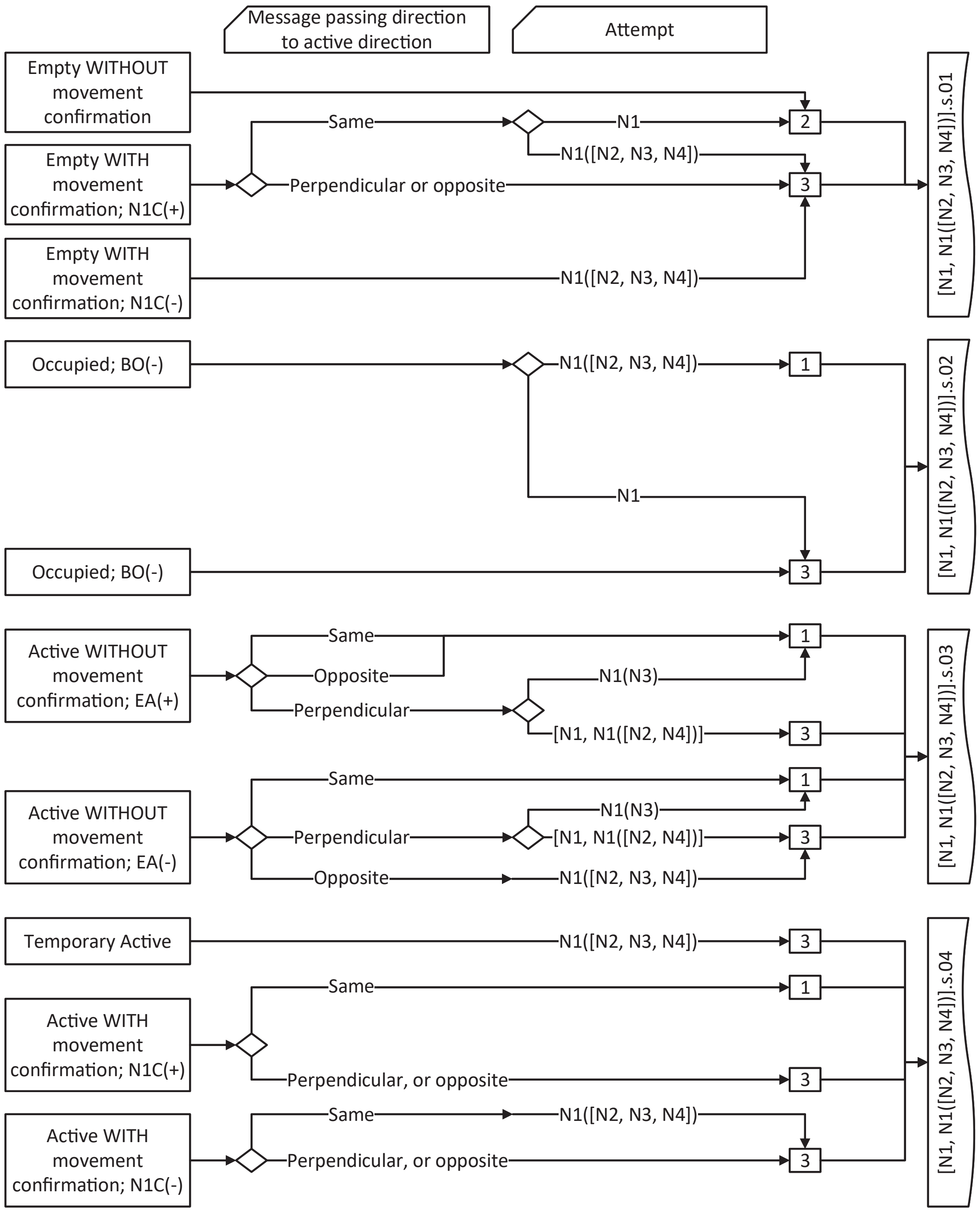


Figure 48: Example of $[N_2, N_3, N_4].g.01$. Since there is no other confirmations on c_1^a, then it can initiate future attempts, but c_3^a cannot initiate further attempts because it has already confirmed the box's left movement. Suppose c_3^a's active direction is left, and c_2^e is not the target or intermediate target of c_3^a's box. When c_2^e has no other confirmations except the confirmation of N_1 to move c_3^a's box, then c_2^e can initiate N_2 to N_4.

From step 3 to step 8, an empty conveyor module with movement confirmation can also initiate N_2 to N_4, when it meets the conditions in rule $[N_2, N_3, N_4].g.01$. The benefit of empty conveyor modules initiating attempts is that they form non-breaking "visual aisles" for active boxes. This idea was used in the GridStore, GridPick, and GridSequence systems. In GridHub, we still implement this idea, and name it *forward attempts*. Furthermore, the forward attempts are optional in the control algorithms, so they can be enabled or disabled.

Every conveyor module that initiates N_2 to N_4 is considered an active conveyor module. The active direction of this kind of empty module is the movement direction it confirms, and its immediate destination is its neighbor, which is in the active direction. For example, c_1^a and c_2 in Figure 48 are both considered active conveyor modules. Their active directions are both left, and their immediate destinations are their left neighbors respectively.

3.2.2.6 Rules to process seek message of N_1 and $N_1([N_2, N_3, N_4])$



Message passing direction to active direction
Attempt
Empty WITHOUT movement confirmation
Empty WITH movement confirmation; N1C(+)
Empty WITH movement confirmation; N1C(-)
Same
Perpendicular or opposite
N1
N1([N2, N3, N4])
Occupied; BO(-)
Occupied; BO(-)
Active WITHOUT movement confirmation; EA(+)
Opposite
Perpendicular
N1(N3)
[N1, N1([N2, N4])]
Active WITHOUT movement confirmation; EA(-)
Temporary Active
Active WITH movement confirmation; N1C(+)
Perpendicular, or opposite
Active WITH movement confirmation; N1C(-)
1
2
3
[N1, N1([N2, N3, N4])].s.01
[N1, N1([N2, N3, N4])].s.02
[N1, N1([N2, N3, N4])].s.03
[N1, N1([N2, N3, N4])].s.04

Figure 49: Rules of processing N_1 and $N_1([N_2, N_3, N_4])$'s seek messages.

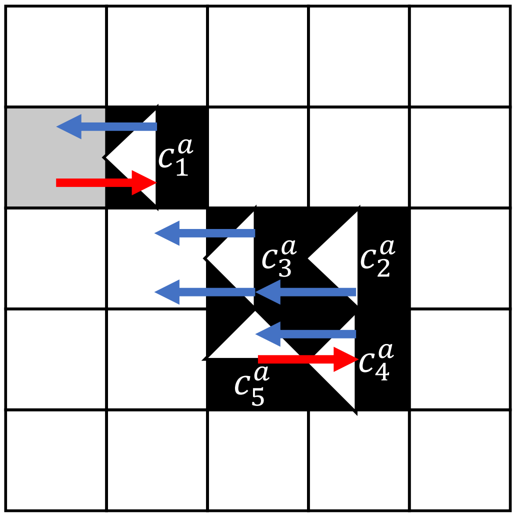


Figure 50: Examples of rules for processing N_1 and $N_1([N_2, N_3, N_4])$'s seek messages (part 1). The seek messages initiated by c_1^a and c_4^a fail according to $[N_1, N_1([N_2, N_3, N_4])].s.02$ and $[N_1, N_1([N_2, N_3, N_4])].s.03$ respectively. The seek messages from c_2^a and c_3^a can be passed due to $[N_1, N_1([N_2, N_3, N_4])].s.03$.

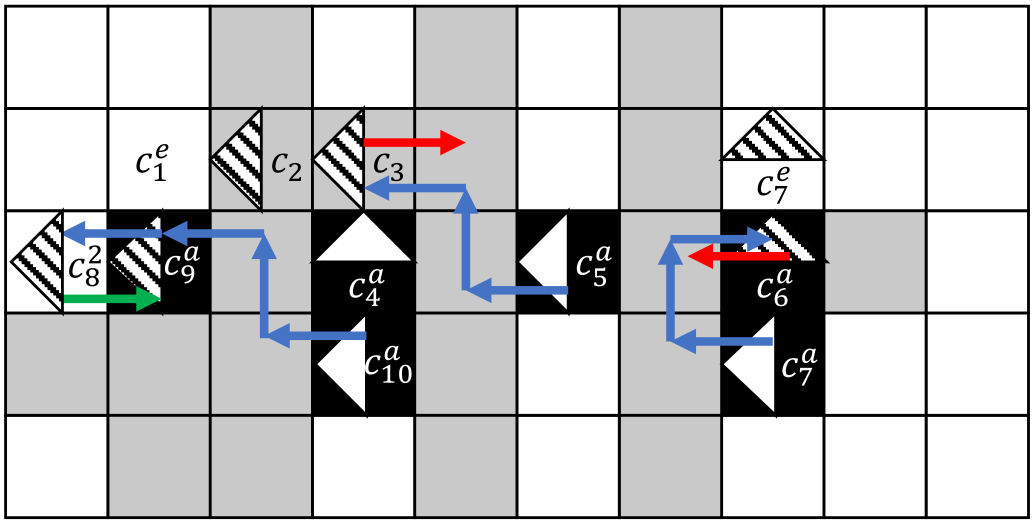


Figure 51: Examples of rules for processing N_1 and $N_1([N_2, N_3, N_4])$'s seek messages (part 2). c_8^a replies with a confirm message according to $[N_1, N_1([N_2, N_3, N_4])].s.01$. Both c_3 and c_6^a cannot pass seek messages ($[N_1, N_1([N_2, N_3, N_4])].s.04$).

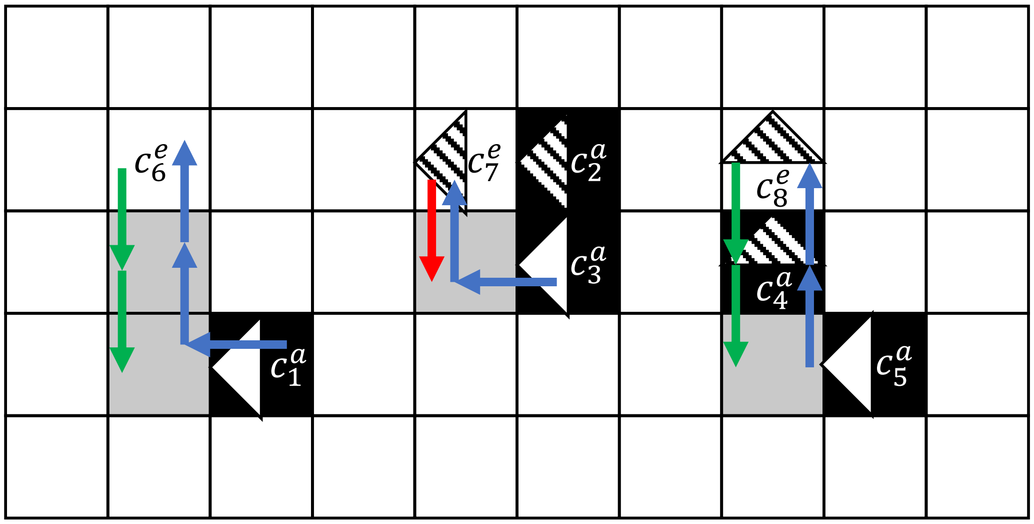


Figure 52: Examples of rules for processing N_1 and $N_1([N_2, N_3, N_4])$'s seek messages (part 3). c_1^a triggers its left neighbor to initiate $N_1(N_2)$. The two conveyor modules in "occupied without movement confirmation" states pass the seek message according to rule $[N_1, N_1([N_2, N_3, N_4])].s.02$. c_6^e replies to the seek message with a confirm message according to rule $[N_1, N_1([N_2, N_3, N_4])].s.01$. Because c_7^e is in the "empty with movement confirmation" state, it replies fail to the seek message initiated by c_3^a ($[N_1, N_1([N_2, N_3, N_4])].s.01$). c_5^a initiates the seek message that is passed by c_4^a. c_8^e responds with a confirm message. The passing is according to $[N_1, N_1([N_2, N_3, N_4])].s.04$, and the replying is according to $[N_1, N_1([N_2, N_3, N_4])].s.01$. The reason is that the existing confirmation is marked by messages belongs to N_1.

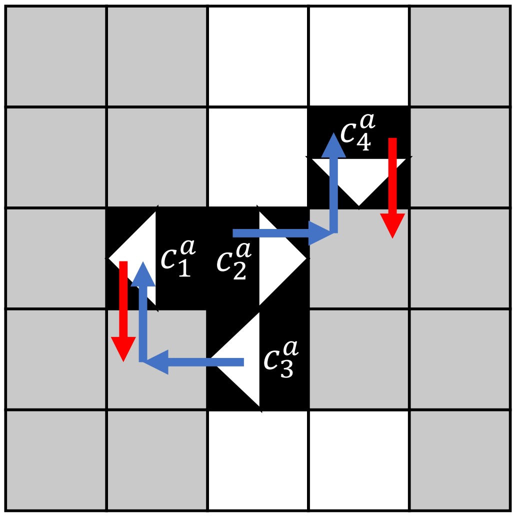


Figure 53: Examples of rules for processing N_1 and $N_1([N_2, N_3, N_4])$'s seek messages (part 4). c_3^a initiates a seek message with perpendicular direction to c_1^a's active direction. c_1^a replies with a fail message according to $[N_1, N_1([N_2, N_3, N_4])].s.03$. c_4^a replies with a fail message to the seek message initiated by c_2^a, when the matched higher priority direction is "down" ($[N_1, N_1([N_2, N_3, N_4])].s.03$).

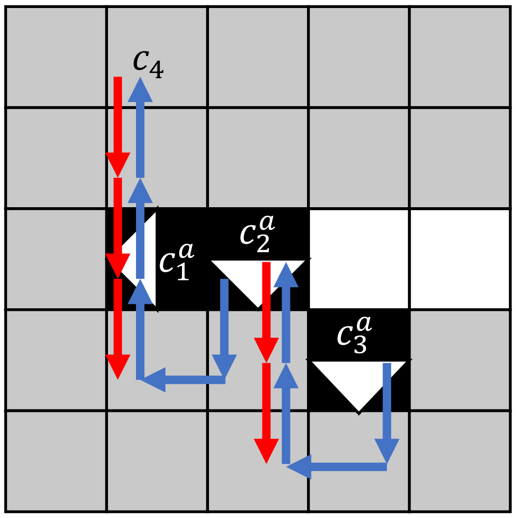
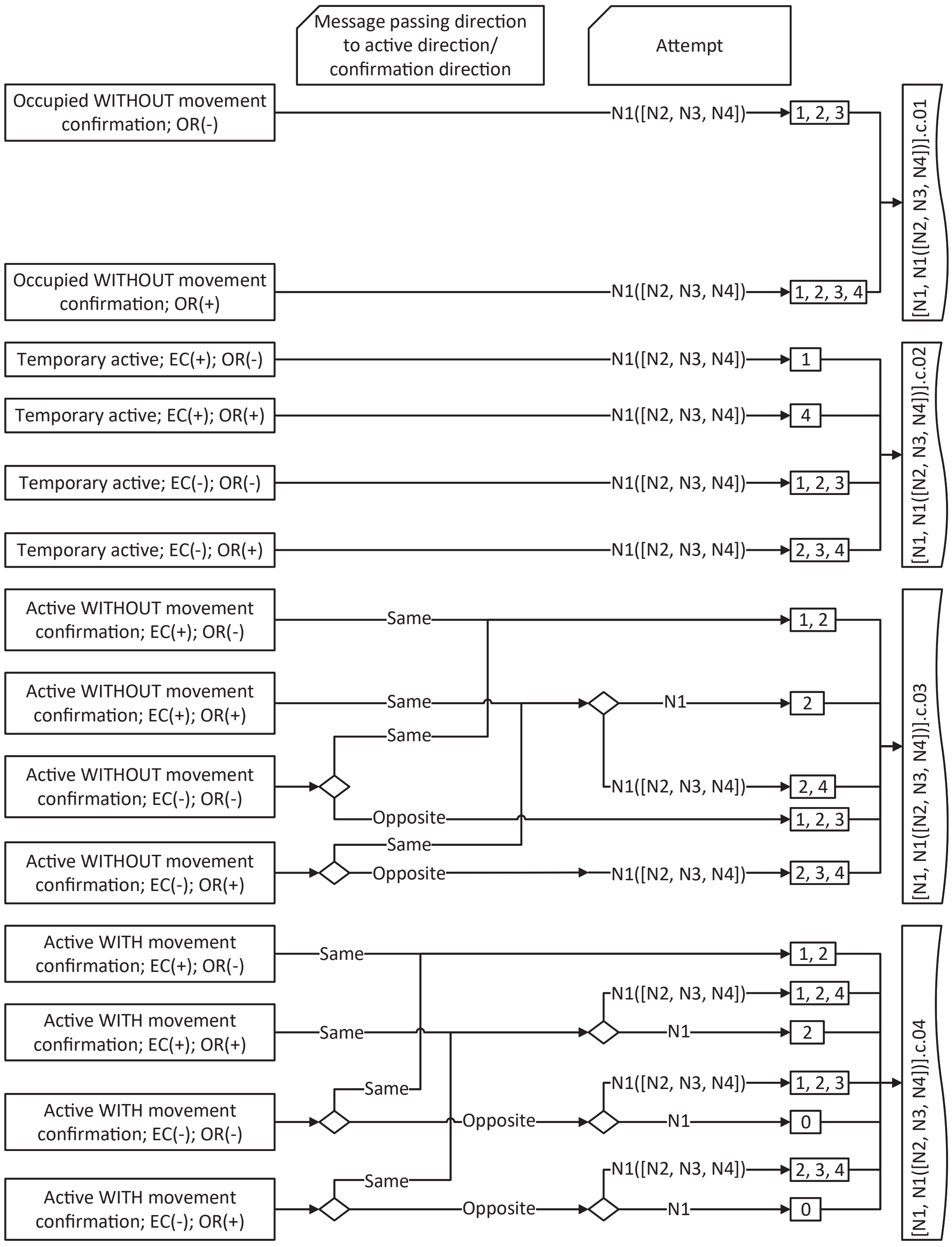


Figure 54: Examples of rules for processing N_1 and $N_1([N_2, N_3, N_4])$'s seek messages (part 4). c_1^a passes c_2^a's seek message ($[N_1, N_1([N_2, N_3, N_4])].s.03$). c_4 responds with a fail message to the seek message because c_4 is located at the system border ($[N_1, N_1([N_2, N_3, N_4])].s.02$). c_2^a replies with a fail message to c_3^a's seek message, when the current matched higher priority direction is "down" ($[N_1, N_1([N_2, N_3, N_4])].s.03$.

3.2.2.7 Rules to process confirm message of N_1 and $N_1([N_2, N_3, N_4])$



Message passing direction to active direction/ confirmation direction
Attempt
Occupied WITHOUT movement confirmation; OR(-)
Occupied WITHOUT movement confirmation; OR(+)
Temporary active; EC(+); OR(-)
Temporary active; EC(+); OR(+)
Temporary active; EC(-); OR(-)
Temporary active; EC(-); OR(+)
Active WITHOUT movement confirmation; EC(+); OR(-)
Active WITHOUT movement confirmation; EC(+); OR(+)
Active WITHOUT movement confirmation; EC(-); OR(-)
Active WITHOUT movement confirmation; EC(-); OR(+)
Active WITH movement confirmation; EC(+); OR(-)
Active WITH movement confirmation; EC(+); OR(+)
Active WITH movement confirmation; EC(-); OR(-)
Active WITH movement confirmation; EC(-); OR(+)
Same
Opposite
N1
N1([N2, N3, N4])
1, 2, 3
1, 2, 3, 4
1
4
2, 3, 4
1, 2
2
2, 4
1, 2, 4
0
[N1, N1([N2, N3, N4])].c.01
[N1, N1([N2, N3, N4])].c.02
[N1, N1([N2, N3, N4])].c.03
[N1, N1([N2, N3, N4])].c.04

Figure 55: Rules of processing N_1 and $N_1([N_2, N_3, N_4])$'s confirm messages.

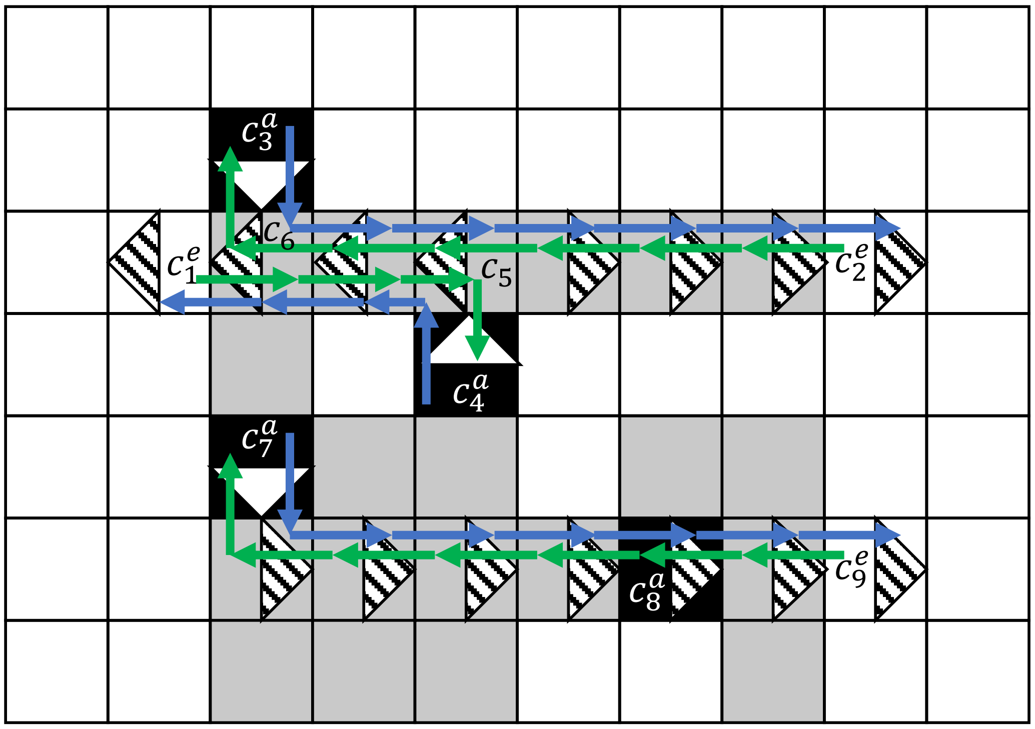


Figure 56: The examples of rules of processing N_1 and $N_1([N_2, N_3, N_4])$'s confirm messages. Let the matched higher priority direction be left. According to $[N_1, N_1([N_2, N_3, N_4])].c.02$, when modules pass a confirm message to c_3^a, conveyor modules located between c_5 and c_6 cannot mark movement confirmations. When c_8^a processes c_7^a's confirm message, it marks movement confirmation ($[N_1, N_1([N_2, N_3, N_4])].c.03$).

3.2.2.8 Rules to process fail message of N_1 and $N_1([N_2, N_3, N_4])$

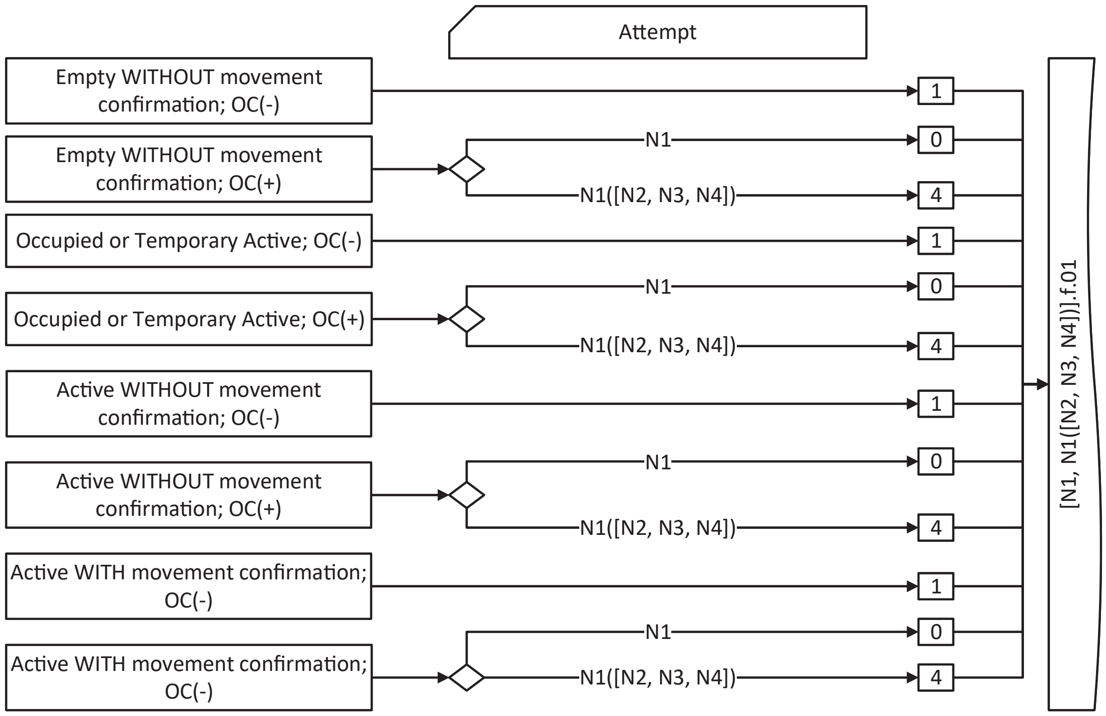

Attempt
Empty WITHOUT movement confirmation; OC(-)
Empty WITHOUT movement confirmation; OC(+)
Occupied or Temporary Active; OC(-)
Occupied or Temporary Active; OC(+)
Active WITHOUT movement confirmation; OC(-)
Active WITHOUT movement confirmation; OC(+)
Active WITH movement confirmation; OC(-)
Active WITH movement confirmation; OC(-)
N1
N1([N2, N3, N4])
1
0
4
[N1, N1([N2, N3, N4])].f.01

Figure 57: Rules of processing N_1 and $N_1([N_2, N_3, N_4])$'s fail messages.

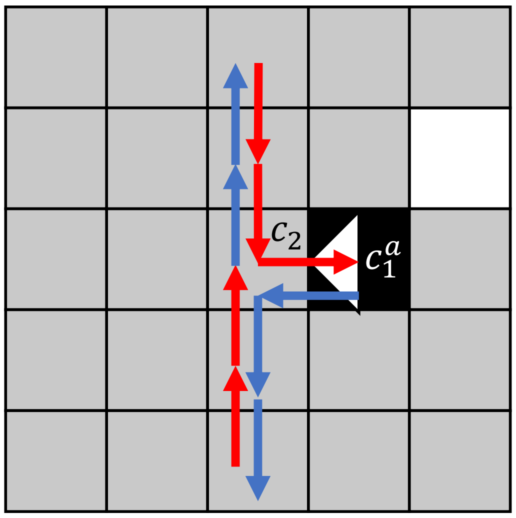


Figure 58: The examples of rules of processing N_1 and $N_1([N_2, N_3, N_4])$'s fail messages. A $N_1(N_2)$'s fail message is passed to the original conveyor module (c_2) which passes the N_2's fail message to the active module c_1^a.

3.2.2.9 Rules to process seek message of N_2 and $N_2([N_3, N_4])$

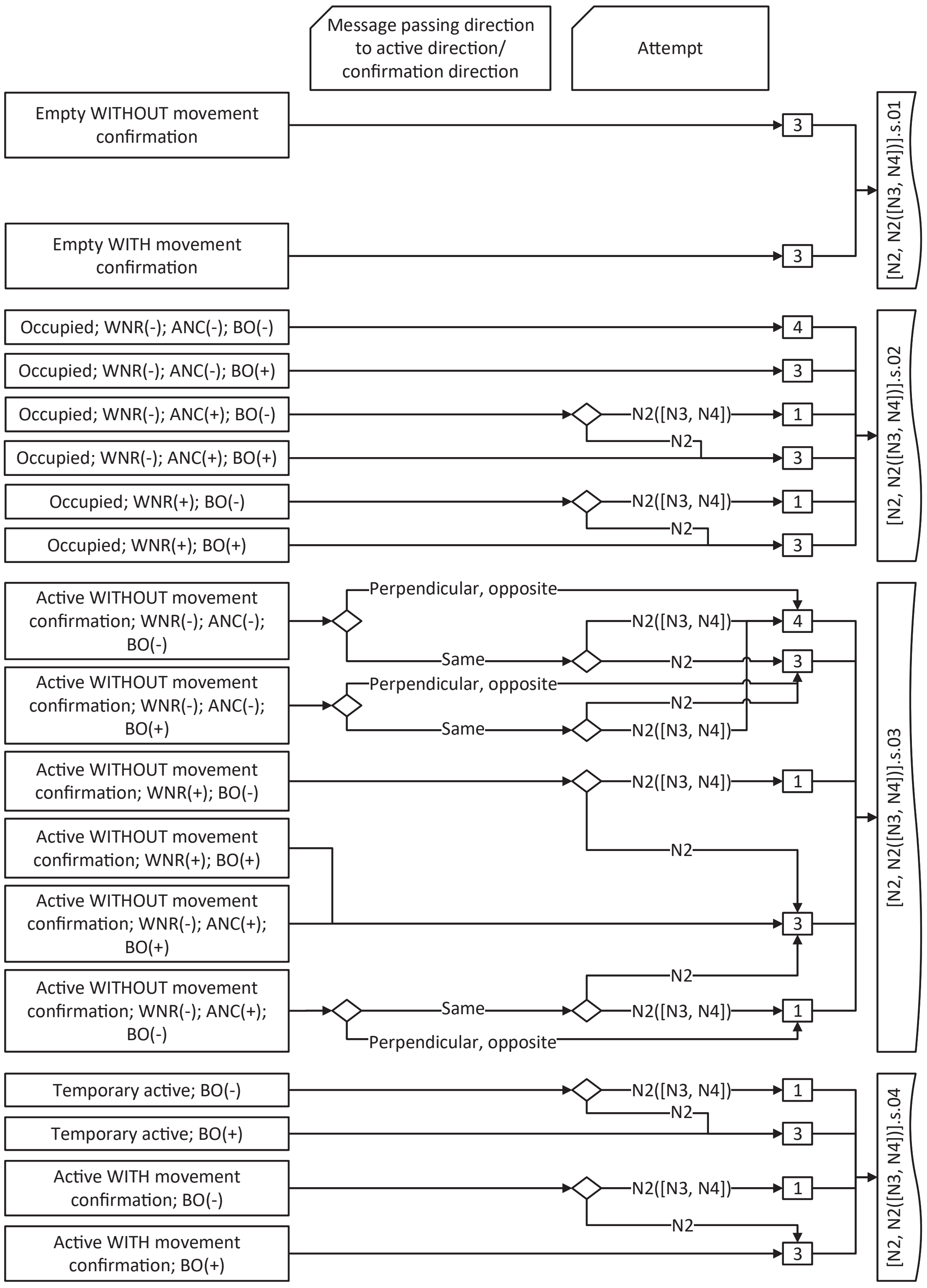

Message passing direction to active direction/ confirmation direction
Attempt
Empty WITHOUT movement confirmation
Empty WITH movement confirmation
[N2, N2([N3, N4])].s.01
Occupied; WNR(-); ANC(-); BO(-)
Occupied; WNR(-); ANC(-); BO(+)
Occupied; WNR(-); ANC(+); BO(-)
Occupied; WNR(-); ANC(+); BO(+)
Occupied; WNR(+); BO(-)
Occupied; WNR(+); BO(+)
N2([N3, N4])
N2
[N2, N2([N3, N4])].s.02
Active WITHOUT movement confirmation; WNR(-); ANC(-); BO(-)
Active WITHOUT movement confirmation; WNR(-); ANC(-); BO(+)
Active WITHOUT movement confirmation; WNR(+); BO(-)
Active WITHOUT movement confirmation; WNR(+); BO(+)
Active WITHOUT movement confirmation; WNR(-); ANC(+); BO(+)
Active WITHOUT movement confirmation; WNR(-); ANC(+); BO(-)
Perpendicular, opposite
Same
[N2, N2([N3, N4])].s.03
Temporary active; BO(-)
Temporary active; BO(+)
Active WITH movement confirmation; BO(-)
Active WITH movement confirmation; BO(+)
[N2, N2([N3, N4])].s.04
1
3
4

Figure 59: Rules of processing N_2 and $N_2([N_3, N_4])$'s seek messages.

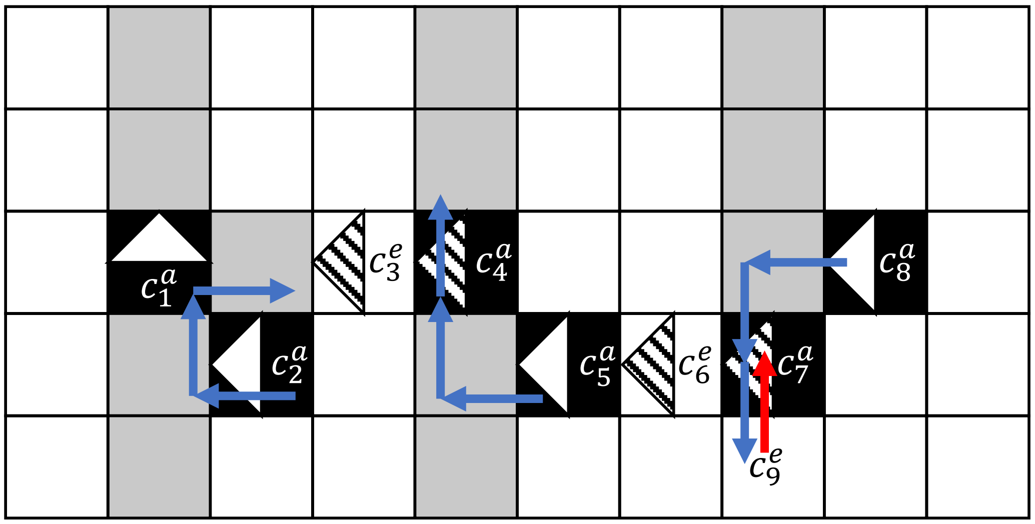


Figure 60: Examples of rules of processing N_2 and $N_2([N_3, N_4])$'s seek messages (part 1). After processing the seek message from c_2^a, c_1^a initiates $N_1(N_3)$ because the message belongs to $N_2(N_3)$, but not N_2 ($[N_2, N_2([N_3, N_4])].s.03$). Because c_4^a is in the "active with movement confirmation" state, it passes $N_2(N_3)$'s seek message ($[N_2, N_2([N_3, N_4])].s.04$). The same actions are repeated by c_7^a, but c_9^e has to reply with a fail message because it is in the "empty with out movement confirmation" state. Suppose c_{10} does not succeed in one set of $N2(N_3)$. Since it is at the system border, it replies with a fail message to $N2(N_3)$'s seek message, whose passing direction is up ($[N_2, N_2([N_3, N_4])].s.02$).

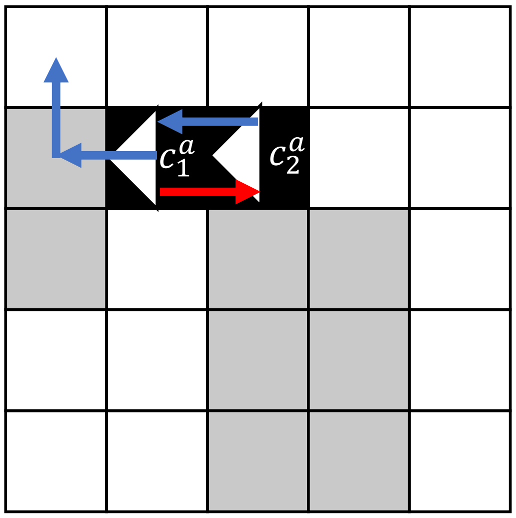


Figure 61: Examples of rules of processing N_2 and $N_2([N_3, N_4])$'s seek messages (part 2). c_1^a initiates N_2's seek message and passes this message to its left neighbor. That module starts $N_1(N_2)$ based on $[N_2, N_2([N_3, N_4])].s.02$. c_1^a fails N_2's seek message because its active direction is the same as the message's passing direction ($[N_2, N_2([N_3, N_4])].s.03$).

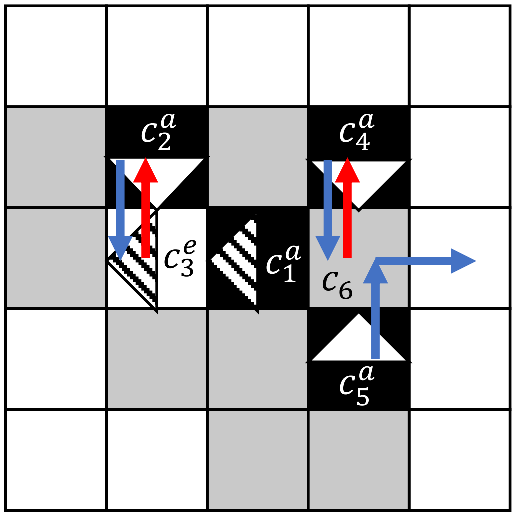


Figure 62: Examples of rules of processing N_2 and $N_2([N_3, N_4])$'s seek messages (part 3). c_3^e replies with a fail message to the seek message from c_2^a because it gives confirmation to receive c_1^a's box in a previous step ($[N_2, N_2([N_3, N_4])].s.02$). Both c_4^a and c_5^a initiate N_2's seek messages. Suppose c_6 receives and processes the message from c_5^a first. When c_6 processes the message from c_4^a, it replies with a fail message because it is waiting for the response of a $N_1(N_2)$ message ($[N_2, N_2([N_3, N_4])].s.02$).

3.2.2.10 Rules to process confirm message of N_2 and $N_2([N_3, N_4])$

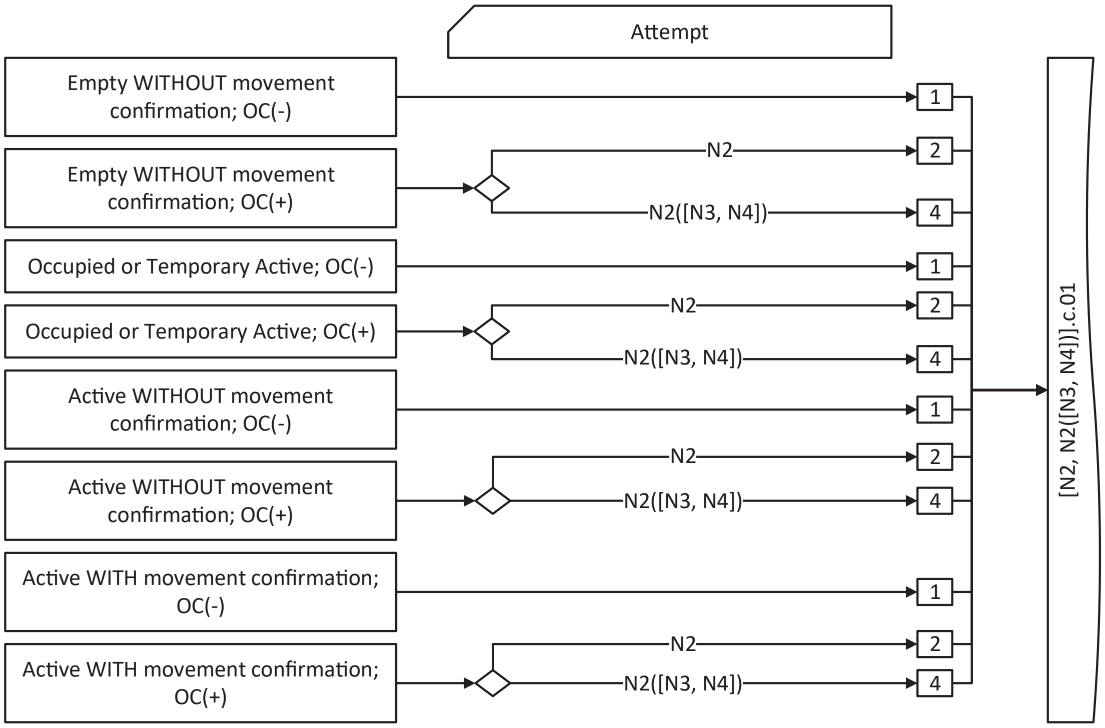

Attempt
Empty WITHOUT movement confirmation; OC(-)
Empty WITHOUT movement confirmation; OC(+)
Occupied or Temporary Active; OC(-)
Occupied or Temporary Active; OC(+)
Active WITHOUT movement confirmation; OC(-)
Active WITHOUT movement confirmation; OC(+)
Active WITH movement confirmation; OC(-)
Active WITH movement confirmation; OC(+)
N2
N2([N3, N4])
1
2
4
[N2, N2([N3, N4])].c.01

Figure 63: Rules of processing N_2 and $N_2([N_3, N_4])$'s confirm messages.

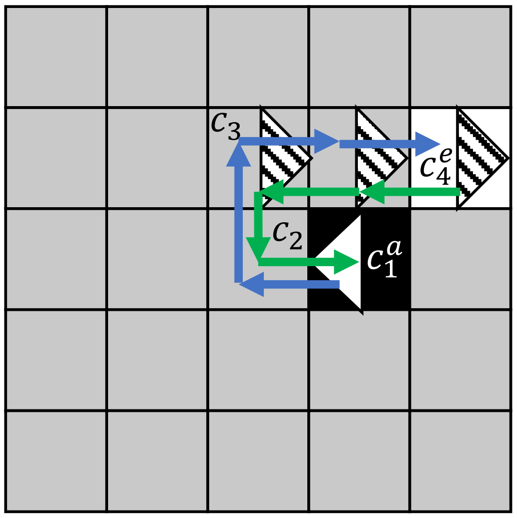


Figure 64: The examples of rules of processing N_2 and $N_2([N_3, N_4])$'s confirm messages. After c_4^e replies to the seek message from c_1^a, box movements are arranged. c_3 sends $N_2(N_3)$'s confirm message to c_2. Then c_2 sends N_3's confirm messages to c_1^a.

3.2.2.11 Rules to process fail message of N_2 and $N_2([N_3, N_4])$

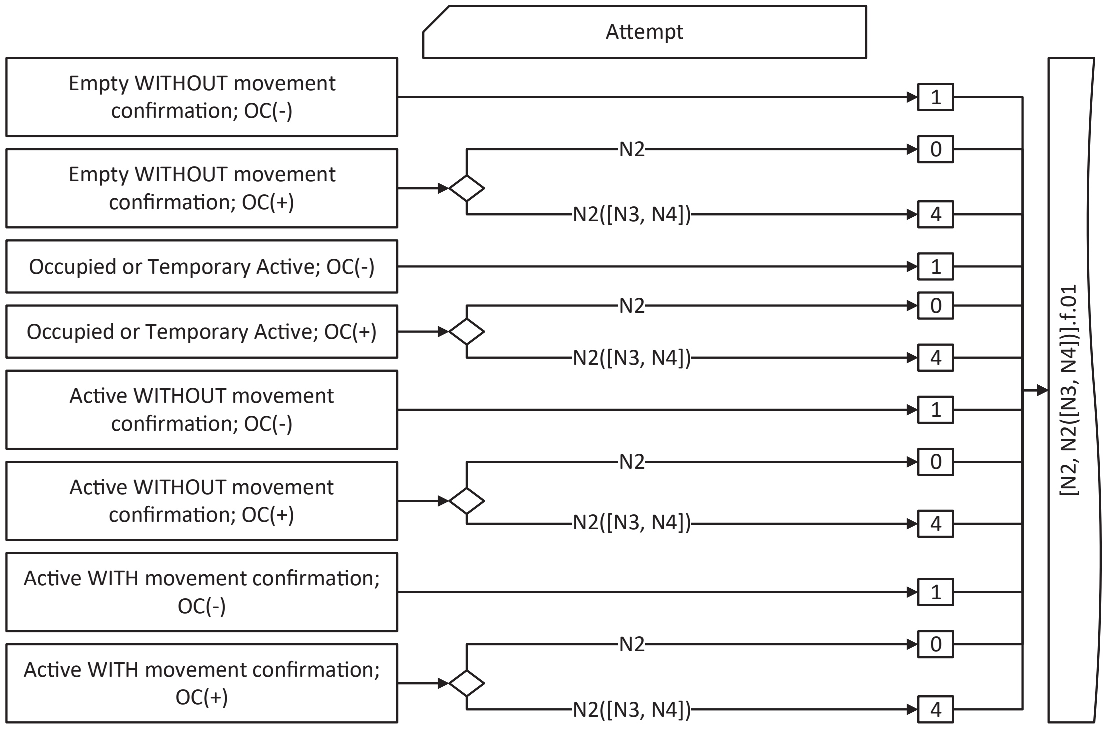

Attempt
Empty WITHOUT movement confirmation; OC(-)
Empty WITHOUT movement confirmation; OC(+)
Occupied or Temporary Active; OC(-)
Occupied or Temporary Active; OC(+)
Active WITHOUT movement confirmation; OC(-)
Active WITHOUT movement confirmation; OC(+)
Active WITH movement confirmation; OC(-)
Active WITH movement confirmation; OC(+)
N2
N2([N3, N4])
1
0
4
[N2, N2([N3, N4])].f.01

Figure 65: Rules of processing N_2 and $N_2([N_3, N_4])$'s fails messages.

Other negotiation rules are simple or have similar procedures to the above rules, so we put them in Appendix 7.4.2). The examples shown above to pass messages can be used as references for the additional negotiation rules.

3.2.3 Convey phase

To move a box, a conveyor module has to be in the state of "active with movement confirmation" or "temporary active." While a box is being moved, box information such as its transfer task is copied to its immediate destination. Then, the restriction of task re-assignment is also updated by increasing the counter of restrictions if they are not expired.

After GridHub executes activities in this phase, the whole iteration of control algorithms is completed, and the system enters a new iteration, continuing to move boxes to their targets or intermediate targets.

3.3 Examples of an entire iteration and a running system

The first example (see Figure 66 to 69) shows message interactions of GridHub's complete iteration. In this case, the only empty module's location belongs to category 4 (see Figure 39 and 40 for details). Hence, the first three attempts fail, but N_4 is successful.

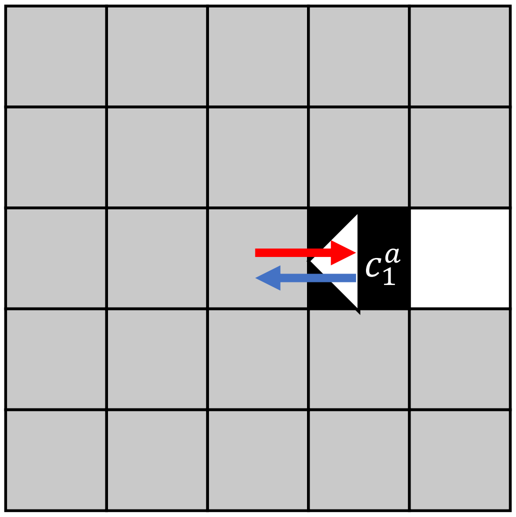


Figure 66: First example: N_1's example. This N_1 is failed directly by c_1^a's left neighbor according to $[N_1, N_1([N_2, N_3, N_4])].s.02$.

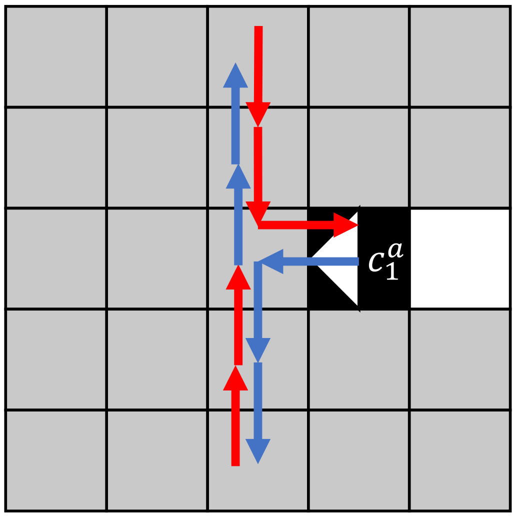


Figure 67: First example: N_2's example. N_1 is initiated by c_1^a, two $N_1(N_2)$ are initiated by c_1^a's left neighbor sequentially. Because the no empty modules are find, $N_1(N_2)$ and N_2 fail.

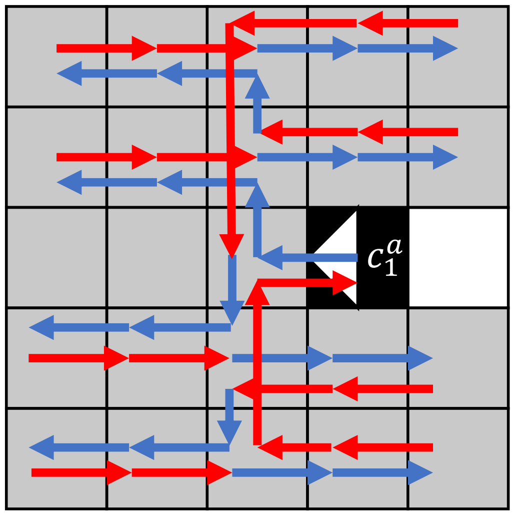


Figure 68: First example: N_3's example. $N_1(N_3)$ searches empty modules in every row except the row c_1^a is located.

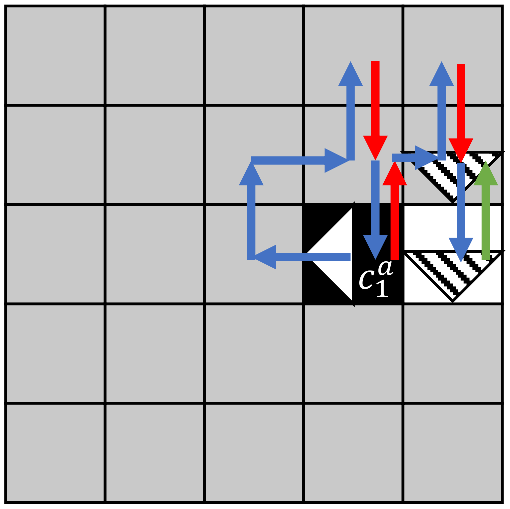


Figure 69: First example: N_4's example. Finally, $N_1(N_4)$ finds an empty module and arrange box movements. In the next iteration, the empty module's location can be searched by $N_1(N_3)$.

The second example (see Figure 70 to 72) uses snapshots from simulation directly. The targets of the two working boxes are the upper-left corner and upper-right corner respectively.

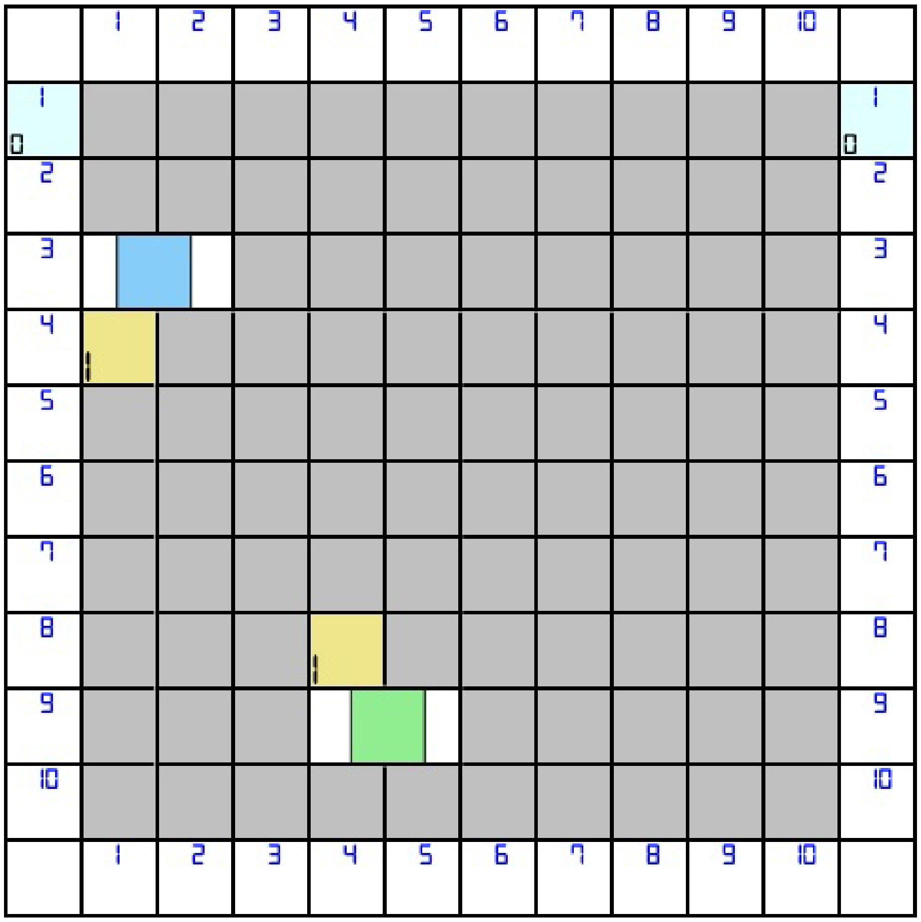


Figure 70: Second example: snapshots of a running simulation for GridHub (iteration 1). In the beginning of the iteration, the two conveyor modules that hold these boxes are both in "active without movement" states. The upper active module finds an empty module category 2 via N_2; the lower active module finds an empty module category 4 via N_4. (Colors - directions: blue - right, green - left, yellow - up, pink - down.)

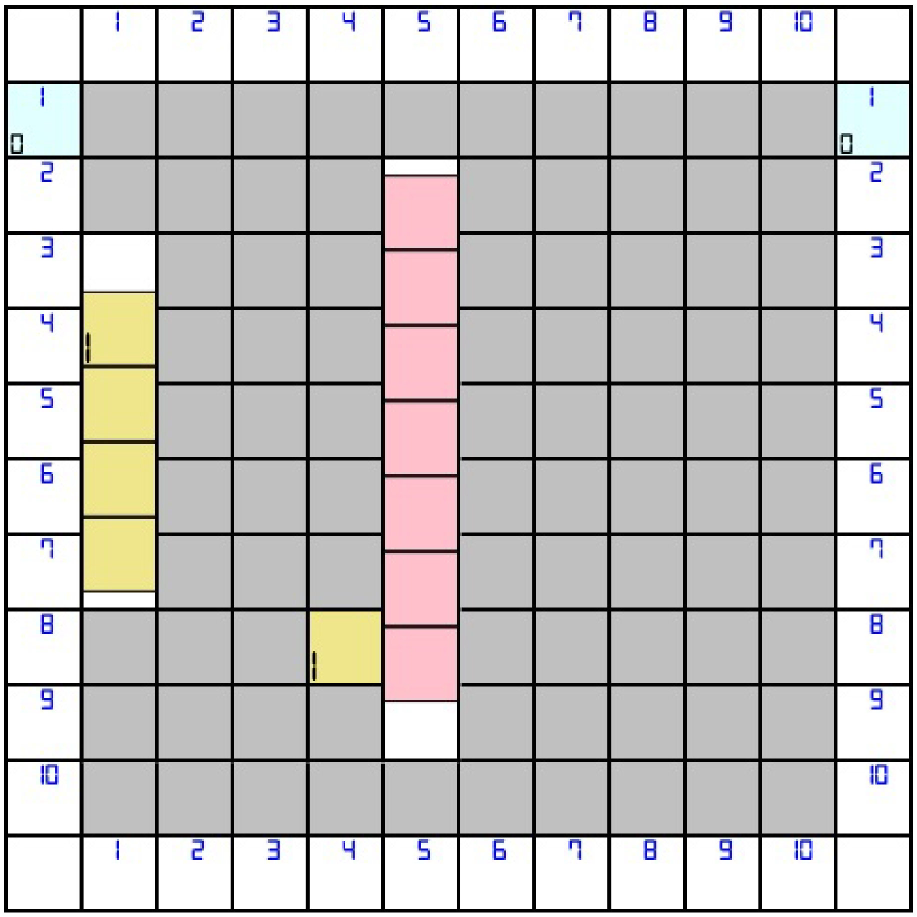


Figure 71: Second example: snapshots of a running simulation for GridHub (iteration 2). After previous negotiations, the empty module's locations are "switched." The upper active module can move its box to its immediate destination directly; Because the forward attempt is enabled, the upper active module also iniates N_2 and N_3, and the N_3 is successful. The lower active module initiates N_1 to N_3. (Colors - directions: blue - right, green - left, yellow - up, pink - down.)

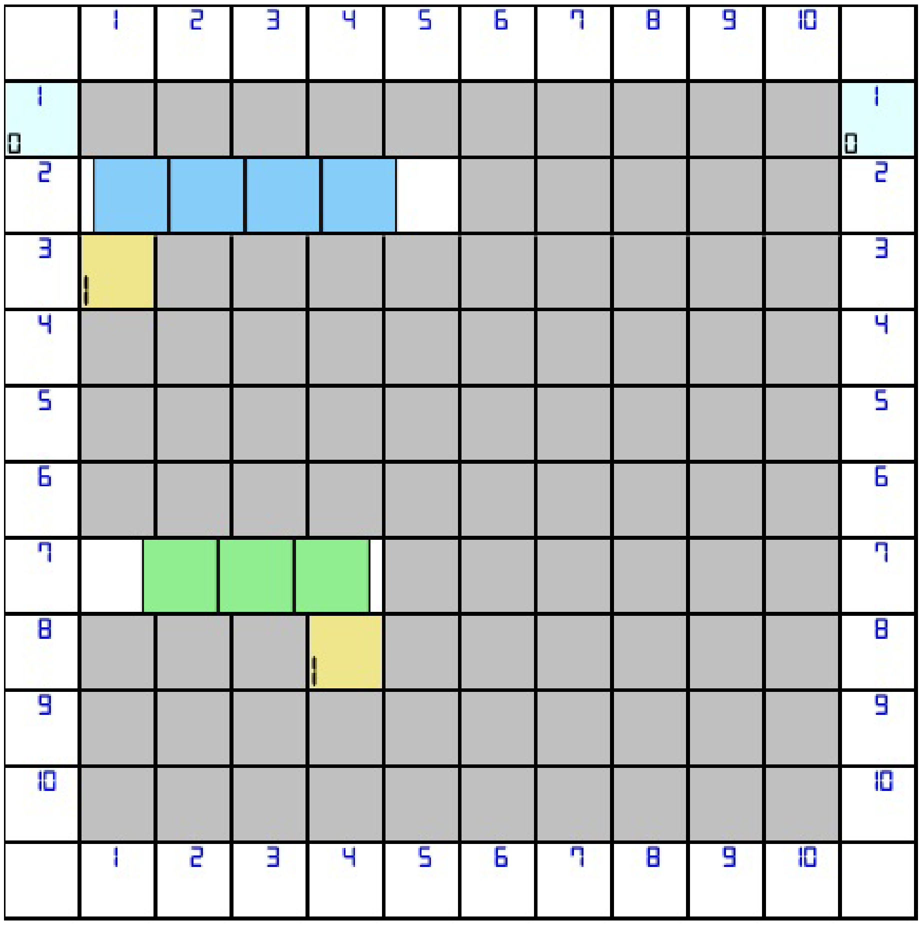


Figure 72: Second example: snapshots of a running simulation for GridHub (iteration 3). The active modules find empty modules category 2 respectively. (Colors - directions: blue - right, green - left, yellow - up, pink - down.)

CHAPTER 4

DEADLOCK and LIVELOCK

There are several pathological behaviors in GridHub. These behaviors can cause deadlock, livelock, and affect the transferring processes of active boxes. In this chapter, we review these behaviors and the related concepts first then we explore the deadlock and livelock in GridHub.

After an active conveyor runs the algorithms in the negotiate phase, there are three possible results: it is muted or muted and moved, it successfully completes an attempt, or it fails all attempts. The reason of failure is a *block*, which if left unresolved results in a system deadlock. Except the blocks, the other patterns of activities, are *negotiation behaviors* or *behaviors* of GridHub.

4.1 Preliminary consideration

4.1.1 Notations

An arbitrary GridHub is expressed as GH. The directions in GridHub are defined as "left," "right," "up," and "down" from the reader's point of view. Any conveyor module in a GridHub is denoted as c_i ($i \in \mathbb{N}$), and the set of all conveyors is C. When a GridHub is modeled by a grid graph, the graph can also be written as $GH(V, E)$. The conveyor modules are represented by the vertices V, and E is the set of edges (neighborhoods) among any two conveyor modules. To remain consistent, let $V = C$, and every element of V be expressed by the same symbol, such as c_i. Additional notations are listed in Table 7. Additionally, when an empty module initiates N_2, N_3, and N_4 if the forward attempt is enabled in GridHub, it is also considered an active conveyor module and expressed as c_i^a.

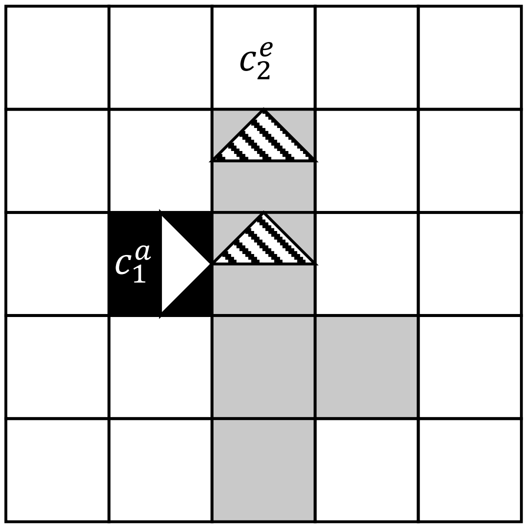
TABLE 7: Notations to describe GridHub ($i \in \mathbb{N}$).

Notations	Explanation
$Dr = \{L, R, U, D\}$	Set of directions in GridHub
x, y	A conveyor module's location in GH, which
	is recorded by the column and row it locates
c_i^a, c_i^e	Active, empty conveyor module
dr, $dr(\perp)$, $dr(\|)$	Element of Dr, perpendicular, opposite but parallel direction
d_{N_i} ($d_{N_i} \in Dr$)	Seek message passing direction of N_i
ad_{c_i} or $ad_{c_i^a}$	Active direction of an active conveyor module
C, C^e, C^a	Set of all, empty, active conveyor modules
M^e, M^a	The number of empty, active modules in GH
$M^e_{\min}$	Lower limitation of the empty conveyor modules
$M^a_{\max}$	Upper limitation of the active conveyor modules
rp_{dr}	Limitation to re-activate a conveyor module on dr
ni_{dr}	Counted time limits
	to re-activate conveyor module on dr
t, t_i	Time in GridHub that is counted by iteration

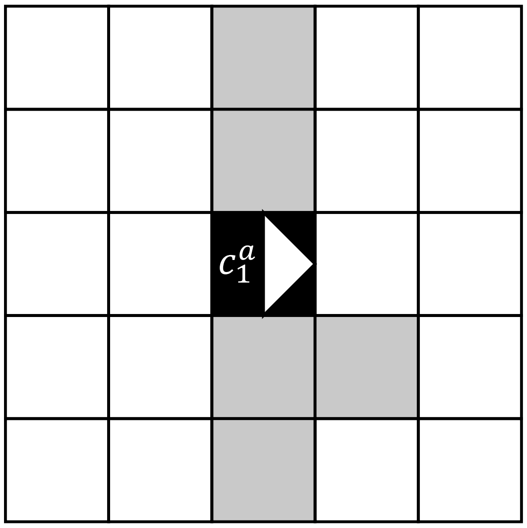
4.1.2 Potential paths of successful attempts

Some of the negotiation behaviors are related to the concept "potential paths," so we explain this concept first.

When there is one attempt in a GridHub, a confirmation message is passed back to the active conveyor module which initiated the attempt. Consequently, a group box movements is arranged along the path of passing the confirmation message. There are also potential movements along the same path in future iterations (Figure 73a to 73b).



(a) Iteration 1.



(b) Iteration 3.

Figure 73: Examples of box movements. In iteration 1, c_1^a initiates an N_2 attempt, and this attempt finds c_2^e. After the N_2 completed in the last iteration, the active box moves into its immediate destination by N_1 in iteration 3,. In these three iterations, box movements are arranged along the paths (shown in Figure 74, which are used to pass seek and confirmation messages.

The message passing path used to pass the successful seek messages and confirmation messages is called one *potential path of movements* or *potential path.* A potential path is recorded in a *4*-tuple $P_k = \{N_h,\ c_i^a,\ c_j^e,\ V_k\}$ $(h,\ i,\ j,\ k \in \mathbb{N})$: N_h is the attempt which the confirmation message belongs to, and we also say that N_h "generates" or "makes" P_k, or P_k is made by N_h; c_i^a is the active conveyor module that initiated N_i, so we can also say that c_i^a makes P_k. c_j^e is the empty conveyor module that replies with the confirmation message to N_h; The set of vertices of a potential path of P_k (except c_i^a and c_j^e) is V_k. In this dissertation, all of the potential paths are represented by orange arrows. The arrows are pointing to the c_j^e. Because some attempts have nested attempts, a potential path made by these attempts can be divided into several parts based on which nested attempts the messages belong to (Figure 74).

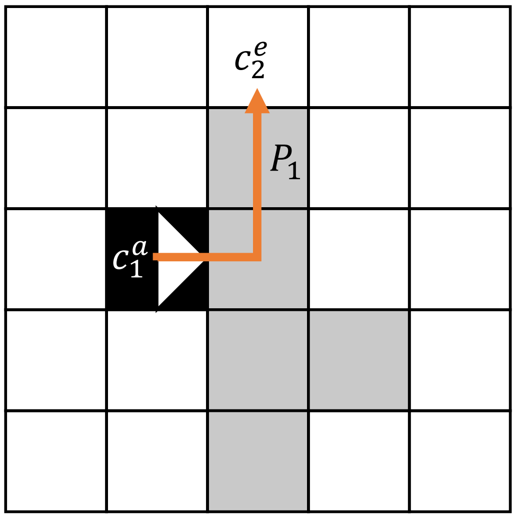


Figure 74: Potential path example. P_1 is made by N_2, and an $N_1(N_2)$ nests in this N_2, so this N_2 has two parts. After decomposing N_2, $P_{1,1}$ is made by $N_1(N_2)$, and $P_{1,2}$ is generated by N_2.

Conveyor modules covered by $P_{1,1}$ have to move their boxes in the current iteration (see Figure 73a). Hence, the part of a potential path that is made by N_1 or $N_1([N_2, N_3, N_4])$ is called the *movement part* of this potential path (Figure 75).

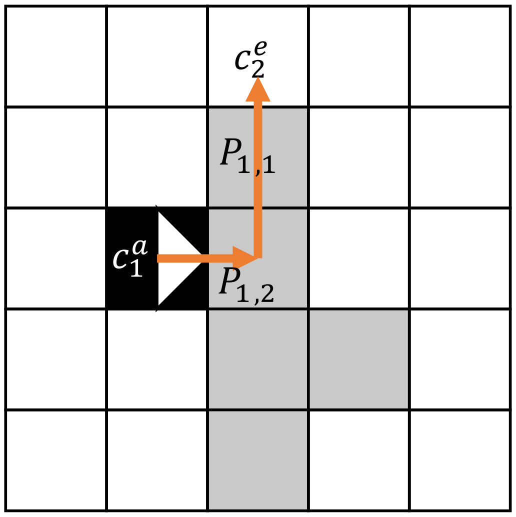


Figure 75: Example of movement part of a potential path. $P_{1,1}$ and $P_{1,2}$ are made by N_2 and N_1 respectively. $P_{1,2}$ is the movement part.

The potential paths are made by different attempts, and these attempts may be initiated sequentially. We use *level of a potential path* to distinguish the potential paths made by different attempts. The level of a potential paths is written as l_{P_k}. The higher level potential paths are made earlier than the lower level potential paths (Figure 76). The complete levels of potential paths are listed in Table 8.

TABLE 8: Levels of attempts and P_k.

Active Direction of module	Attempts	l_{P_k}
L or R	N_1	8
U or D	N_1	7
L or R	$N_1(N_2)$	6
U or D	$N_1(N_2)$	5
L or R	$N_1(N_3)$	4
U or D	$N_1(N_3)$	3
L or R	$N_1(N_4)$	2
U or D	$N_1(N_4)$	1

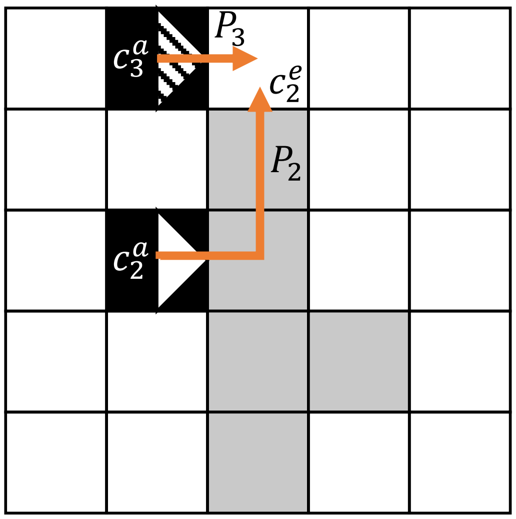
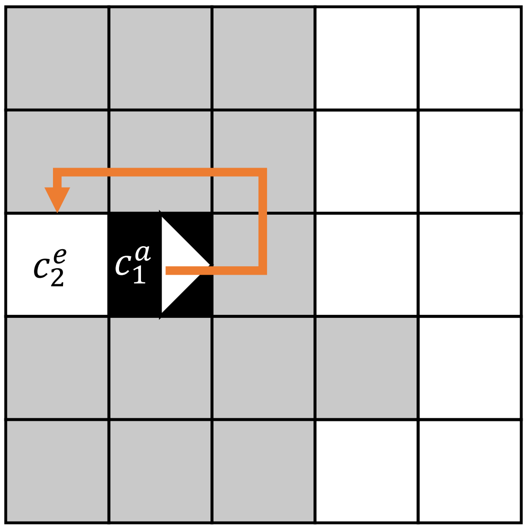
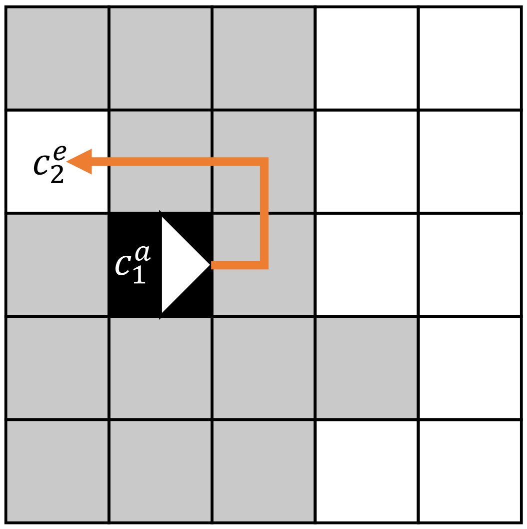


Figure 76: Examples of levels of different P_k. P_3 has higher level than P_2.

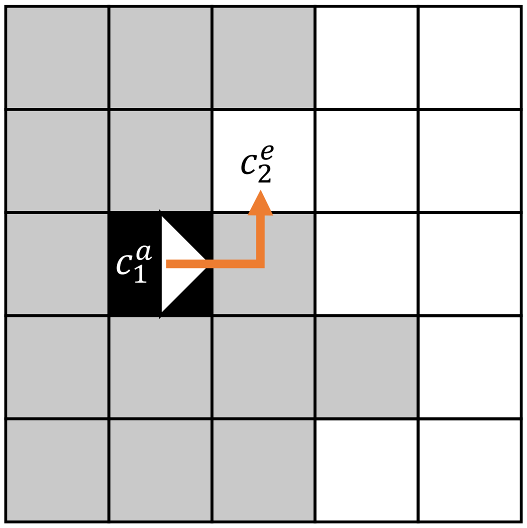
When the location of an empty module changes, the potential paths that are generated by the same active conveyor modules could be changed over a series of iterations. When there are no other paths to be made, the levels of these potential path change from low to high (Figure 77a to 77d), and this pattern is the ideal way of changing potential paths. The active boxes progress their transferring processes in this pattern. However, the negotiation behaviors generated by conveyor modules do not let the potential paths change in the ideal way, and the details are in Section 4.3.



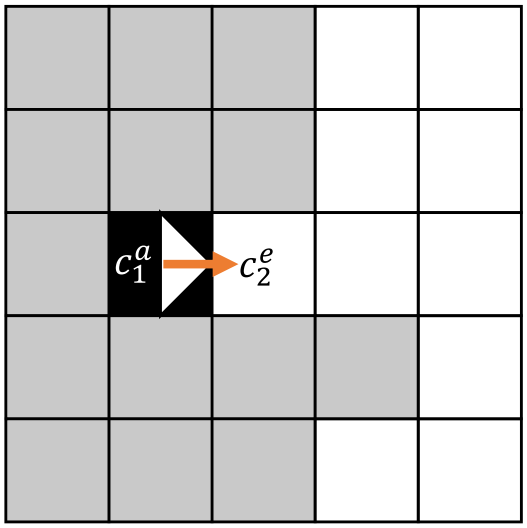
(a) iteration 1, $l_{P_k} = 2$



(b) iteration 2, $l_{P_k} = 4$



(c) iteration 3, $l_{P_k} = 6$



(d) iteration 4, $l_{P_k} = 8$

Figure 77: Potential path changes over iterations.

4.2 Blocks in GridHub

A *Block* in *GH* is the case that one conveyor module replies with a fail message to a seek message which is initiated by the other module in one iteration of GridHub. In other words, a seek message is "blocked" by another conveyor module. We describe a block in a *3*-tuple $B_{e,h} = \{N_i^b, c_j^a, c_k^b\}$ ($e \in \mathbb{N}$ indicates types of blocks, and $h, i, j, k \in \mathbb{N}$), where

- N_i^b is the attempt to which the blocked seek message belongs. For example, when a seek message of $N_1(N_2)$ is blocked, it means the nested N_1 seek message is blocked;
- c_j^a is the active conveyor module initiating N_i^b; and
- c_k^b is the blocking conveyor module that replies with a fail message to the blocked seek message.

After any elements of a block change, then the block changes to a new block, or the block disappears. Every $B_{e,h}$ may have a corresponding solution that can make $B_{e,h}$ disappear. The solution is called as $R_{e,h}$.

4.2.1 List of blocks and solutions

Blocks occur when seek messages receive fail messages. We organize the blocks according to all of the failed actions in the negotiation rules (see Figure 78). However, we do not account for the failed actions caused by completing all nested attempts ("ANC(+)" in the flowcharts); other failed actions caused them. For example, the module that initiates $N_1(N_2)$ has to reply N_2 with a fail message after the two nested attempts fail.

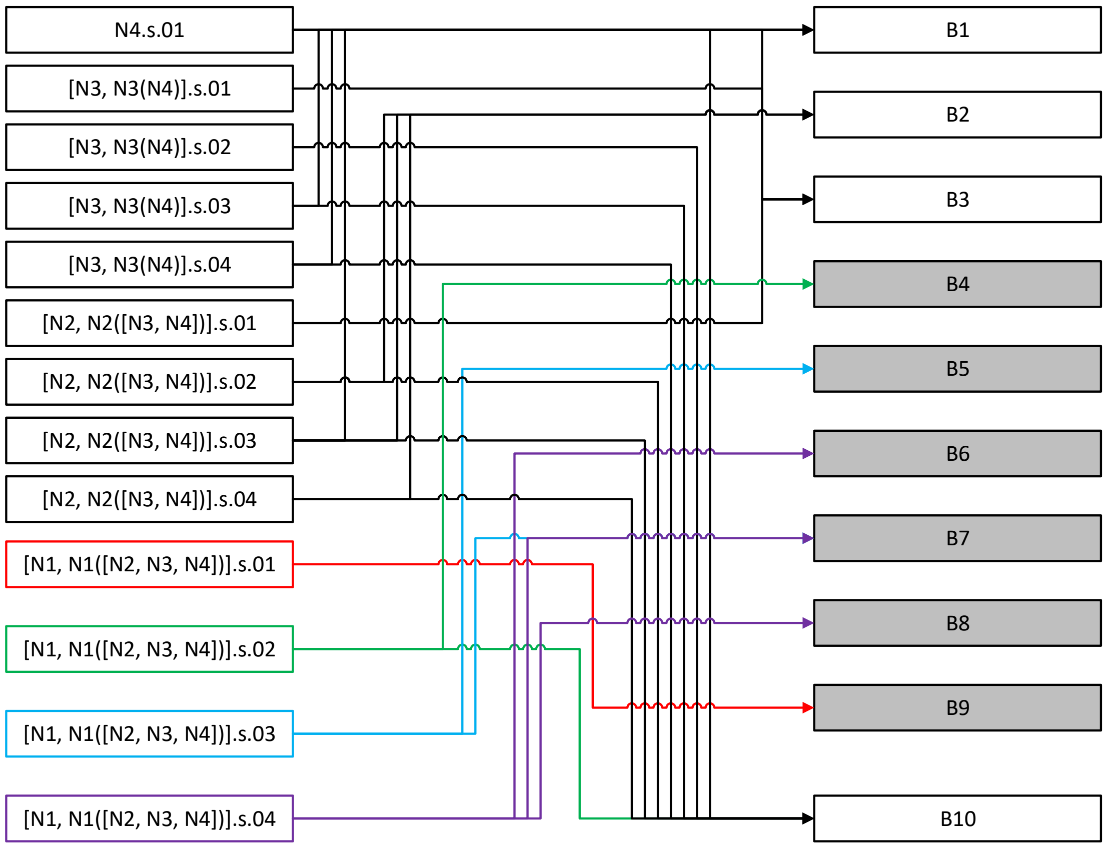


Figure 78: Block and rule relationship. The blocks that are related to N_1 or $N_1([N_2, N_3, N_4])$ are colored in grey. The rules that are related to N_1 or $N_1([N_2, N_3, N_4])$ are in different colors.

$B_{1,h}$ c_k^b's state is active or active with movement confirmation. It replies with a fail message to a seek message of N_2, N_3 and N_4 due to c_k^b's active direction being the same as the seek messages' passing direction (see rule $[N_2, N_2([N_3, N_4])].s.03$, $[N_2, N_2([N_3, N_4])].s.04$, $[N_3, N_3(N_4)].s.03$, $[N_3, N_3(N_4)].s.04$, and $N_4.s.01$ for details).

$R_{1,h}$: c_k^b can also initiate attempts to move its own box. When c_k^b can move boxes, the other conveyor modules that are "behind" it can also move their boxes after the N_1 is successful.

$B_{2,h}$ c_k^b's state is temporary active, active with or without movement confirmation, or occupied. When it is waiting for response of the other $N_1(N_2)$'s seek message, or it has a movement confirmation, it replies with a fail message to the seek message of

N_2 (see $[N_2, N_2([N_3, N_4])].s.02$, $[N_2, N_2([N_3, N_4])].s.03$, $[N_2, N_2([N_3, N_4])].s.04$ for details).

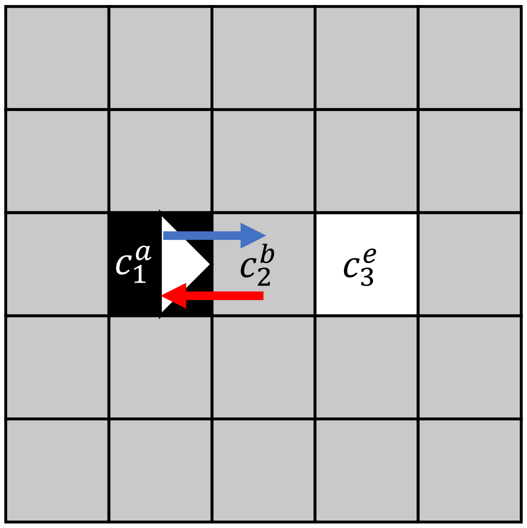
$R_{2,h}$: This block case will disappear after c_k^b moves its box out, or there is no other seek message waiting for response.

$B_{3,h}$ c_k^b's state is empty with or without movement confirmation. It replies with a fail to all of the seek messages except $[N_1, N_1([N_2, N_3, N_4])]$ (see rule $[N_2, N_2([N_3, N_4])].s.01$, $[N_3, N_3(N_4)].s.01$, and $N_4.s.01$ for details).

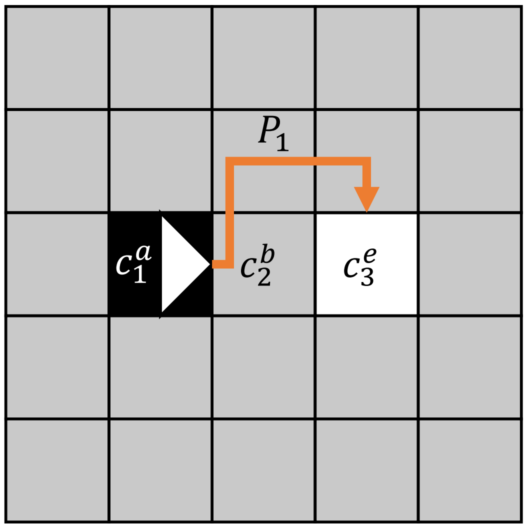
$R_{3,h}$: after a box moves to c_k^b and changes its state to occupied, then the block disappears.

$B_{4,h}$ c_k^b's state is occupied. It blocks a seek message of N_1 initiated by c_i^a according to $[N_1, N_1([N_2, N_3, N_4]).s.02]$.

$R_{4,h}$: c_i^a can also access other empty modules via N_2, N_3, or N_4. An example is in Figure 79.



(a) $B_{4,1}$. c_2^b blocks N_1 that is initiated by c_1^a.

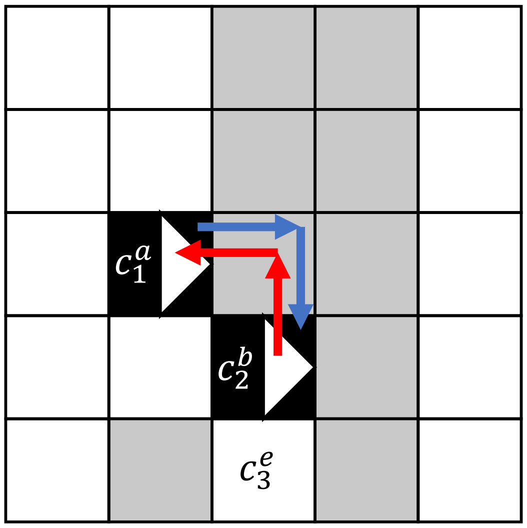


(b) $R_{4,1}$. c_1^a can use attempt N_4 to access c_3^e, and the potential path is P_1.

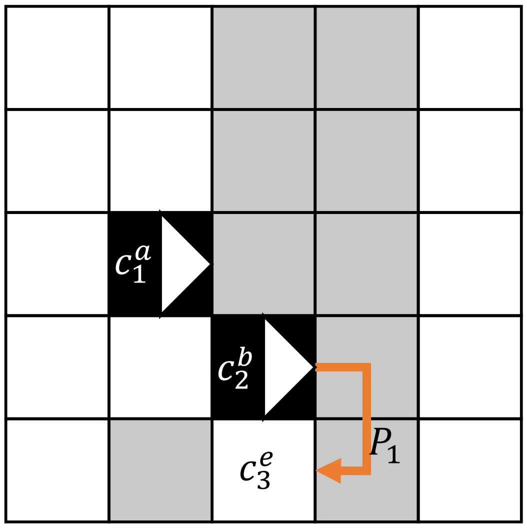
Figure 79: $B_{4,1}$ and $R_{4,1}$.

$B_{5,h}$ c_k^b's state is active without movement confirmation. It blocks a seek message of $N_1([N_2, N_4])$ initiated by c_j^a, when c_k^b's active direction is perpendicular to the message passing direction (see $[N_1, N_1([N_2, N_3, N_4])].s.03$ for details).

$R_{5,h}$: let the empty module which is going be reached by the blocked seek message $[N_1, N_1([N_2, N_3, N_4])].s.01$ in $B_{5,h}$ to be c^e_l. Another seek message of $N_1(N_3)$ initiated by c^b_k can also reach c^e_l in the same iteration or in later iterations. After c^b_k moves its box, the block disappears. An example is in Figure 80.



(a) $B_{5,1}$. c^b_2 blocks $N_1(N_2)$ that is initiated by c^a_1 and passed toward c^e_3.



(b) $R_{5,1}$. c^b_2 can access c^e_3 by $N_1(N_3)$ later, and the potential path is P_1.

Figure 80: $B_{5,1}$ and $R_{5,1}$.

$B_{6,h}$ c^b_k's state is active with movement confirmation or temporary active. It blocks a seek message of $N_1([N_2, N_3, N_4])$ which is initiated by c^a_j, because it has movement confirmation perpendicular to the message passing direction. The rule that makes this block is $[N_1, N_1([N_2, N_3, N_4])].s.04$.

$R_{6,h}$: after the movement of c^b_k's box is completed, the block disappears.

$B_{7,h}$ c^b_k's state is active with or without movement confirmation, or temporary active. It blocks a seek message of $N_1([N_2, N_3, N_4])$ initiated by c^a_j, when the seek message's passing direction is opposite to its active direction, and the message passing direction has lower priority than c^b_k's active direction. The rule according to is $[N_1, N_1([N_2, N_3, N_4])].s.03$ and $[N_1, N_1([N_2, N_3, N_4])].s.04$

$R_{7,h}$: If there is no movement confirmation of c^b_k, the block disappears in the later iterations when the seek message's passing direction has higher priority than c_k^b's active direction; if c_k^b has movement confirmation, then the block disappears after c_k^b moves its box.

$B_{8,h}$ c_k^b's state is active with movement confirmation, or temporary active. It blocks a seek message of $N_1([N_2, N_3, N_4])$ initiated by c_j^a, if the confirmation is not marked by N_1 and the confirmation direction is as the same as the message passing direction (see $[N_1, N_1([N_2, N_3, N_4])].s.04$).

$R_{8,h}$: after c_k^b completes its box's movement, the block disappears.

$B_{9,h}$ c_k^b' state is empty with movement confirmation. It replies with a fail message to a seek message of N_1 or $N_1([N_2, N_3, N_4])$ when the message passing direction is different than the confirmation direction. It can also reply with a fail message to the seek message of $N_1([N_2, N_3, N_4])$ in the same message passing direction when its confirmation is not made by N_1. The rule is $[N_1, N_1([N_2, N_3, N_4])].s.01$

$R_{9,h}$: the block case will disappear after the movement of c_k^b completes.

$B_{10,h}$ When c_i^b is occupied and located at one of the system borders, c_k^b can reply with a fail messages to all of the seek messages of N_2, $N_2([N_3, N_4])$, N_3, $N_3(N_4)$, N_4, N_1, $N_1([N_2, N_3, N_4])$. All rules related to system border conditions can cause this block. An example of this block is in Figure 81.

$R_{10,h}$: the reason of this block to occur is that other active modules have "used" empty modules. Hence, when the other active module moves out of the system, c_j^a eventually has chance to find an empty module.

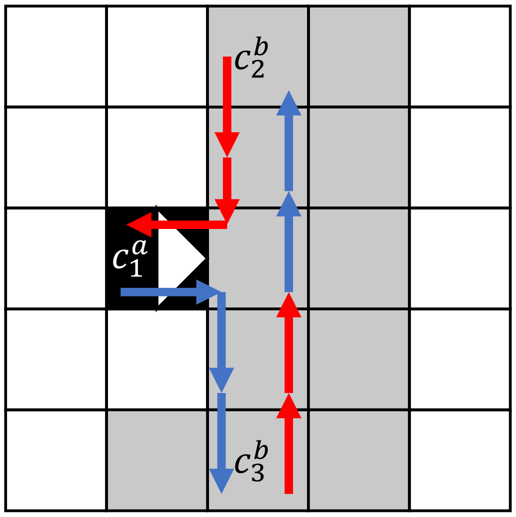


Figure 81: $B_{10,1}$. Both c_2^b and c_3^b can reached by the seek messages of $N_1(N_2)$. They have to reply with fail messages because they are on the system border.

4.2.2 Starvation of empty conveyor modules

In $B_{10,h}$, the reason of it is that no empty conveyor modules are available, so we also call this case *temporary starvation*. $B_{10,h}$ always occurs at the end of a process to pass a seek message. Because we have shown that $B_{10,h}$ is temporary, we consider it a fact and omit to mention it in the deadlock discussion in Section 4.4.

To reduce occurrence of temporary starvation, we have to either increase the number of empty conveyor modules (effects are discussed in Chapter 5), or use other attempts to arrange box movements into other empty conveyor modules, or wait for the active modules that using the empty modules move away.

4.3 Negotiation behaviors in GridHub

When there is only one active, and one or more empty conveyor modules, in a GridHub, the active conveyor module either transfers its box to the box's immediate destination, or tries to change the location of empty modules by arranging the box's movements. In every iteration, the movements of the boxes are necessary, so the transfer process keeps making progress even when the active box is waiting for space to move. The scenario described above is the ideal case to move an active box. However, it is not always possible to have exactly one active conveyor module in a

GridHub. Below we describe the negotiation behaviors among active modules and how active boxes' transferring processes are affected by them.

4.3.1 List of negotiation behaviors

All of the negotiation behaviors listed below are related to two conveyor modules. First, these two conveyors form a behavior. The conveyor module that makes an attempt to affect the other module is called the *generator* or the *maker* conveyor module of the behavior. The conveyor module affected by a behavior is called the *included* conveyor module.

Direct mute When two active conveyor modules have opposite active directions and face each other, one must be muted according the negotiation rules. Then, they form the behavior of *direct mute* (Figure 82). The conveyor module that is directly muted may not move its box.

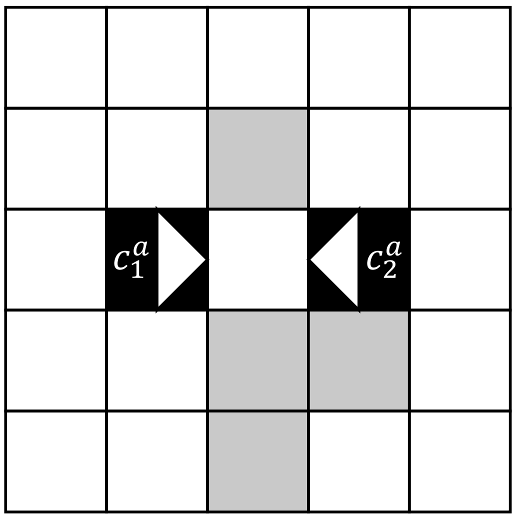


Figure 82: Direct Mute example. Suppose $pd = L$, then c_2^a is muted. We can also state that c_1^a makes the mute behavior directly, and c_2^a includes a direct mute.

Indirect mute When an active conveyor module that has no movement confirmations and confirms a movement of opposite moving directions to the active direction, it is *indirect muted* (Figure 83).

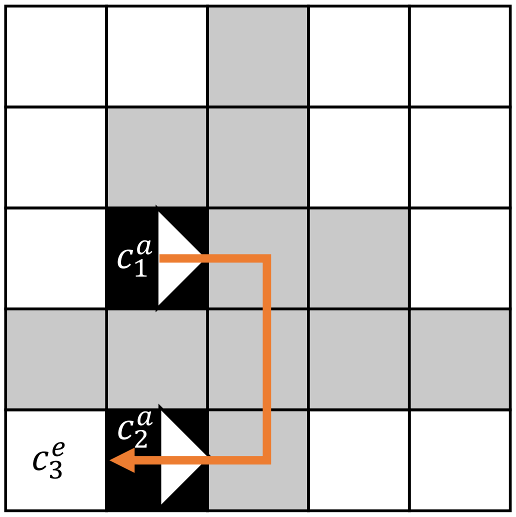


Figure 83: Indirect mute example. c_1^a makes a potential path, and it indirectly mutes c_2^a, or c_2^a includes an indirect mute behavior. c_2^a must move its box in the current iteration.

Cross mute When an active conveyor module has no movement confirmations and it confirms a movement to a direction which is perpendicular with its active direction, this active conveyor module is *cross muted* (Figure 84).

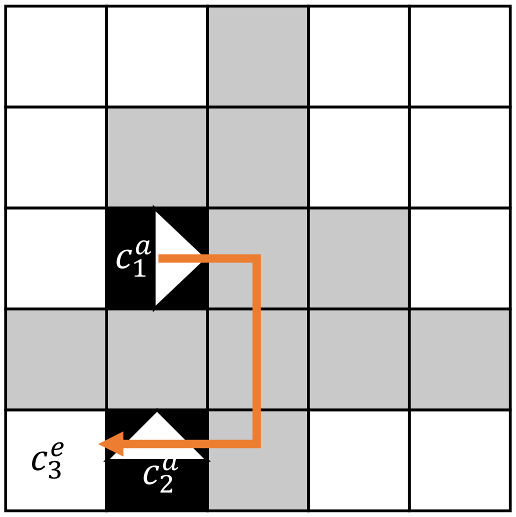


Figure 84: Cross mute example. A potential path is made by c_1^a, and c_2^a is crossly muted. c_2^a must move its box in the current iteration.

Indirect push If an active box does not have any movement confirmations or confirmations of successful attempts but it confirms movements of other attempts

(N_2 to N_4) along its active direction, this conveyor module is in an *indirect push* case (Figure 85).

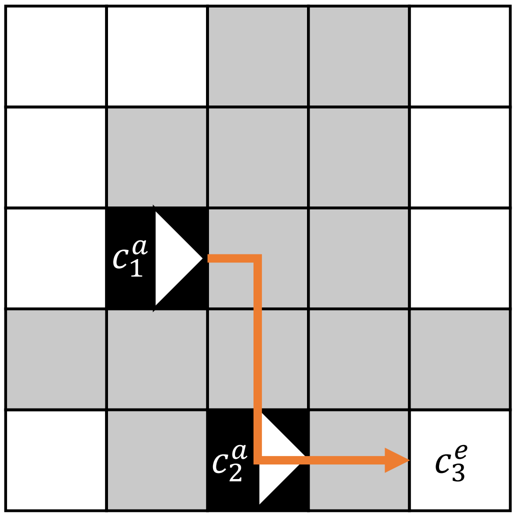


Figure 85: Indirect push example. c_2^a does not have any movement confirmations. A potential path made by c_1^a covers c_2^a, then c_2^a includes an indirect push behavior. c_2^a must move its box in the current iteration.

Overlapping of potential paths When two or more potential paths share movement parts, these potential paths are *overlapped*. According to the negotiation rule ($[N_1, N_1([N_2, N_3, N_4])].s.04$), only potential paths with level *1* or *2* can be overlapped by other potential paths with any levels (Figure 86).

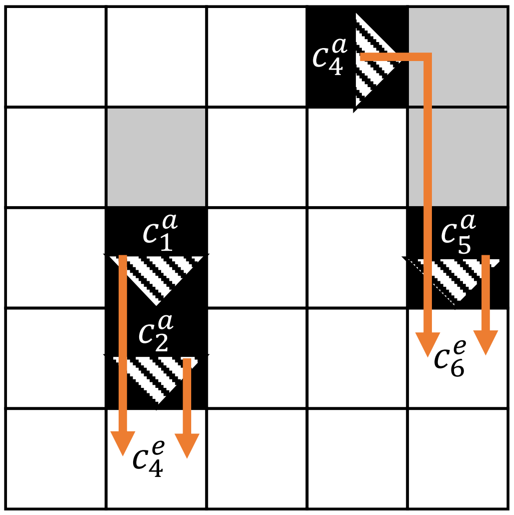


Figure 86: Overlapping examples. The potential path made by c_2^a is overlapped by the path made by c_1^a. Both of these paths are generated by attempt N_1, and levels are 2. The path made by c_5^a is also overlapped by the path made by c_4^a, but the path made by c_4^a has level 4.

Fake confirmation potential paths made by N_1 or $N_1([N_2, N_3, N_4])$ In Figure 87, P_1 and P_4 are potential paths made by c_1^a and c_3^a respectively. Now suppose the matched higher priority direction in this iteration is up(U). When c_5, c_6, and c_7 pass the confirmation messages, some confirmations in the opposite direction are already marked. Then, either of the following processes are possible, and they are called *fake confirmations*.

- While c_5, c_6, and c_7 are passing confirmation message back to c_1^a, these three conveyor modules have confirmed to move down; according to rule $[N_1, N_1([N_2, N_3, N_4])].c.02$ and $[N_1, N_1([N_2, N_3, N_4])].c.04$, the down movement confirmations of these conveyor modules are removed before marking the up movement confirmations.
- While c_5, c_6, and c_7 are passing a confirmation message back to c_3^a, these three conveyor modules have confirmed to move up; according to the same rule, no down movement confirmations can be placed on these conveyor modules.
- Some of c_5, c_6, and c_7 have confirmed to move down, but some have confirmed to move up. The first two processes are performed by these three conveyor

modules according to which direction they have confirmed to move their boxes.

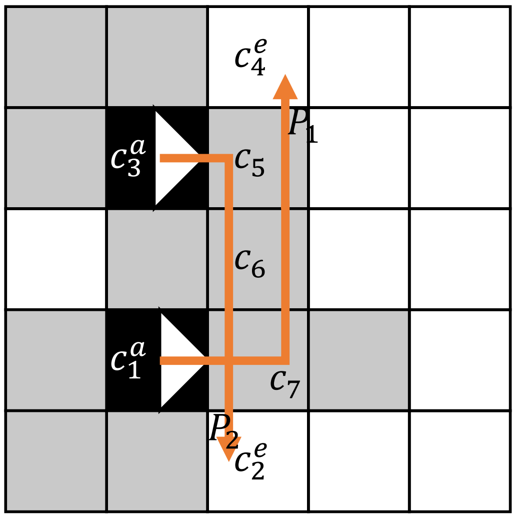


Figure 87: Fake confirmation examples.

Overwriting of potential paths made by $[N_2, N_3, N_4]$ When an active conveyor module has a confirmation of $[N_2, N_3, N_4]$, one $N_1([N_2, N_3, N_4])$ is made by another active conveyor module which arranges movement for it. Then this active conveyor module's potential path is *overwriting*. In other words, the moving part of another potential path covers this active conveyor module. There are two cases of this behavior (See Figure 88 and 89).

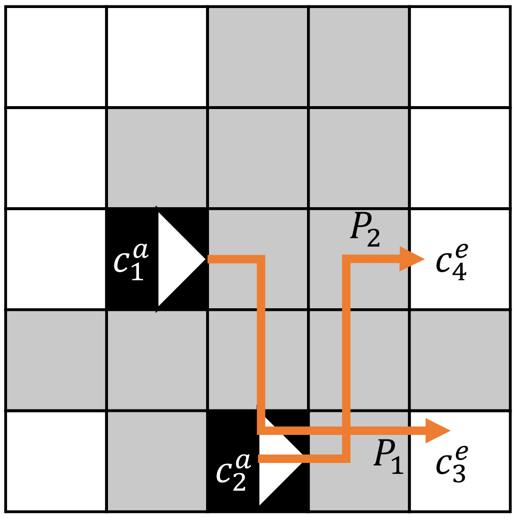


Figure 88: The same direction overwriting. P_2 is overwritten by another potential path; the movement direction of c_2^a's box after P_2 being overwritten is the same as its active direction.

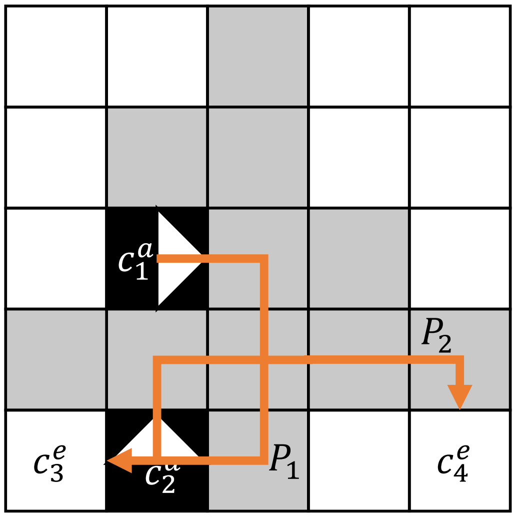


Figure 89: The different direction overwriting. The movement direction of c_2^a's box after P_2 being overwritten is different from its active direction. Furthermore, c_2^a may also be muted by these behaviors, and c_2^a must move its box in the current iteration.

4.3.2 Negotiation behaviors and transfer process

In GridHub, the Manhattan distance is the shortest distance from one conveyor module to another. The shortest distance to move a box from one conveyor to another is calculated by the number of iterations, following one

Manhattan path without any stops. However, it is almost impossible to move an active box to its target in the shortest time. The first reason is that GridHub is a high density storage system. Like other puzzle-based storage systems, the active box has to wait for other box movements to change the locations of empty conveyor modules. The other reason is the behaviors listed above delay the transfer process.

Several behaviors that can delay the transfer process of an active box include: direct mute, cross mute, indirect mute, overwriting potential paths to different directions (See Figure 89), and fake confirmation. The fake confirmation cannot change the location of an empty conveyor module as desired, and other b()ehaviors mute the active boxes.

The overlap of potential paths, indirect push, and the overwriting of potential paths in the same direction (see Figure 88) are the activities that prevent the delay of transfer process.

4.4 Deadlock

We list one observation of deadlock in the grid-based system in Chapter 2. In GridHub, if an empty conveyor replies with confirmation message to a seek message, box movements must be arranged. Since no loop routes are created for active boxes, eventually, active boxes can be transferred to their targets. If a seek message cannot receive a confirmation message, a fail message is its response. Then, deadlock is possible. We have summarized the failure cases of passing seeking messages as "blocks." In this section, we give the definition of deadlock according to blocks, and then we show that GridHub is deadlock free by proof that there will never be an indefinite block.

It is easy to imagine pathological cases in which GridHub will deadlock, but we exclude these cases by assumption (Figure 90). To avoid these cases in Figure 90, we make two important assumptions:

1. No new working boxes are generated, and,
2. No new entered boxes.

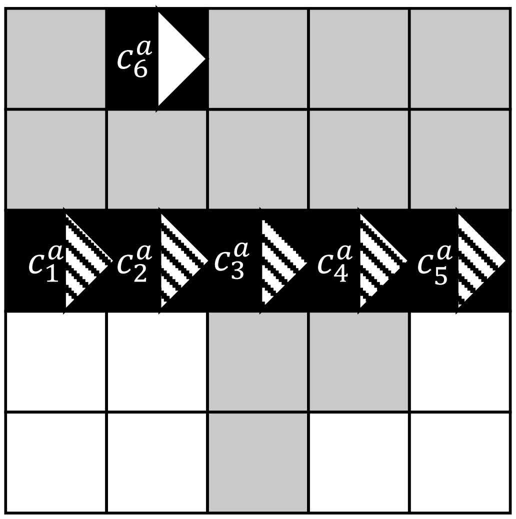


Figure 90: One example of GridHub without the assumptions. Suppose the new boxes are being added to the system; c_1^a to c_5^a keep moving their boxes out of the system, but c_6^a cannot move its box.

There are two categories of deadlock—global and local. The *global deadlock* case of *GH* is that at least one blocks exist and prevent all active boxes' transferring processes, and these blocks continue indefinitely.

For every active conveyor module, we also check whether the attempts it initiates cannot succeed forever. A *Local deadlock* in occurs when any block is present and it continues indefinitely, while other active boxes' transferring processes are not prevented.

Based on these definitions, we have two methods (Lemma 4.1 and 4.2) to judge whether *GH* is in a deadlock conidtions. The proofs follow directly from the definitions.

Lemma 4.1. *GH* is free of global deadlock iff at least one $B_{e,i}$ can be resolved.

Lemma 4.2. *GH* is free of local deadlock when every existing block can be resolved.

Then, we try to find whether *GH* is free of the both deadlock cases using induction. First, we consider the simplest case.

Lemma 4.3. In a GridHub *GH*, if $M^a = 1$ and $M^e = 1$, then *GH* is global deadlock free and local deadlock free.

Proof. Let the active conveyor module is c^a_j and $ad_{c^a_j} = R$, let the empty module be c^e_l.

When $M^e = \mathit{1}$, according to the negotiation rules for N_{1} through N_{4}, all of the conveyor modules in *GH* can be reached by at least one seek message of N_{1} or $N_{\mathit{1}}([N_{\mathit{2}}, N_{\mathit{3}}, N_{\mathit{4}}])$ initiated by the c^a_j.

The set of conveyor modules which can be reached by the seek messages of N_{1}, $N_{\mathit{1}}(N_{\mathit{2}})$, $N_{\mathit{1}}(N_{\mathit{3}})$ and $N_{\mathit{1}}(N_{\mathit{2}})$ are $V_{\mathit{1}} = \{c^a_j\}$, $V_{\mathit{2}} = \{c|x = x, y \neq y\}$, $V_{\mathit{3}} = \{c|x \neq x, y \neq y\}$, and N_{4} is $V_{\mathit{4}} = \{c|xneqx, y = y\}$ respectively. If $C^s = \{c_i|c_i \notin C^a, c_i \notin C^e\}$, and $V_{\mathit{1}} \cap V_{\mathit{2}} \cap V_{\mathit{3}} \cap V\mathit{4} = V' \supseteq C^s$, we can conclude that all of conveyor modules except c^a_j, c^e_l can be reached by one or more seek messages of N_{1}, $N_{\mathit{1}}(N_{\mathit{2}})$, $N_{\mathit{1}}(N_{\mathit{3}})$ and $N_{\mathit{1}}(N_{\mathit{2}})$. According to rule $[N_{\mathit{1}}, N_{\mathit{1}}([N_{\mathit{2}}, N_{\mathit{3}}, N_{\mathit{4}}])].s.\mathit{01}$, when a seek message is passed to c^e_l, it replies confirm message.

Apply Lemma 4.1, *GH* is global deadlock free; Apply Lemma 4.2, *GH* is local deadlock free.

For the other cases (when $ad_{c^a_i} = L$, $ad_{c^a_i} = U$ or $ad_{c^a_i} = D$), the same conclusion can be obtained by the reasoning steps above. □

The next step is checking all of the possible blocks and conclude that all of the possible blocks or combination of blocks cases can disappear.

Lemma 4.4. For an empty conveyor module c^e_l in *GH*, if at least one message is intended to be sent to it, and these messages are blocked before they arrive, the blocks are always possible to be resolved.

Proof. When some seek messages of some attempts are intended to be sent to c^e_l, a blocking conveyors replies with fail messages before the arrive at c^e_l. These block cases described above can be $B_{\mathit{1},h}$, $B_{\mathit{2},h}$, $B_{\mathit{4},h}$, $B_{\mathit{5},h}$, $B_{\mathit{6},h}$, $B_{\mathit{7},h}$, and $B_{\mathit{8},h}$.

Let the blocking conveyor in these blocks be c^b_k, and the active conveyor module in these blocks be c^a_j. Except $B_{\mathit{4},h}$ and $B_{\mathit{5},h}$, the reasons of the above blocks are either some of the c^b_k have movement confirmations or the message passing direction of the blocked messages do not have higher priority. Hence, those blocks will disappear after the confirmed movements are accomplished or the message passing directions have higher priority.

In the case of $B_{4,h}$, the c_j^a can access other empty conveyor modules in other attempts (see $R_{4,h}$ for details) However, the seek messages of other attempts can be in block cases of $B_{1,h}$, $B_{2,h}$, $B_{3,h}$, $B_{5,h}$, $B_{6,h}$, $B_{7,h}$, $B_{8,h}$, $B_{9,h}$. Except $B_{5,h}$, the above blocks can disappear after movements are complete or the blocked messages passing directions have higher priority.

In the case of $B_{5,h}$, the c_k^b of this block can send a seek message of $N_1(N_3)$ in the same iteration or future iterations (see the example in Figure 80b). However, all of the seek messages of N_3 can be in block cases of $B_{1,h}$, $B_{2,h}$, $B_{3,h}$, $B_{6,h}$, $B_{7,h}$, $B_{8,h}$, $B_{9,h}$. These blocks can disappear after movements are completed or the blocked messages passing directions have higher priority. □

Lemma 4.5. When an empty module c_l^e is the blocking conveyor module of some $B_{e,h}$, these blocks are always possible to disappears.

Proof. c_h^e could be the blocking conveyor of $B_{3,h}$ and $B_{9,h}$. The reason of these blocks are either c_k^b is empty, or c_k^b has movement confirmation. After a box moves to c_k^b or the confirmed movement is completed, the blocks are resolved. □

Lemma 4.6. In GH, when $M^e \geq 1$ and $M^a \geq 2$, if c_j^a is the blocking conveyor modules of one or more $B_{e,h}$, these blocks can always be resolved.

Proof. When c_j^a is the blocking conveyor module of these blocks, the possible blocks cases are $B_{1,h}$, $B_{2,h}$, $B_{5,h}$ to $B_{8,h}$. Except $B_{5,h}$, the reason of above blocks are either some of the c_k^b have confirmed movements or the blocked messages passing directions do not have higher priority. These blocks will disappear after the confirmed movements are done or the message passing directions have higher priority.

For $B_{5,h}$, let c_l^e be the empty conveyor module, which the blocked seek message is going to reach, then c_j^a can send a seek message of $N_1(N_3)$ to reach the c_l^e and arrange group of box movements. However, all of the seek messages of N_3 can be in block cases of $B_{1,h}$, $B_{2,h}$, $B_{3,h}$, $B_{6,h}$, $B_{7,h}$, $B_{8,h}$, $B_{9,h}$. The reason of above blocks are either some of the c_k^b have confirmed movements or the blocked messages passing directions do not have higher priority. These blocks will disappear after the confirmed movements are done or the message passing directions have higher priority □

Lemma 4.7. In GH, when $M^e \geq 1$ and $M^a \geq 2$, if c_j^a is the conveyor module that initiates N_i^b in $B_{e,h}$, $B_{e,h}$ can always be resolved.

Proof. When c_j^a is original conveyor modules of these blocks, c_j^a can be in all of the cases. Besides $B_{5,h}$, the blocks will disappear after the confirmed movements are done or the messages passing directions have higher priority.

For $B_{5,h}$, let c_l^e is the conveyor module which the blocked seek message is going to reach, then c_j^a can send a seek message of $N_1(N_3)$ to reach c_l^e and arrange box movements. However, all of the seek messages of N_3 can be in the block cases of $B_{1,h}$, $B_{2,h}$, $B_{3,h}$, $B_{6,h}$, $B_{7,h}$, $B_{8,h}$, and $B_{9,h}$. The reason of above blocks are either some of the c_k^b have confirmed movements or the blocked messages passing directions do not have higher priority. These blocks will disappear after the confirmed movements are done or the messages passing directions have higher priority □

Finally, based on the lemmas, we prove that GH is free of both the local and global deadlock.

Theorem 4.8. GH is globally deadlock free when $|C| \geq 2$, $M^a \geq 1$ and $M^e \geq 1$.

Proof. Proof by induction on both M^e and M^a.

When $M^e = 1$ and $M^a = 1$, apply Lemma 4.3 directly, and the proof is done.

When $M^e = 1$ and $M^a > 1$: Suppose GH is global deadlock free when $M^a = |C^a| - 1$ $(a \geq 2)$ and $M^e = 1$. Except c_0^a, the other active modules' locations are fixed in an iteration. Several scenarios are possible after fixing the location of c_0^a:

1. c_0^a is not the blocking conveyor of any other $B_{e,h}$, GH keeps global deadlock free.

2. c_0^a is a blocking conveyor in at least one $B_{e,h}$, and the blocked seek messages is intended to be sent to c_l^e. Apply Lemma 4.4 and Lemma 4.1, then GH is global deadlock free.
3. c_0^a is the conveyor module, which initiates the blocked seek messages, in at least one $B_{e,h}$, and the blocked seek messages is intended to be sent to c_l^e. Apply Lemma 4.4 and Lemma 4.1, then GH is still global deadlock free.

When $M^e > 1$ and $M^a = |C^a|$: Suppose GH is global deadlock free When $M^e = |C^e| - 1$ ($|C^e| \geq 2$) and $M^a = |C^a|$ ($M^a \geq 2$). Except c_0^e, the other empty modules' locations are fixed in an iteration. Following scenarios are possible after fixing the location of c_0^e:

1. If some seek messages are intended to be sent to c_0^e, and these attempts are blocked. Apply Lemma 4.4 and Lemma 4.1, then GH is still global deadlock free.
2. c_0^e is the blocking conveyor in some $B_{e,h}$. Apply Lemma 4.5 and Lemma 4.1, then GH is still global deadlock free
3. c_0^e does not in both of the scenarios above, no new blocks appear. Apply Lemma 4.1, and the GH is still global deadlock free.

□

Theorem 4.9. GH is local deadlock free when $|C| \geq 2$, $M^a \geq 1$ and $M^e \geq 1$.

Proof. Proof by induction on both M^e and M^a.

When $M^a = 1$ and $M^e = 1$, apply Lemma 4.3 directly, and the proof is done.

When $M^e = 1$ and $M^a \geq 2$: Suppose GH is local deadlock free when $M^e = 1$ and $M^a = |C^a| - 1$. Except c_0^a, the other active modules' locations are fixed in an iteration. Several scenarios are possible after fixing the location of c_0^a:

1. c_0^a is not the blocking conveyor module of any other $B_{e,h}$, there is no new blocks appears. Apply Lemma 4.2 and GH keeps local deadlock free.
2. c_0^a is a blocking conveyor module of at least one $B_{e,h}$. Apply Lemma 4.6 and Lemma 4.2, then GH is still local deadlock free.

3. c^a_0 is the conveyor module initiates the blocked seek message of one $B_{e,h}$. Apply Lemma 4.7 and Lemma 4.2, then GH is local deadlock free.

When $M^e > 1$ and $M^a = |C^a|$: Suppose GH is local deadlock free when $M^e = |C^e| - 1$ and $M^a = |C^a|$. Except c^e_0, the other empty modules' locations are fixed in one iteration. Several scenarios are possible after locate c^e_0.

1. If some seek messages are intended to be sent to c^e_0, and these attempts are blocked. Apply Lemma 4.7 and Lemma 4.2, then GH is still local deadlock free.
2. c^e_0 becomes blocking conveyors in some $B_{e,h}$. Apply Lemma 4.5 and Lemma 4.2, then GH is still local deadlock free
3. c^e_0 does not in both of the scenarios above, no new blocks appears. Apply Lemma 4.2, and the GH is still global deadlock free.

□

4.5 Livelock

In this section, we first define livelock and prove the necessary condition to cause livelock in GridHub. Then, we examine the scenarios where mute and the activation of active conveyor modules can cause livelock. We prove that a GridHub is conditional livelock free. Finally, we discuss livelock in GridHub when limitations of active conveyor modules are higher. The methods used to reduce livelock risks are also described.

4.5.1 Introduction of livelock

The term "livelock" comes from research in computer network routing (Toueg, 1980; Gravano et al., 1994). However, those livelock problems are not comparable to livelock in the grid-based systems, because there are no buffers for boxes in every conveyor module. In published works of the other grid-based systems, the livelock processes were also discussed (see Chapter 2). Schwab (2015) studied livelock in the AGV system. Livelock was the case that some modules in the system performed endless "circling movements." Seibold (2015) defines livelock in material handling systems as items which repeat movements but which cannot not be moved to their targets.

In GridHub, livelock can also occur. The *livelock* in GridHub is that an active conveyor having the same successful attempt repeatly, but its later attempt is prevented. The following actions can prevent the transfer process of active boxes. First, the blocks can prevent the transfer process of active boxes, but these have been shown to be temporary. Second, "fake confirmation" behavior can prevent the transfer process, but it is also temporary. The actions that can permanently prevent the transfer processes of active boxes are the behaviors that include mute activities. Additionally, when a conveyor has these behaviors, it will re-activate in future iterations. Then, mute and the re-activation can be considered together as *mute and re-activation* behaviors. The make conveyor module mutes the included conveyor module in the mute behaviors, and the included conveyor will re-activate in future iteration. When this behavior is not *repeatable*, the transfer processes of active boxes are only temporarily prevented. Hence, the only possible factor to prevent the transferring process is repeatable mute and reactivation behaviors. The term "repeatable" in this dissertation means one mute and activation behavior occurs at the same location in GridHub.

In the rest of this discussion, we use the same assumptions as in the deadlock discussion. We prove that repeatable mute and re-activation behaviors are the necessary conditions to prevent the transfer processes of active boxes.

Theorem 4.10. If there are no repeatable mute and re-activation behaviors in GridHub, the process of transferring active boxes cannot be prevented indefinitely and there is no livelock.

Proof. Proof by contradiction. In GH, suppose there are no repeatable mute and re-activation behaviors and at least one active box's transferring process is prevented indefinitely. Let c_i^a be the conveyor module that holds this box. Because there is no deadlock, c_i^a can make successful attempts. The way to indefinitely prevent c_i^a's box transfer process is to repeat the following steps sequentially:

1. At t, c_i^a makes a potential path P_i.
2. At $t' > t$, c_i^a cannot make a higher level P_i.

The reason c_i^a cannot make a higher level P_i is that there exists at least one potential path, such as P_j ($l_{P_j} > l_{P_i}$) that is made by another active conveyor c_j^a at t'.

At t, c_j^a cannot be at the same location or its state cannot be active; otherwise, P_j is made at t instead of t'. The causes of c_j^a which make P_j at t' are:

- When c_j^a's box is moved from the other conveyor modules before t'. In this case, c_i^a's box is moved to its target, and it cannot permanently prevent c_i^a to make P_i.
- c_j^a re-activates due to a mute behavior at $t_0 < t$, and this mute and re-activation is repeatable which prevents the transferring process of c_i^a's box permanently. This contradicts the assumption that there are no mute and re-activation behaviors.

□

4.5.2 Scenarios of the mute and re-activation behaviors

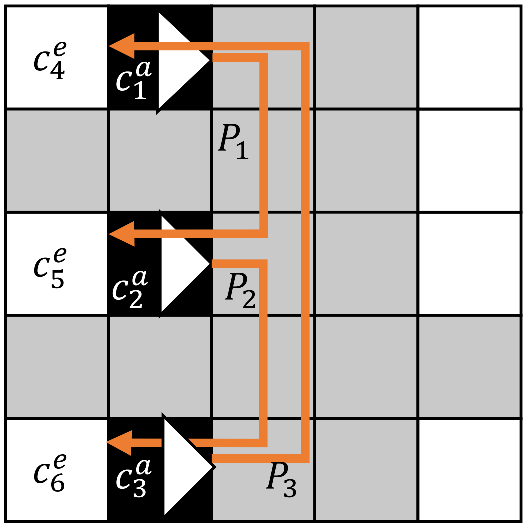
We summarize the possible scenarios of mute and re-activation behaviors before further discussing livelock. Additionally, when a conveyor module is muted in one scenario, it may be re-activated in different scenarios, so the scenarios of mute and re-activation behaviors are summarized respectively.

A mute scenario is β_i ($i \in \mathbb{N}$). Let c_i^a be the active conveyor module to be muted (included conveyor module), and c_j^a be the conveyor module (make conveyor module) that initiates the attempt which mutes c_i^a.

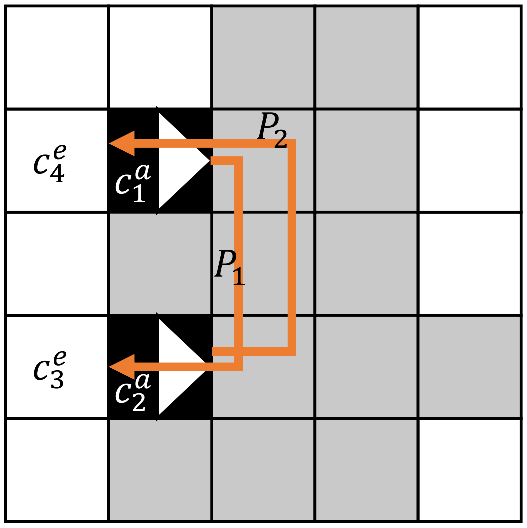
- β_1: c_i^a is directly muted by c_j^a (see Section 4.3.1), but its box is not moved in the current iteration.
- β_2: c_i^a is directly muted, and its box is moved along a potential path P_j that is generated by a $N_1 (N_2)$. The $N_1 (N_2)$ is initiated by c_j^a. In this scenario, the active direction of c_i^a is opposite of the $ad_{c_j^a}$.

- β_3: c_i^a is indirectly muted or overwritten by a potential path. The potential path that makes the mute or overwriting behavior is P_j, and it is generated by a $N_1(N_2)$. The $N_1(N_2)$ is initiated by c_j^a. c_i^a's active direction is perpendicular to $ad_{c_j^a}$, and its active direction is opposite to $d_{N_1(N_2)}$.
- β_4: c_i^a is muted or overwritten by a potential path. The potential path that makes the mute or overwriting behavior is P_j, and it is generated by a $N_1(N_3)$. The $N_1(N_3)$ is initiated by c_j^a. c_i^a's active direction can be either perpendicular or opposite to $ad_{c_j^a}$, and its active direction can be either opposite or perpendicular to $d_{N_1(N_3)}$.
- β_5: c_i^a is indirectly muted or overwritten by a potential path. The potential path that makes the mute or overwriting behavior is P_j, and it is generated by a $N_1(N_4)$. $N_1(N_4)$ is initiated by c_j^a. c_i^a's active direction is perpendicular to $ad_{c_j^a}$, and its active direction is opposite to $d_{N_1(N_4)}$.
- β_6, β_7, and β_8: c_i^a is directly muted by c_j^a, and its box is moved along P_j that is generated by a $N_1(N_2)$, $N_1(N_3)$, and $N_1(N_4)$ respectively. These attempts are initiated by an active conveyor other than c_j^a respectively.

Furthermore, because the negotiations proceed simultaneously, a muted conveyor module can also mute other conveyor modules (Figure 91b and 91a).



(a) c_1^a and c_2^a *mute each other.*



(b) c_1^a mutes c_2^a, c_2^a mutes c_3^a, and c_3^a mutes c_1^a.

Figure 91: Connected mute scenarios (mute each other, and serial of mute).

There are two re-activation scenarios, and they are expressed as γ_j $(i \in \mathbb{N})$.

- γ_1: re-activation when the $ni_{dr} > rp_{dr}$. In this scenario, the muted box stops at a conveyor module before it is re-activated.
- γ_2: re-activation of conveyor modules when the $ni_{dr} \leq rp_{dr}$. This scenario can occur when a conveyor module is cross muted, when the movements are toward their target or after the boxes are muted and moved. Other attempts in the future iteration move the box toward their targets.

4.5.3 Absence of livelock in GridHub

To check whether GridHub is livelock free, we categorize GridHub by the upper limitation of the number of active conveyor modules ($M_{\max}^a$). We then check all possible mute and re-activation scenarios to find whether GridHub is livelock free.

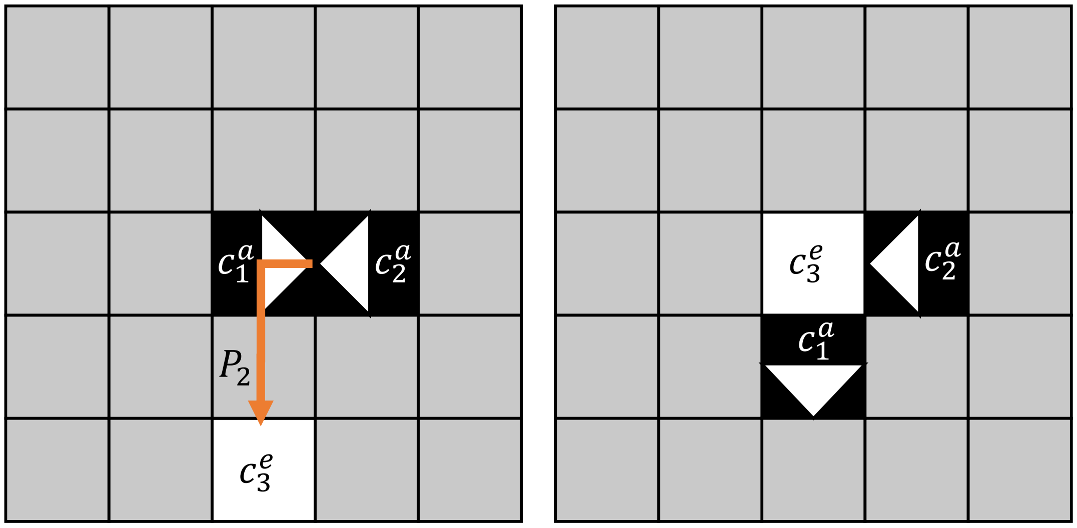
4.5.3.1 *GH* with $M_{\max}^a = 2$ and $M_{\min}^e \geq 2$

Theorem 4.11. If a *GH* has: $M_{\max}^a = 2$, $M_{\min}^e \geq 2$, and $rp_{dr} = 4$ or $rp_{dr} = 6$, it is livelock-free.

Proof. Let the two possible active conveyor modules in GH be c_1^a and c_2^a. When there are no mute or re-activation behaviors, there is certainly no repeatable mute and re-activation behaviors, and the GH is livelock free. When c_1^a is the included conveyor module of β_1, it must be moved. This scenario cannot exist in GH with $M^a_{\max} = 2$ and $M^e_{\min} \geq 2$. When c_1^a is the included conveyor module of β_6, β_7 or β_8 respectively, because the third active module is needed, this scenario does not exist when GH has $M^a_{\max} = 2$ and $M^e_{\min} \geq 2$. When c_1^a and c_2^a mute each other (see Figure 91b), one can re-activate earlier because rp_{dr} is different for different conveyor modules at different times. Consequently, the location of c_2^a or c_1^a can change, and the mute scenario cannot repeat.

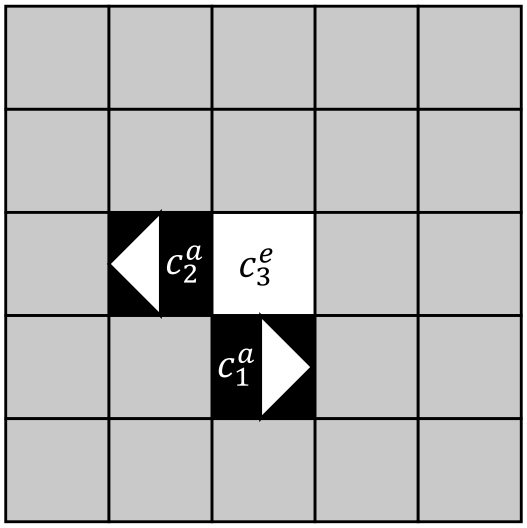
When c_1^a is the included conveyor module of β_2, c_2^a's box changes location after c_1^a is directly muted and moved at $t = 1$. Let $M^e_{\min} = 2$:

1. If c_2^a's box moves slowest, the new location of its box is displayed in Figure 93a or Figure 93b, when c_1^a is re-activated in γ_1.
2. c_2^a's location is displayed in Figure 92b, when c_1^a is re-activated in γ_2

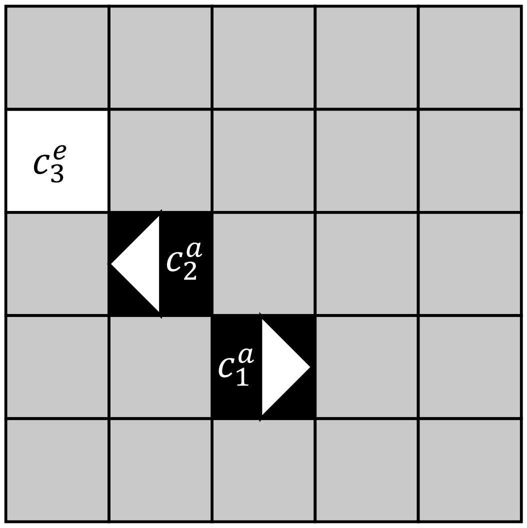


(a) $t = 1$. (b) $t = 2$.

Figure 92: β_2, γ_1 or γ_2 (part 1).



(a) $t = 6$.



(b) $t = 8$.

Figure 93: β_2, γ_1 or γ_2 (part 2).

When c_1^a is the included conveyor module of β_3, c_2^a's box changes location after c_1^a is directly muted and moved at $t = 1$. Let $M^e_{\text{min}} = 2$:

1. If c_2^a's box moves slowest, the new locations of its box is displayed in Figure 95a or Figure 95b, when c_1^a is re-activated in γ_1.
2. Because there are only two active modules, and $d_{N_1(N_2)} = ad_{c_1^a}(\|)$, γ_2 is impossible.

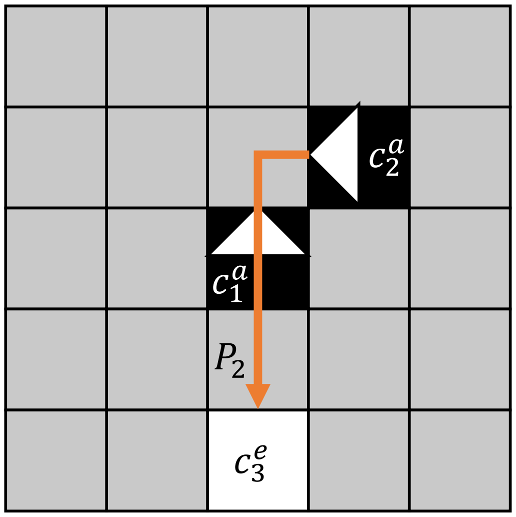
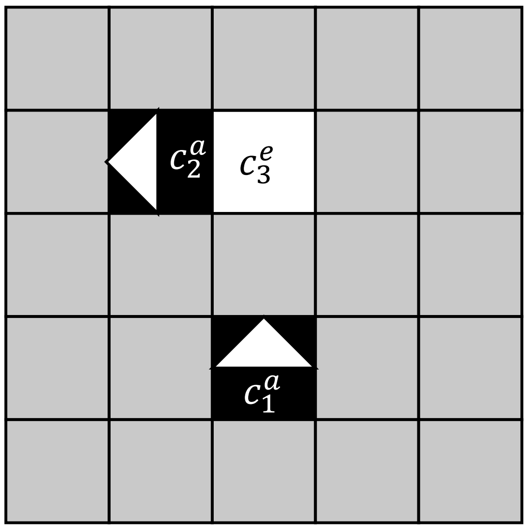
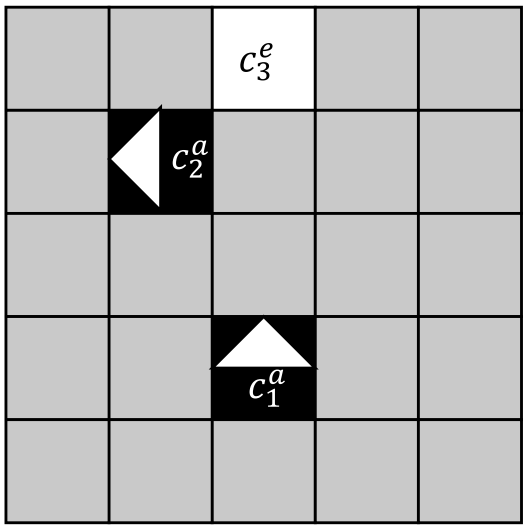


Figure 94: β_3 and γ_1 (part 1). When $t = 1$.



(a) $t = 6$.

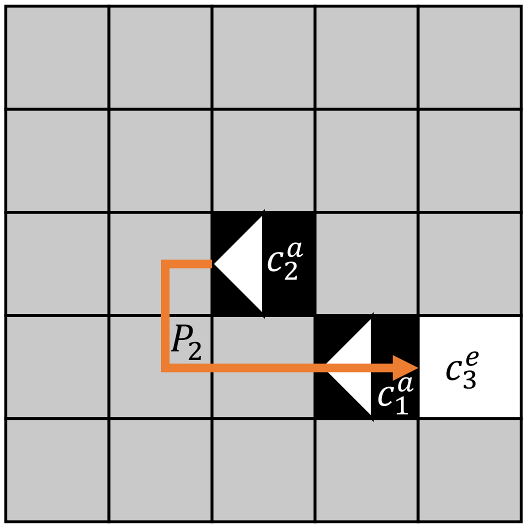


(b) $t = 8$.

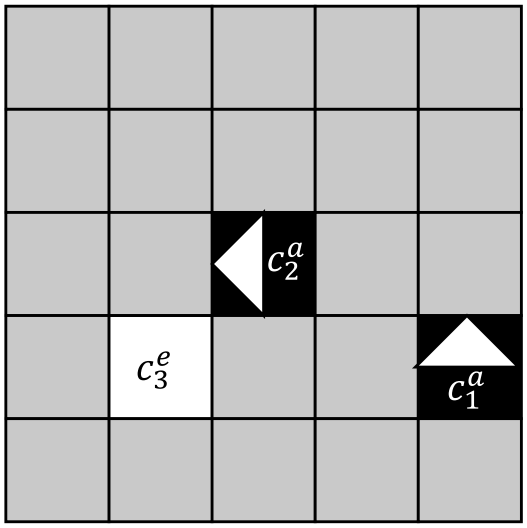
Figure 95: β_3 and γ_1 (part 2).

When c_1^a is the included conveyor module of β_4, except when these two modules mute each other, c_2^a's box changes location after c_1^a is muted and moved at $t = 1$ Let $M^e_{\min} = 2$:

1. If c_2^a's box moves slowest, the new location of its box is displayed in Figure 97a or Figure 97b, when c_1^a is re-activated in γ_1.
2. c_2^a's location is displayed in Figure 96b, when c_1^a is re-activated in γ_2

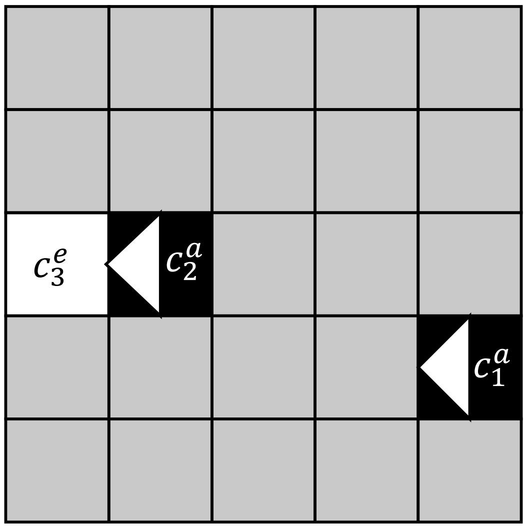


(a) $t = 1$.

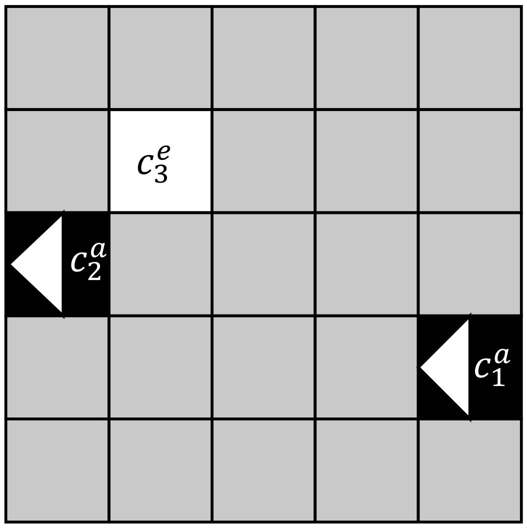


(b) $t = 2$.

Figure 96: β_4, γ_1 or γ_2 (part 1).



(a) $t = 6$.



(b) $t = 8$.

Figure 97: β_4, γ_1 or γ_2 (part 2).

When c_1^a is the included conveyor module of β_5, c_2^a's box changes location after c_1^a is indirectly muted and moved at $t = 1$. Let $M_{\min}^e = 2$:

1. If c_2^a's box moves slowest, the new location of its box is displayed in Figure 99a or Figure 99b, when c_1^a is re-activated in γ_1.

2. Because there are only two active modules, and $d_{N_1(N_2)} = ad_{c_1^a}(\|)$, γ_2 is impossible.

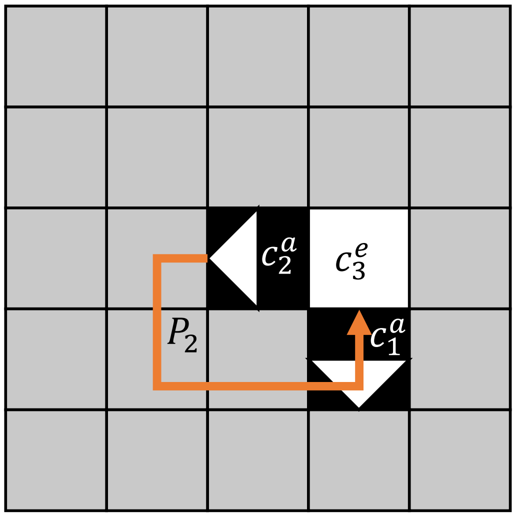
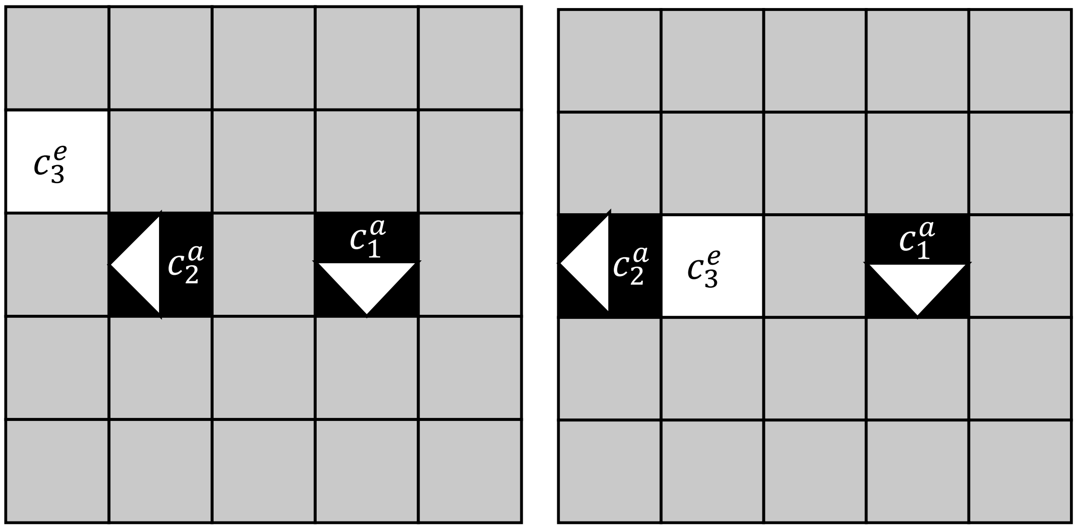


Figure 98: β_5 and γ_1 (part 1). $t = 1$.



(a) $t = 6$. (b) $t = 8$.

Figure 99: β_5 and γ_1 (part 2).

For both of the re-activation scenarios related to β_2, β_3, β_5 and the other possibilities of β_4, the second empty conveyor module can be in any other location in GH. Both c_1^a and c_2^a can make new potential paths and make the mute of c_1^a not repeatable.

Let $M^e_{\min} > 2$ in the above scenarios. c^a_2's box can be moved to a farther location than those mentioned above. Additionally, the active conveyor modules have more than one chance to make potential paths, meaning repeatable mute behaviors cannot occur.

After checking the scenarios of mute and re-activation, we conclude that there is no repeatable mute and re-activation. □

From Theorem 4.11, we can show a corollary between any mute and re-activation scenarios.

Theorem 4.12. For any scenarios of mute and re-activation, when all of the following conditions are true, the same mute and re-activation behaviors cannot be repeated.

1. The conveyor module that mutes other conveyor modules changes its location.
2. The conveyor module also makes progress in the transfer process.
3. There are enough empty conveyor modules for every active conveyor module to make at least one alternative potential path in every iteration.

Proof. Proof by checking all of the mute and re-activation scenarios.

In scenario β_2, β_3, β_4, or β_5, the impossibility of repeatable mute and re-activation behaviors were shown when proving Theorem 4.11.

In scenario β_1, β_6, β_7, and β_8, in order to cause repeatable mute and re-activation behaviors, another active conveyor module is needed. There are three possibilities:

1. The third active conveyor does not initiate attempts to move the muted box; the result is the same as β_2, β_3, β_4, and β_5's.
2. The third active conveyor initiates an attempt to move the muted box, but in either γ_1 or γ_2, the mute and re-activation is not repeatable.
3. The third active conveyor initiates an attempt to move the muted box to a location, and the mute behaviors can occur again in the new location. Because the active conveyor changes its location, and it progresses in its transfer process, even when the conveyor module holds the same box is muted again,

the active conveyor can still move to its target. After its box arrives at its target, the mute and re-activation does repeat.

□

4.5.3.2 *GH* with $M^a_{\max} = 3$ and $M^e_{\min} \geq 3$

Theorem 4.13. *GH is livelock free when it meets these conditions: $M^a_{\max} = 3$, $M^e_{\min} \geq 3$, $rp_{dr} = 4$ or $rp_{dr} = 6$.*

Proof. There are three active conveyor modules in *GH*. Let the active conveyor modules in *GH* be c^a_1, c^a_2, and c^a_3.

When there are no mute and re-activation behaviors, there are certainly no repeatable mute and re-activation behaviors. Then *GH* is livelock free.

When only two active modules are included in any mute and re-activation behaviors, the problem is reduced to *GH* with $M^a_{\max} = 2$. Apply Theorem 4.11, then *GH* is livelock free.

When all of the three active modules are included in three mute scenarios, these three active conveyor modules must mute in a series (see Figure 91a as example). Since the value of rp_{dr} can be different for different modules, when one of them re-activates earlier, and the number of empty conveyors is enough, the location of this active module can be changed based on the negotiation rule. In this case, apply Theorem 4.12 to show that *GH* is livelock free.

When all three active modules are included in two mute scenarios, two possibilities exist:

1. There is a series of mutes: c^a_1 mutes c^a_2, c^a_2 mutes c^a_3, but c^a_1 is not muted. Due to there being enough empty conveyor modules, c^a_1 can change its box locations. Also, since the value of rp_{dr} can be different for different modules, when one of them re-activates earlier, the location of this module can be changed because there are enough empty modules.
2. The other conveyor module mutes two conveyor modules. For example, c^a_1 mutes c^a_2 and c^a_3. c^a_1 can proceed with the transferring processes and change the location of its active box.

In the above possibilities, apply Theorem 4.12 to show that GH is livelock free. $\square$

4.5.3.3 GH with $M^a_{\max} = 4$ and $M^e_{\min} \geq 4$

Theorem 4.14. *GH is livelock free when it meets these conditions: $M^a_{\max} = 4$, $M^e_{\min} \geq 4$, $rp_{dr} = 4$ or $rp_{dr} = 6$.*

Proof. There are four active conveyor modules in GH. Let the active conveyor modules in GH be c^a_1, c^a_2, c^a_3, and c^a_4. When there are no mute and re-activation behaviors, there are certainly no repeatable mute and re-activation behaviors, and the GH is livelock free. When only two active conveyor modules are involved in any mute and re-activation behaviors, the problem is reduced to GH with $M^a_{\max} = 2$, in which case the GH is livelock free (Theorem 4.11). When there are only three active conveyor modules included in any mute and re-activation behaviors, the problem is reduced to GH with $M^a_{\max} = 3$. Apply Theorem 4.13, then GH is livelock free.

When all four modules are involved in two mute scenarios, two possibilities exist:

1. One pair of active conveyor modules mute each other. Since the value of rp_{dr} can be different for different modules, when one of them re-activates early, the location of this module can change.
2. Two modules are muted by the other two respectively. Since the other two can still proceed with their transferring processes, the locations of the two active boxes can be changed.

In both possibilities, apply Theorem 4.12 to show GH is livelock free.

When all four modules are involved in three mute scenarios, the following possibilities exist:

1. If two active conveyor modules mute each other, the other active module is muted by the fourth active module. Since the value of rp_{dr} can be different for different modules, when one of them re-activates early, the location of this one can be changed, while the other active conveyor module can process its transferring process and change its location.

2. If a series of mute scenarios make three conveyor modules become muted, the other conveyor module is not muted. For example, c_1^a mutes c_2^a, c_2^a mutes c_3^a, c_3^a mutes c_4^a, but c_1^a keeps active. In the above case, c_1^a can move its box so the box location changes.
3. If three conveyor modules are muted together by the other conveyor module, then the remaining active conveyor module can proceed with its transferring process and change the location of its active box.

In the above possibilities, apply Theorem 4.12 to show *GH* is livelock free.

When all four modules are included in four mute scenarios, two possibilities exist:

1. If two pairs of active conveyor modules mute each other in scenario β_3, for each pair of muted modules, when one of them re-activates early, the location of its active box changes.
2. If a series of mutes exist, for example, c_1^a mutes c_2^a, c_2^a mutes c_3^a, c_3^a mutes c_4^a, and c_4^a mutes c_1^a, when active module re-activates early, the location of this module can be changed.

In the above possibilities, apply Theorem 4.12 to show that *GH* is livelock free. □

4.5.3.4 *GH* with $M^a_{\max} > 4$ and $M^e_{\min} > 4$

In this case, it is hard to infer whether or not the locations of active boxes can be changed. Hence, *GH* may not meet the conditions of Theorem 4.12. In other words, an active module which mutes other modules, may not able to move their boxes due to waiting on each other.

4.5.4 Method to reduce livelock risks in GridHub

In other grid-based systems, livelock is either proven to be impossible or are solved by another method. For example, Seibold (2015) stated that the boxes' routes in the GridSorter were never circular, so livelock could not occur. Schwab (2015) implemented a detect-solve procedure to detect the livelock process first, solving the livelock based on the priorities of AGVs.

In GridHubs, which we have proven to be livelock free because of mute and re-activation behaviors, the risk of livelock always exists. The number of active conveyor modules may have to be reduced in order to remove or reduce the risk of livelock in GridHub. There are two approaches:

1. Limit the task assignment or the departure information assignment directly.
2. Increase the values of rp_{dr}, in order to make the muted conveyor "silent" for a longer period and reduce the number of active modules. The effect of increasing the length of time is shown in Section 5. From the measured data, this method is ineffective in reducing the number of active conveyor modules.

CHAPTER 5

SYSTEM PERFORMANCE OF UNIT-SIZED GRIDHUB

5.1 Simulation modeling

We run simulations to test GridHub's control algorithms and measure system performance.

5.1.1 Simulation model building in AnyLogic

The AnyLogic is a simulation platform that provides discrete-event and agent-based methods. In GridHub, every conveyor module is an agent that synchronizes at each step when executing control algorithms. Additionally, AnyLogic supports customized programming in Java, which enables the user to test complex logic. Hence, AnyLogic is an effective tool to test GridHub's control algorithms and measure system performance.

In AnyLogic, the "main" agent is unique and created by the platform automatically. The main agent contains all code related to the simulation setting activities. We create a box agent which contains only methods of generating animations and recording data, and a conveyor agent to represent the conveyor module and its the control algorithms. Populations of the box and conveyor agents are added into the main agent. The simulation model then runs the main agent's code.

5.1.1.1 Gates and simulation running sequence

We specify one or a set of gates for a working or newly arrived box, entering or leaving the system and call these gates *input gate(s)* or *output gate(s).* A box enters the system through an available input gate(s). A working box exits the system through an output gate(s), which we call target.

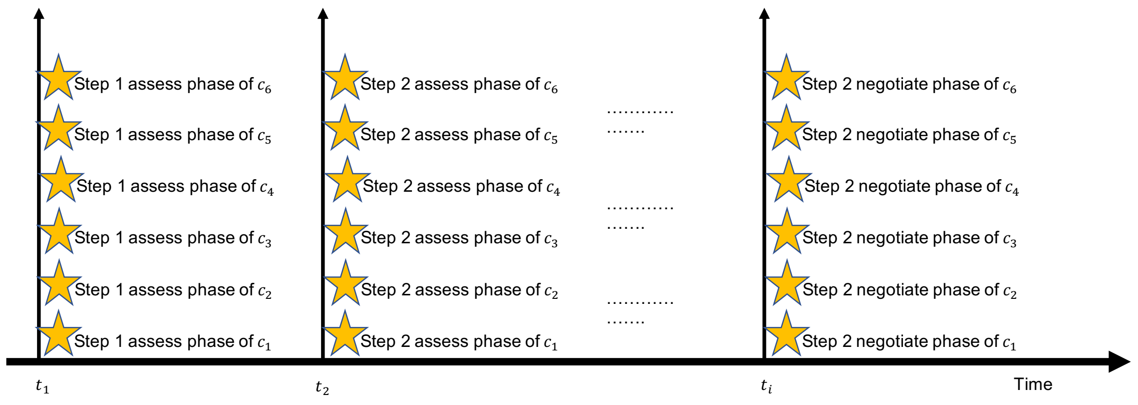
A box must be non-working before it is assigned departure information. In each simulation, every non-working box is randomly selected to assign departure information. Furthermore, working boxes that have the same departure information can be divided into one or more *cluster(s)*. For example, suppose there are *10* boxes assigned to gate *2* on the left edge of a GridHub. We can divide *10* boxes into one cluster; we can also divide them into two clusters (*5* boxes each). We describe more about box clustering later.

We implement the "CONWIP" principle in the GridHubs used for algorithm and performance testing. Similar principles are also used in the GridStore (Gue et al., 2014). In these GridHubs, working boxes remain constant. The detailed process is:

1. When a working box reaches its target and leaves the system at t, another box enters the system in iteration $t+1$.
2. Simultaneously, the GridHub receives an external request and translates it to departure information matching the exiting box's at t. Departure information depends on operational modes explained in Section 5.2.

5.1.1.2 Methods of executing GridHub's algorithms in AnyLogic

In AnyLogic, the "event" utility performs actions occurring at fixed intervals in the simulation (Figure 100). Events then trigger each step of the control algorithm, and we make these events repeat in fixed intervals.



Step 1 assess phase of c_6
Step 1 assess phase of c_5
Step 1 assess phase of c_4
Step 1 assess phase of c_3
Step 1 assess phase of c_2
Step 1 assess phase of c_1
Step 2 assess phase of c_6
Step 2 assess phase of c_5
Step 2 assess phase of c_4
Step 2 assess phase of c_3
Step 2 assess phase of c_2
Step 2 assess phase of c_1
Step 2 negotiate phase of c_6
Step 2 negotiate phase of c_5
Step 2 negotiate phase of c_4
Step 2 negotiate phase of c_3
Step 2 negotiate phase of c_2
Step 2 negotiate phase of c_1
t_1
t_2
t_i
Time

Figure 100: Events and time to occur in AnyLogic. An event triggers each conveyor module to execute actions.

We also use the “dynamic event” to simulate message passing and processing activities. A conveyor agent schedules a dynamic event with its neighbor in order to send and process the message (Figure 101).

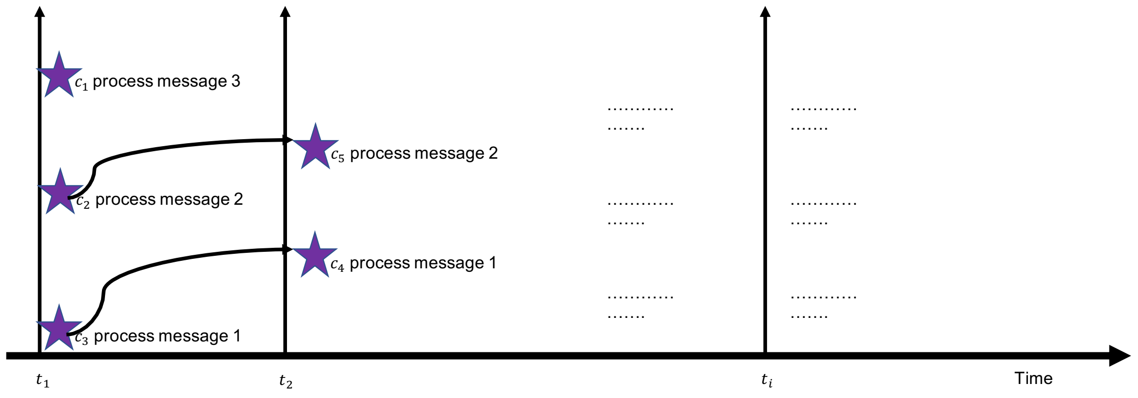


Figure 101: Dynamic events, scheduling relationship and occur time in AnyLogic. c_3 processes message 1 at t_1. Suppose the decision of c_3 is to pass the message to c_4. c_3 schedules a dynamic event for c_4 at t_2. Then, c_4 executes the actions defined by that dynamic event at t_2. The difference of t_1 and t_2 equals the time of passing and buffering messages.

We have stated that there may be many methods for deciding the higher priority directions in every iteration. In AnyLogic, the main agent makes this decision through the following process: First, the main agent stores and shuffles all possible combinations of the higher priority directions in a Java Collection (ArrayList). Then, the main agent chooses the first element of the collection and broadcasts this selection to all conveyor modules.

5.1.2 Determine warm-up period and replications

We determine the warm-up periods using Welch’s method (Mahajan and Ingalls, 2004). Replications of simulations are required to insure accurate output (system throughput). To determine the number of replications needed, we use M.Law (2015)’s equation,

$$\frac{t_{i-1,\ 1-\alpha/2} * \sqrt{\frac{S^2(n)}{i}}}{|X|} < \gamma'$$

.

The number of replications is n, the average throughput is X; the standard deviation is S; and $t_{i-1,\ 1-\alpha/2}$ is the critical t-value when the degree of freedom is $i-1$. If the confidence level is p, then $\gamma' = \frac{1-p}{1+1-p}$. In this case, we use 95% confidence level, and $\gamma' = \frac{1-0.05}{1+1-0.05} = 0.0476$.

5.2 Factors affect the system performance

When designing or installing a GridHub, some questions need to be answered, such the aspect ratio. These questions are factors that affect GridHub's performance. We list these factors first then run a series of simulations to test the system performance.

Operational modes The following operational modes are described according to the material handling tasks described in Chapter 1. Every cluster of boxes can be set to have different operational modes. However, we only set the GridHub in one of the below modes.

m1: In this mode, GridHub performs retrieving tasks. External requests have to specify the edge from which to retrieve the boxes. Hence, the departure information of a working box includes which edge the box will leave. Output gates can be any gate located at the departure edge. Input gates are all gates at the four edges meaning that a newly arrived box can enter the system at any gate on any edge.

m2: In this mode, GridHub performs sorting tasks. External requests have to specify a gate for a working box to leave. All gates on the four edges are input gates.

m3: In this mode, GridHub performs sequencing tasks. Compared to m1, the box departure sequence is given to the boxes in the same box cluster. Gate assignments are the same as m1.

m4: In this mode, GridHub performs sorting and sequencing tasks. Compared to m2, the box departure sequence is given to the boxes in the same box cluster. Gate assignments are the same as m2.

Number of working boxes The number of working boxes in GridHub depends on the number of working boxes in a cluster. We assume the numbers of working boxes in every cluster equals (x_{wk}). Hence, if the number of clusters is fixed, we can use x_{wk} to express the number of working boxes.

Aspect ratio This factor indicates the shape of GridHub. We use the following equation to calculate the aspect ratio:

$$x_{asp} = \frac{number\ of\ columns\ (edges\ are\ excluded)}{number\ of\ rows\ (edges\ are\ excluded)}$$

.

Options for choosing expect paths In Chapter 3, we find two expect paths for each working box moving to its target. Based on expect paths, any of the following four methods can guide a working conveyor module to assign a transfer task (Figure 102 shows examples):

- op1: A box moves to the same column or row as its departure gate, then moves to its target.
- op2: A box moves to a location close to its departure gate, and then it moves to face its departure gate to exit.
- op3: A box moves left or right, and then it moves up or down.
- op4: A box moves up or down, and then it moves left or right.

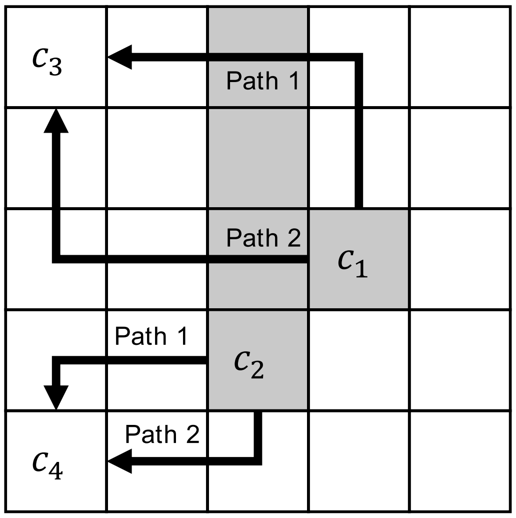


Figure 102: Example of expect paths selection. *c1* and *c2* choose op1 or op2, while they are trying to assign transfer tasks along Path 1 or Path 2. If unable to assign tasks along the initially chosen path (1 or 2), they choose the other. Using op3, all boxes try to move left first, then up or down. Using op4, all boxes try to move up or down first, then left.

Forward attempt In Chapter 3, to initiate *N2*, *N3*, and *N4*, we describe rules of "forward attempt" for conveyor modules in the "empty with movement confirmation" state. Whether the forward attempt is disabled or enabled, system throughput is measured.

Period between mute and re-activation In Chapter 4, we discuss mute and re-activation behaviors. We use the variable rp_{dr} to indicate the time between the mute and re-activation of an active conveyor module. Increasing rp_{dr} may affect the number of active boxes, and in turn, system throughput.

Number of empty conveyor modules In a GridHub with more empty modules, active modules can move their boxes to their immediate destinations easily; however, the utilization rate is lower. To express the number of empty modules, we use x_{emp}, and then measure whether it increases the transferring speed of active boxes.

5.3 Experiments and results

We conduct the following experiments to investigate the GridHub's performance. We run each experiment setting *50* replications. In every replication, the simulation runs *8000* iterations. The first *800* iterations are the warm-up period.

5.3.1 Operational modes and system performance

5.3.1.1 Settings

In these experiments, GridHub has *100* conveyor modules (excluding the edges) and gates located at each edge (Figure 103).

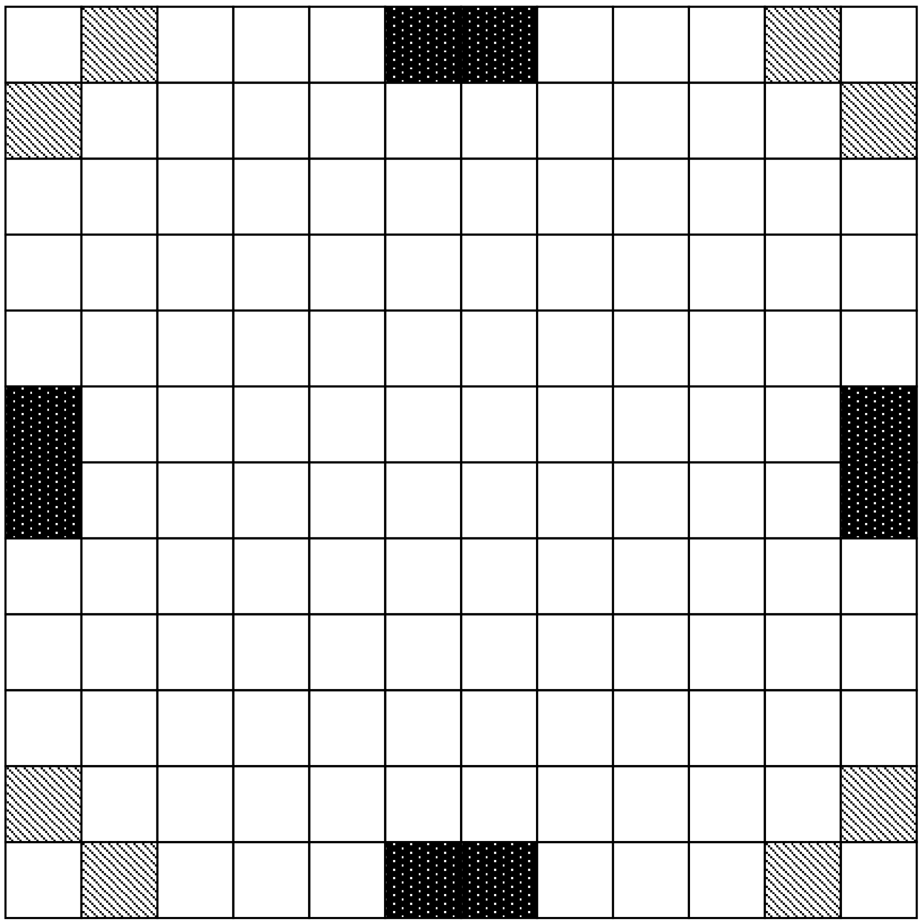


Figure 103: GridHub used for test performance and operational modes, conveyor modules used as gates are shaded.

Additionally, we use op1 to choose expect paths; The forward attempt is enabled; $rp_{dr} = (4, 6)$, $x_{emp} = 20$, and $x_{wk} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. There are two gates at every edge with locations and operational modes shown in Table 9. Every gate is assigned one cluster of boxes. When the departure locations are not specified, we assign every cluster of boxes to one edge.

TABLE 9: Experiment setup and Operational modes.

Mode name	Tasks	Output Gate locations	Specific departure sequence
m1	Retrieving	Any gate on one edge	-
m2	Sortation	Center gates of one edge	-
m2c	Sortation	Corner gates of one edge	-
m3	Sequence	Any gate on one edge	yes
m4	Sequence, sort	Center gates of one edge	yes
m4c	Sequence, sort	Center gates of one edge	yes

5.3.1.2 Discussion

The average system throughput is plotted in Figure 104.

Under m1, working boxes do not have intermediate targets. As the number of working boxes increases, tandem movements of active boxes are more easily formed. Thus, the throughput increases. Under m2, a working box may have intermediate targets, so its expect path is longer than an expect path under m1. Consequently, throughput decreases under m2. m4 is comprised of m2 and m3's activities, so when the number of working boxes increases, throughput decreases.

For effects of the gate locations, if the gates are centrally located under m2, active boxes having perpendicular active directions do not wait for each other to move forward. Under m4, competition for empty modules increases near the gates because when larger sequence boxes arrive earlier, they must make space for the smaller sequence boxes.

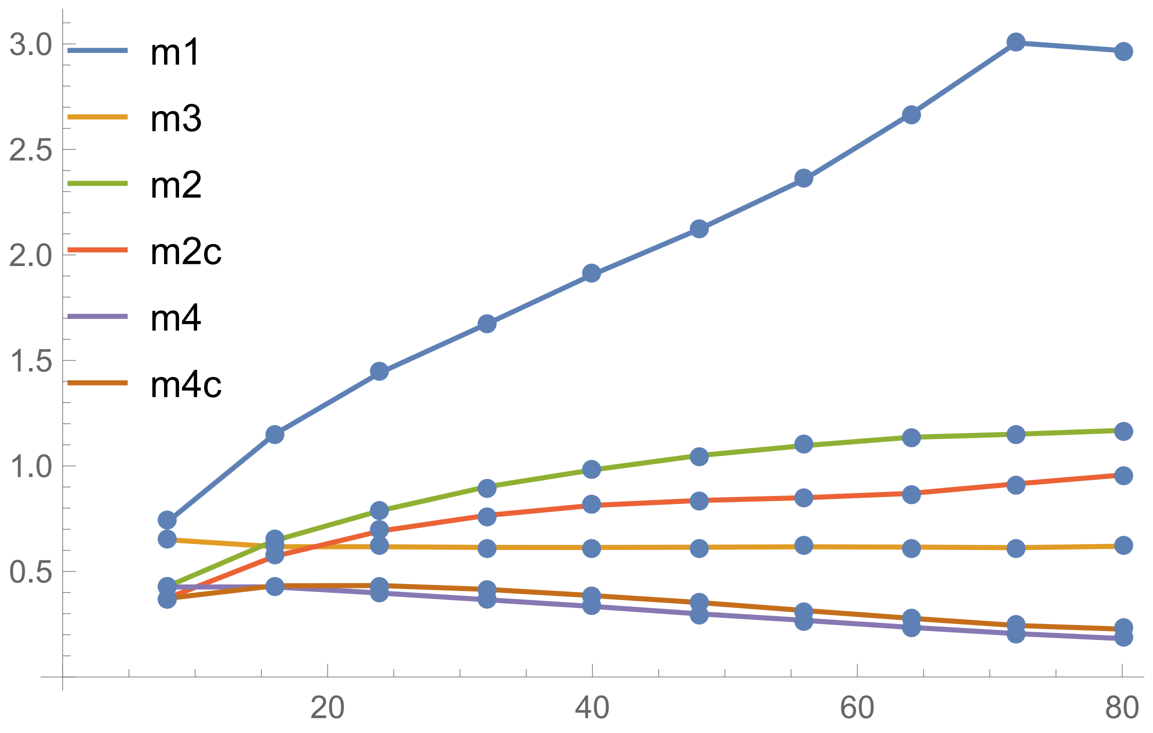


Figure 104: Average system throughput for different operational modes. The y-axis shows the average system throughput counted by "the number of boxes released in every iteration." The x-axis is the number of working boxes (x_{wk} times the number of gates used) in the system.

Because there are multiple working boxes and system storage density is high, it is impossible to move a working box along its expect path without deviation. The length of a working box's expect path is divided by the iterations required to move it out, and we call this the *efficiency* of transferring a working box. The average efficiency of all working boxes in every operational mode is shown in Figure 105.

Since m1 does not require the working boxes to leave at specific locations, a box can exit any location at its departure edge. Thus, the box's expect path length is shorter. As the number of working boxes increases under m1, tandem movements of active boxes have shorter expect paths, resulting in higher efficiency (see Figure 105). In contrast, m3 restricts working boxes leaving the system by blocking transfer task assignment. This means working boxes increase while active boxes cannot. Hence, m3's efficiency is not effected by the number of working boxes (see

Figure 105).

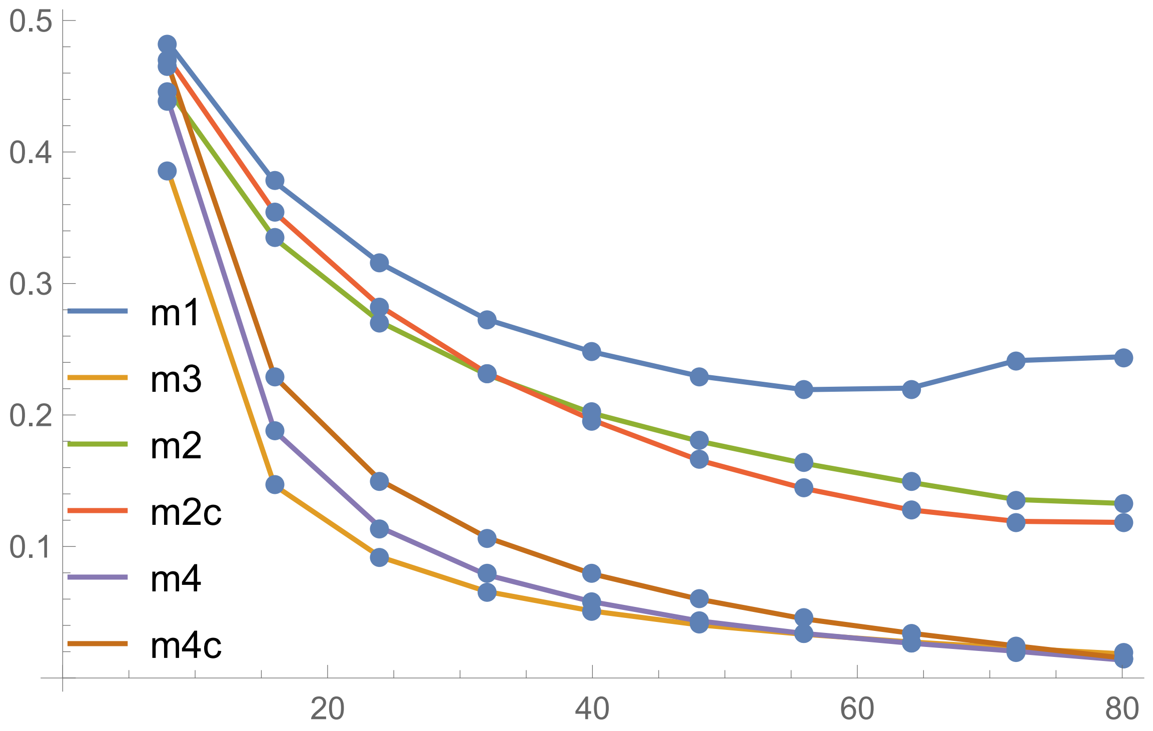


Figure 105: Average box transfer efficiency for different operational modes. The y-axis shows the average box transfer efficiency. The x-axis is the number of working boxes (x_{wk} times the number of gates used) in the system.

5.3.2 Aspect ratios and options for choosing expect paths

5.3.2.1 Settings

All of the shaded gates in Figure 103 are used as output gates for the rest experiments in this chapter. We only choose m2 to run the rest experiments. The reasons are:

- Under m2, box transfer processes are purely dependent on how the negotiations work. Under m3 and m4, transfer task assignments also affect the box transfer process.
- Under m2, there is also a greater chance active conveyor modules will be muted and re-activated, which m1 does not account for.

The experiments are conducted with settings in Table 10, and the forward attempts are enabled.

TABLE 10: Settings of experiment on aspect ratios and options for choosing expect paths.

Setting Name	x_{wk}	x_{asp}	Expect path
E1	1	$\{0.16, 0.25, 1, 4, 6.25\}$	$\{1, 2, 3, 4\}$
E2	2	$\{0.16, 0.25, 1, 4, 6.25\}$	$\{1, 2, 3, 4\}$
E3	3	$\{0.16, 0.25, 1, 4, 6.25\}$	$\{1, 2, 3, 4\}$

5.3.2.2 Discussion

We measure the result by system throughput (see Figure 106).

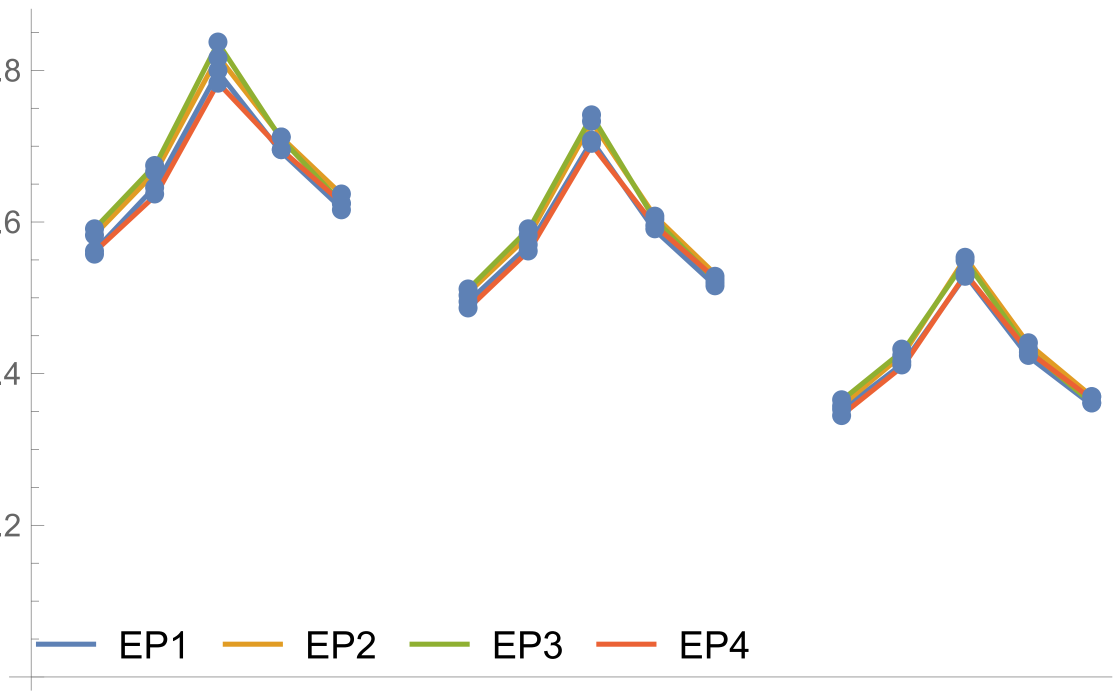


Figure 106: Average system throughput when aspect ratio and expect routes are changed. The y-axis shows the average system throughput counted by "the number of boxes released in every iteration." The group of lines from left to right are for the case that the number of working boxes equals *16*, *32*, and *48* (or x_{wk} changes from *1* to *3*); the dots on each line represent when the aspect ratio is changed from *0.25* to *6.25*. In the legend, "EP" represents the expect path selection from 1 to 4 shows in Figure 102.

We use 3-way ANOVA to check whether aspect ratios, options for choosing expect path, and x_{wk} significantly affect throughput. From Table 11, we conclude all factors and their interactions are significant. We also conduct a TukeyHSD test for detailed comparisons (See Appendix 7.4.2).

According to GridHub's control algorithm, active boxes with left and right active directions proceed before the active boxes with up and down active directions. According to the options for choosing expect path, when the GridHub's aspect ratio is small, modules assign left and right transfer tasks first. Thus, we achieve higher throughput by moving active boxes left and right. When the aspect ratio grows, the effect is opposite. Furthermore, with an aspect ratio close to *1*, the average length of working boxes' expect paths shortens, resulting in increased throughput.

TABLE 11: ANOVA results of the experiments on aspect ratios and options for choosing expect paths (output from R).

Factors	Df	Sum Sq	Mean Sq	F value	$Pr(> F)$
x_{asp}	4	16.57	4.142	6.651e+04	<2e-16
ExpectPath	3	0.25	0.082	1.312e+03	<2e-16
x_{wk}	2	32.32	16.161	2.595e+05	<2e-16
x_{asp}:ExpectPath	12	0.08	0.007	1.106e+02	<2e-16
x_{asp}:x_{wk}	8	0.23	0.029	4.709e+02	<2e-16
x_{asp}:x_{wk}	6	0.02	0.003	5.224e+01	<2e-16
x_{asp}:ExpectPath:x_{wk}	24	0.01	0.000	7.394e+00	<2e-16
Residuals	2940	0.18	0.000		

5.3.3 Number of empty conveyor modules and limitation of task assignments

5.3.3.1 Settings

All of the shaded gates in Figure 103 are used as output gates. Experiment settings are in Table 12. The sets in the table are: $EMP = \{16, 24, 32, 40, 48\}$ and $RP = \{(4,6),(12,14),(20,22),(28,30),(36,38)\}$.

TABLE 12: Settings for the experiment on the number of empty conveyor modules and limitation of task assignments.

Setting name	x_{wk}	Forward attempt	x_{emp}	rp_{dr}
E1	*1*	Yes	$x_{emp} \in EMP$	$rp_{dr} \in RP$
E2	*1*	NO	$x_{emp} \in EMP$	$rp_{dr} \in RP$
E3	*2*	Yes	$x_{emp} \in EMP$	$rp_{dr} \in RP$
E4	*2*	NO	$x_{emp} \in EMP$	$rp_{dr} \in RP$
E5	*3*	Yes	$x_{emp} \in EMP$	$rp_{dr} \in RP$
E6	*3*	NO	$x_{emp} \in EMP$	$rp_{dr} \in RP$

5.3.3.2 Result and discussion

The system performance is displayed in Figure 107. First, we conclude that enabling forward attempts can increase the system throughput. Second, the increased number of empty modules can increase the system throughput. Third, when rp_{dr}'s values increase, the system throughput is also affected: This effect is most obvious when the x_{wk} is higher, and this occurs when more active boxes are muted, and there is less competition for empty modules. Consequently, the system throughput increases slightly. When x_{wk} is low, increasing rp_{dr} has negative effects on the system throughput because working boxes are muted even when they can find empty modules. When x_{emp} is higher, empty modules are wasted. However, changing rp_{dr}'s values does not effect throughput more than changing the number of empty modules.

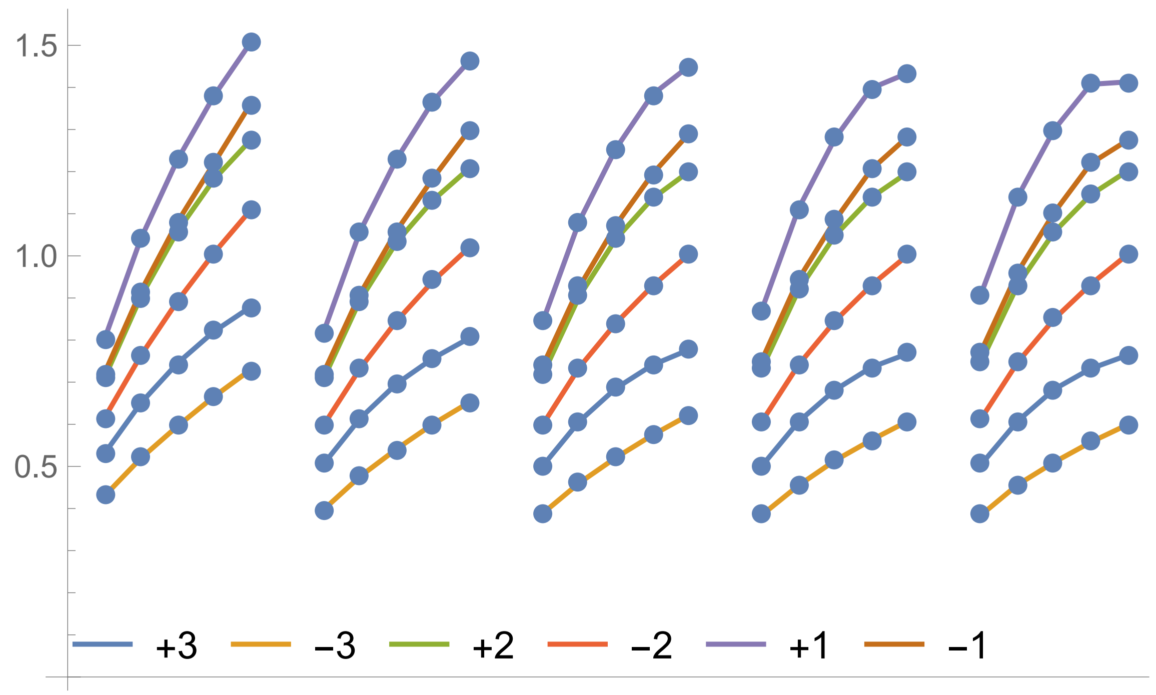


Figure 107: Average system throughput of experiment on the number of empty conveyor modules and limitation of task assignments. The y-axis shows the average system throughput counted by "the number of boxes released in every iteration." The group of lines from left to right are for the case that rp_{dr} changes in the sequence shows in RP; the dots on each line represent when the number of empty modules is changed from *16* to *48*. In the legend, "+" means that forward attempt is enabled; the number indicates the value of x_{wk}.

In Figure 108, the number of active conveyor modules (not counting the edges) is displayed. First, this value is not greatly affected by enabling forward attempts. Second, the increased number of empty conveyors reduces interaction between active conveyors. Thus, when an active conveyor module is muted, the possibility of its re-activation in γ_1 is greater. On the other hand, fewer empty conveyor modules increase the likelihood that the muted active conveyor modules will be re-activated in γ_2. Hence, the number of active conveyor modules is lower when there are fewer empty modules.

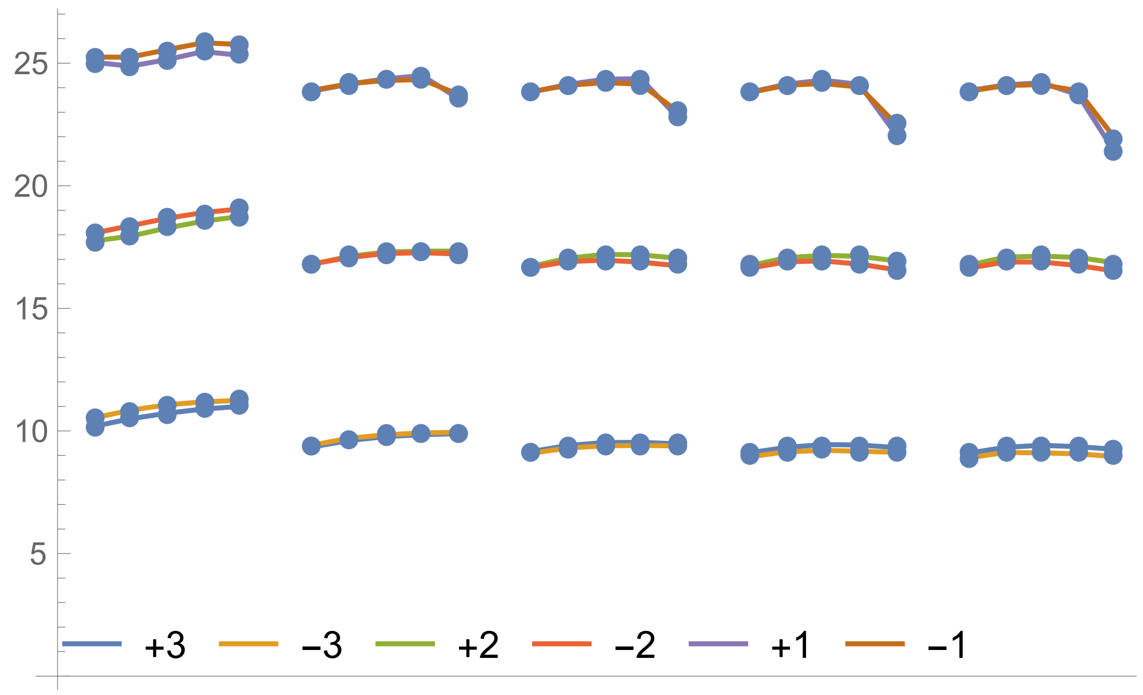


Figure 108: Average number of active conveyor modules per iteration. The y-axis shows the average number of active conveyor modules. The group of lines from left to right are for the case that rp_{dr} changes in the sequence shows in RP; the dots on each line represent when the number of empty modules is changed from *16* to *48*. In the legend, "+" means that forward attempt is enabled; the number indicates the value of x_{wk}.

5.3.4 Negotiation behaviors and transfer processes of active boxes

5.3.4.1 Settings

We use the same settings in the last experiment by fixing rp to *4* or *6* and disabling forward attempt. The only changing variables are $x_{wk} = 1, 2, 3$. So we only consider the negotiation behaviors and the transferring process of active boxes.

5.3.4.2 Results and discussion

All factors that affect GridHub's performance alter the transportation process of every active boxes. When a GridHub is fixed and the active boxes are moved in and out quickly, the throughput is higher. The transportation process of active boxes can be divided into consecutive iterations. Except the iterations where the active module moves the box to its immediate destination, additional iterations are in the following categories:

- The iterations where the active conveyor module receives confirmation of N_2, N_3, and N_4. In this iteration, although the box stops at the conveyor module, the transferring process is progressing.
- The iterations where the active conveyor module fails all four attempts.
- The iterations where the module is muted by the negotiation behaviors which are explained in Section 4.3.

We show statistical results of the negotiation behaviors and the transferring process of active boxes. The data is collected while the simulation is running: First, the number of iterations it takes to move a working box in and out of the system is recorded. Second, when active conveyor modules hold these boxes, the number of negotiation behaviors they make or include are counted and recorded. Additionally, the forward attempt is disabled because the behaviors cannot be recorded accurately when enabled. As we have seen, the value of rp_{dr} does not greatly affect the system performance, so we only use $rp_{dr} = (4, 6)$. The notations of the recorded data are listed in Table 13.

TABLE 13: Notations of negotiation behaviors' records.

Notations	Explanation
y	Number of iterations taken to move the box in and out of the system.
x_1	Number of iterations where all attempts fail.
x_2	Number of iterations where some of N_2, N_4 and N_4 are successful.
x_3	Times of the module when it is directly muted.
x_4	Times of the module when it is indirectly muted.
x_5	Times of the module when it is crossly muted and moved toward the box's target.
x_6	Times of the module when it is crossly muted and moved beyond the box's target.
x_7	Times of the module when it is indirectly pushed.
x_8	Times of the module when its potential path is overwritten opposite to its active direction
x_9	Times of the module when its potential path is overwritten crossly toward the box's target
x_{10}	Times of the module when its potential path is crossly overwritten beyond the box's target
x_{11}	Times of the module when its potential path is overwritten in its active direction
x_{12}	Times of the module when its potential path overlaps other paths in N_1
x_{13}	Times of the module when its potential path overlaps other paths in $N_1([N_2, N_3, N_4])$.
x_{14}	Times of the module when its potential path has fake confirmations

The average numbers of negotiation behaviors on every box are plotted in Figure 109. First, as the number of empty modules increases, these negotiation behaviors occur less often. Second, a higher number of working boxes means a higher number of active conveyor modules. The more active modules the system generates, the more negotiation behaviors. The only exception is the direct mute behaviors which increase slightly when the number of empty modules increases. When GridHub is being emptied, active boxes have a greater chance of facing each other, which is the cause of muting behaviors.

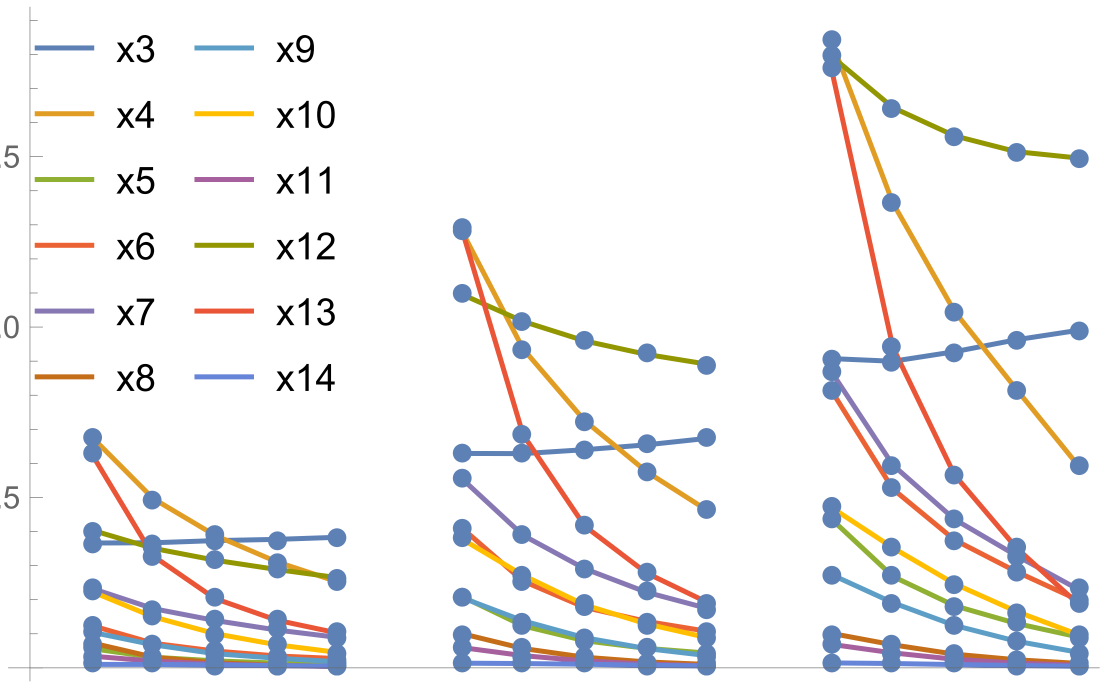


Figure 109: Average occurrence of negotiation behaviors of an individual box. The y-axis show the average occurrence of negotiation behaviors of an individual box. The group of lines from left to right are for the case that x_{wk} changes from *1* to *3*; The dots on each line represent when the number of empty modules is changed from *16* to *48*. The variables in the legend are explained in Table 13.

To verify how the negotiation behaviors affect the transferring process of active boxes, the data of negotiation behaviors is categorized by the box clusters. In other word, we gather the data from boxes which leave at the same gate. Then, we run linear regressions on every cluster's data. In Section 4.3.2, we state that some negotiation behaviors do not delay the transferring processes of active boxes. Their effect cannot be seen in the result of the regression model. Hence, we only include negotiation behaviors that delay the transferring process in the regression models. The raw data of x_4, x_5 and x_8, x_9 are also combined before running the model

($x_{46} = x_4 + x_6$, $x_{810} = x_8 + x_{10}$), because both of them cause the same delay.

$$y = c_0 + c_1x_1 + c_2x_2 + c_3x_3 + c_{46}x_{46} + c_5x_5 + c_{810}x_{810} + c_9x_9 + c_{14}x_{14}$$

A sample of the coefficients and adjusted R-square are displayed in Table 14. Based on the coefficients, all of the muted activities have positive effects on increasing the length of transferring, which means they can delay the transferring of process. Additionally, we have extra *239* regression results based on the experiment settings and departure gates of boxes. We omit these result here. If the reader need to read the result, please contact the author.

TABLE 14: Samples of regression result. The experiment settings are: $x_{wk} = 1$, and $x_{emp} = 16$. The data is collected from the upper left corner gate of the left edge.

Variables	Estimate	Std. Error	t-value	Pr(>|t|)
(Intercept)	4.213831351	0.158807708	26.53417396	1.29E-149
x_1	1.363541065	0.024669446	55.27246287	0
x_2	1.649634454	0.015254744	108.1391138	0
x_3	7.519686015	0.122179489	61.54622228	0
x_{46}	8.795853765	0.080180133	109.7011622	0
x_5	1.478583312	0.269020143	5.496180672	3.98E-08
x_{810}	9.207924756	0.154512154	59.59353041	0
x_9	0.951490659	0.239842568	3.967146735	7.33E-05
x_{14}	2.943257252	0.730400117	4.029650576	5.63E-05
Adjusted R square	0.923302353			

5.4 Conclusion

We have examine several factors affecting the performance of GridHub by measuring the system throughput.

First, the operational modes affect the transfer task assignments, and then the system throughput changes after the transfer task assignments change.

If the operational modes are fixed to m2, the aspect ratios and expect path selections can both affect the system throughput. When $x_{asp} = 1$ results the highest throughput in most of the situations. The guide of expect paths can also affect system performance, but its effect is weaker than the aspect ratio.

If fix all factors above, we found that the empty number of conveyor modules, the limitations of task re-assignment, and the forward attempt can change the system performance. Enabling the forward attempt is good for higher throughput; increasing the number of empty modules results higher throughput, but reduces the space utilization; increasing the limitation of task re-assignment has better effects when the number of working boxes is high.

The transferring processes of active boxes reflects the system throughput. These processes are interfered by the occurrences of negotiation behaviors, and the system settings affect the occurrences of the behaviors.

CHAPTER 6

GRIDHUB FOR NON-UNIT-SIZED BOXES

The flexibility and high throughput of GridHub enables it to handle various applications. For example, GridHub is capable of working as the π-hub of the Physical Internet (Montreuil, 2011; Meller et al., 2013; Ballot et al., 2012). In regular warehouses, GridHub can be used as an automatic storage and retrieval system. However, in many practical applications or conceptual systems, such as warehouses or the Physical Internet, cartons or π-containers (Montreuil, 2011) have different sizes. If we make every conveyor module in GridHub capable of handling one storage item, then this would make the required conveyor module very large (see Figure 110), which waste significant space.

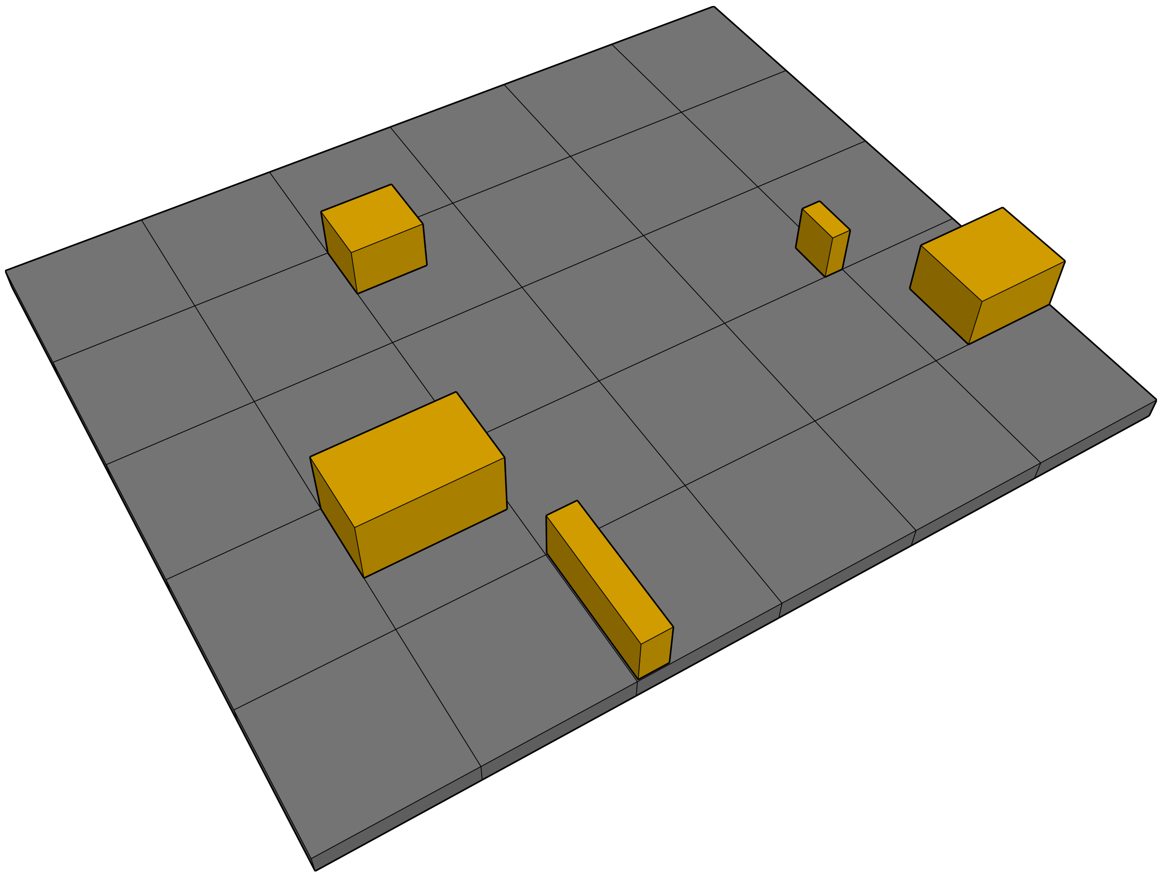


Figure 110: Using one type of big conveyor to fit boxes with different sizes.

We modify the GridHub algorithm to accommodate non-unit-sized boxes. We refer to this new system as *NU GridHub* (GridHub which can only handle unit-sized boxes is called GridHub or *unit-sized GridHub*). In the unit-sized GridHub, one conveyor module can only hold one box, and one box only occupies one conveyor module. In NU GridHub, one box may occupy multiple conveyor modules. Additionally, similar to the unit-sized GridHub, the entire NU GridHub is assumed to be a rectangle, and the boxes in NU GridHub are rectangular (Figure 111). In this chapter, we describe NU GridHub, its control algorithms, and its system performance.

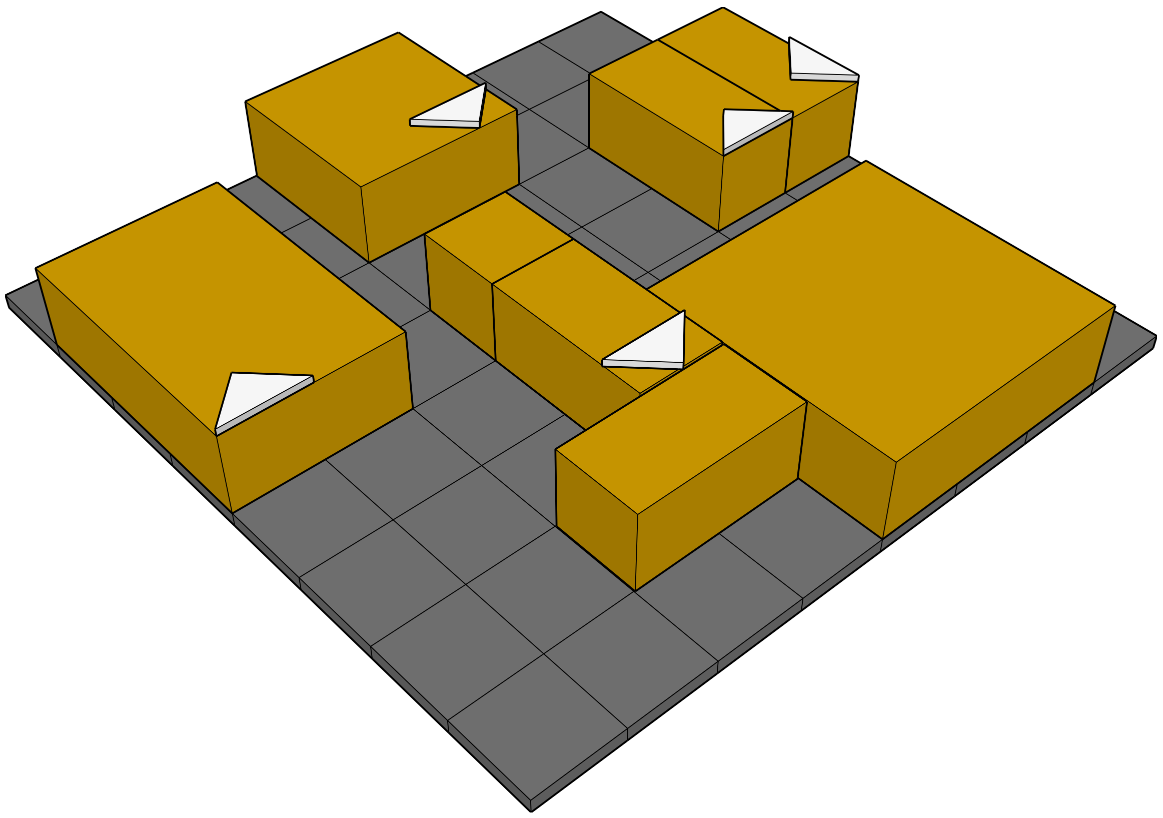


Figure 111: Example of a NU GridHub. The white triangles indicate the boxes' active directions.

6.1 Description of NU GridHub

NU GridHub also consists of identical square conveyors, that can communicate and move boxes to their four neighbors. The system architecture of NU GridHub is the same as the unit-sized GridHub's, but NU GridHub is capable of moving both unit-sized and non-unit-sized boxes.

6.1.1 Box sizes and organization of conveyor modules

Boxes in NU GridHub have different sizes, and they can occupy more than one conveyor. We use b_i to represent an individual box, and the size of a box by two numbers $x \times y$. Variable x is the number of conveyor modules the box occupies counted along horizontally axis; y is the number of conveyor modules along the vertical axis (See examples in Figure 112).

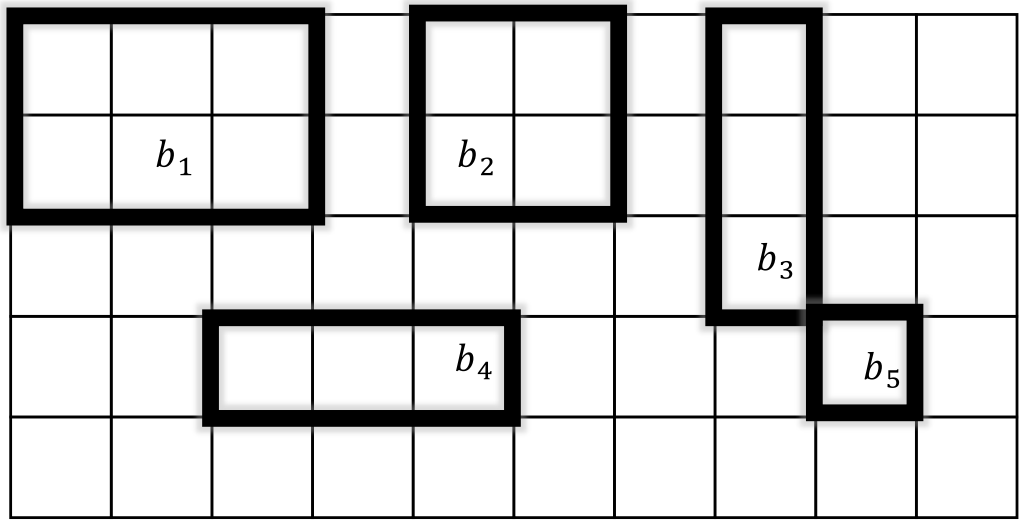


Figure 112: Examples of boxes in NU GridHub. The sizes of b_1 to b_5 are 2×3, 2×2, 1×3, 3×1, and 1×1 respectively.

Conveyor modules occupied by the same box are called *a group* in this chapter. Since a box can be expressed as b_i, we call the group of conveyors under b_i "b_i's group," or use b_i to indicate the group directly. To organize every group of conveyor modules occupied by a single box, relationships among these conveyor modules have to be established based on the conveyors' neighborhoods. This relationship is called the *master-slave relation* of conveyor modules.

If every conveyor module that is in a group is considered a vertex in a graph, we can use two tree structures to describe the relationships of the conveyor module group. These structures are similar to those described by Dominik et al. (2016), but we define two trees, and they have more functions. We use the example in Figure 113 to further illustrate this structure. Some special cases of the tree structure are in Figure 114.

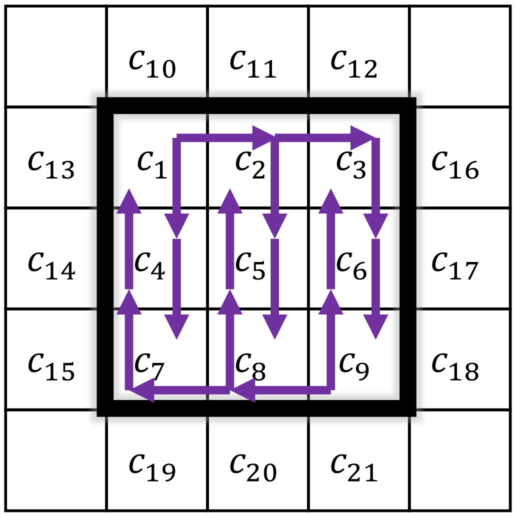


Figure 113: General example of tree structure. The first tree structure is named S_1. S_1's root is the upper-left conveyor module (c_1) covered by the box. We call this conveyor module the *upper-left root master* (MUL_r). A module's slave in this structure is "upper-left horizontal slave" ($SULH$) or "upper-left vertical slave" ($SULV$).

We name the second tree structure S_2. S_2 is rooted at the lower-right conveyor module (c_9). This conveyor module is the *down-right root master* (MDR_r). A module's slave in this structure is "down-right horizontal slave" ($SDRH$) or "down-right vertical slave" ($SDRV$).

The details of the master-slave relationship are in Table 15.

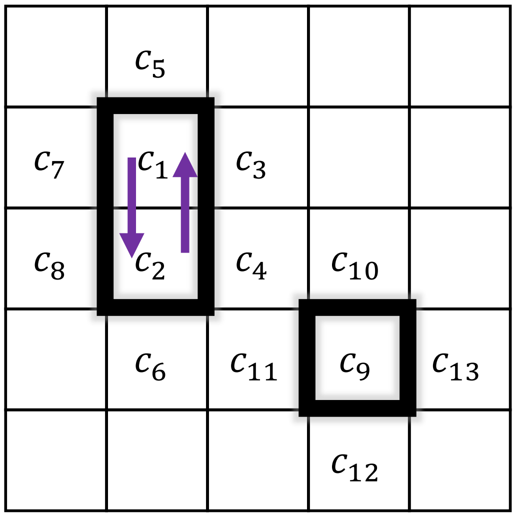


Figure 114: Special examples of tree structure. c_1 is the MUL_r, and c_2 is the MDR_r; two conveyor modules can be the same conveyor module, such as the 1×1 box in Figure 113.

In Figure 113 and 114, the purple arrows represent the details of S_1 and S_2. In S_1, when a purple arrow points away from one conveyor module (for example c_2 in Figure 113) to another conveyor module (for example c_3 in Figure 113), it means that c_2 is c_3's master, and c_3 is c_2's master. More details are listed in Table 15. Additionally, we use purple arrows to indicate message passing activities in this relationship.

TABLE 15: Details of the master-slave relationship using the group in Figure 113 as an example.

Conveyor	*MUL*	*SULH*	*SULV*	*MDR*	*SDRH*	*SDRV*
c_1	itself	c_2	c_4	c_4	no	no
c_2	c_1	c_3	c_5	c_5	no	no
c_3	c_2	no	c_6	c_6	no	no
c_4	c_1	no	c_7	c_7	no	c_1
c_5	c_2	no	c_8	c_8	no	c_2
c_6	c_3	no	c_9	c_9	no	c_3
c_7	c_4	no	no	c_8	no	c_4
c_8	c_5	no	no	c_9	c_7	c_5
c_9	c_6	no	no	itself	c_8	c_6

6.1.1.1 Forward face

The *forward face* is a group of conveyor modules covered by one edge of a box. The size of a forward face is the number of conveyor modules it covers, which is equal to the x or y value. Every group of conveyors has four forward faces, which is up, down, left and right (See example in Figure 115).

MUL_r leads the activities of all conveyor modules in the left and up forward faces; MDR_r leads the activities of all conveyor modules in the down and right forward faces; Hence, we call MUL_r the *corresponding master* of the up and left direction; MDR_r is the corresponding master of the down and right direction. When a negotiation message is passed by a group of conveyors, the corresponding master MUL_r or MDR_r passes the message first. For example in Figure 115, when the group passes a message to the right, c_9 passes to its right neighbor c_{18} first. Every master module also asks its slave module to repeat the same message passing activities: c_9 also ask c_6 to pass a copy of the same message to c_{17}, and c_6 does the same. The message in this case is *duplicated* along one forward face. Hence, this forward face is the *corresponding* forward face, which is in a message passing direction, when we describe passing messages.

The *corresponding side of a forward face* or *corresponding side* is the set of neighbor modules of the forward face, but they are not the members of the conveyor group. When a group of conveyors tries to move a box, all modules of the corresponding side have to be in empty without movement state (see Figure 115).

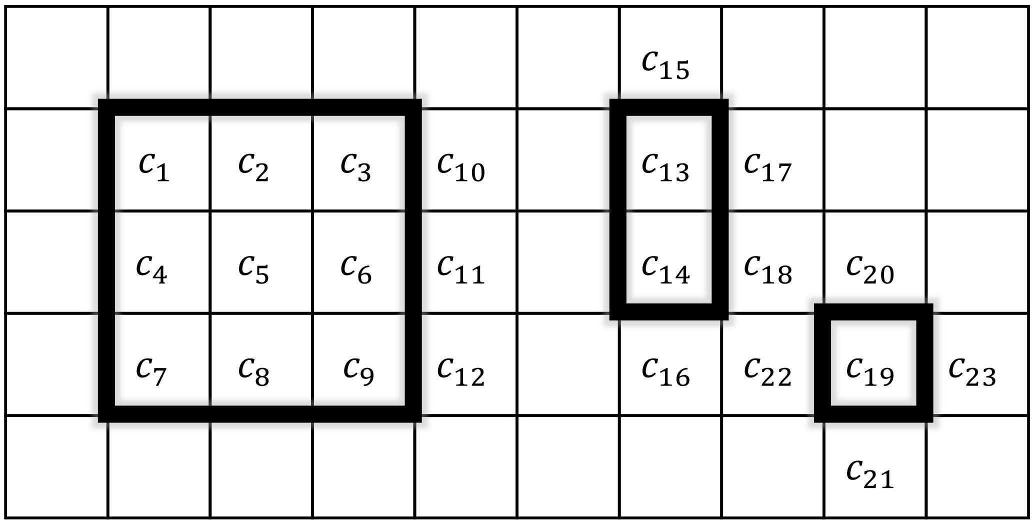


Figure 115: Forward face example. $\{c_3, c_6, c_9\}$ is the right forward face of the left box, and this forward face's size is *3*; the up forward face of the group, which is covered by the *1* $\times$ *2* box, is c_{13}; the left or right forward faces of the *1* $\times$ *2* box are both $\{c_{13}, c_{14}\}$; c_{19} are the left, right, up, and down forward face of the "group" under the *1* $\times$ *1* box.
$\{c_{10}, c_{11}, c_{12}\}$ are the corresponding side of the right forward face of the *3* $\times$ *3* box's group. When arranging the box to move right, they have to be empty with out any confirmations.

6.1.1.2 Building procedures of the tree structures

The process of building the relationships are represented in Figure 116 to 119. The build process starts at MUL_r, and S_1 is built earlier than S_2. When a box enters the system or it moves to a new destination, MUL_r starts to perform this procedure.

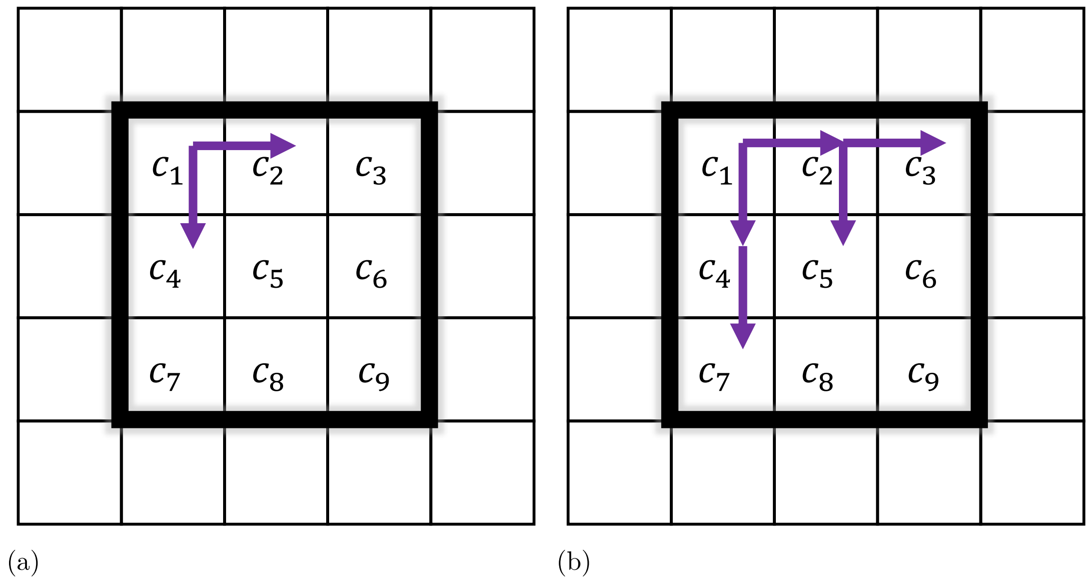


Figure 116: Steps of building S_1 (part 1). The building process starts at MUL_r. While connecting conveyor modules, the box information is also spread among these conveyors.

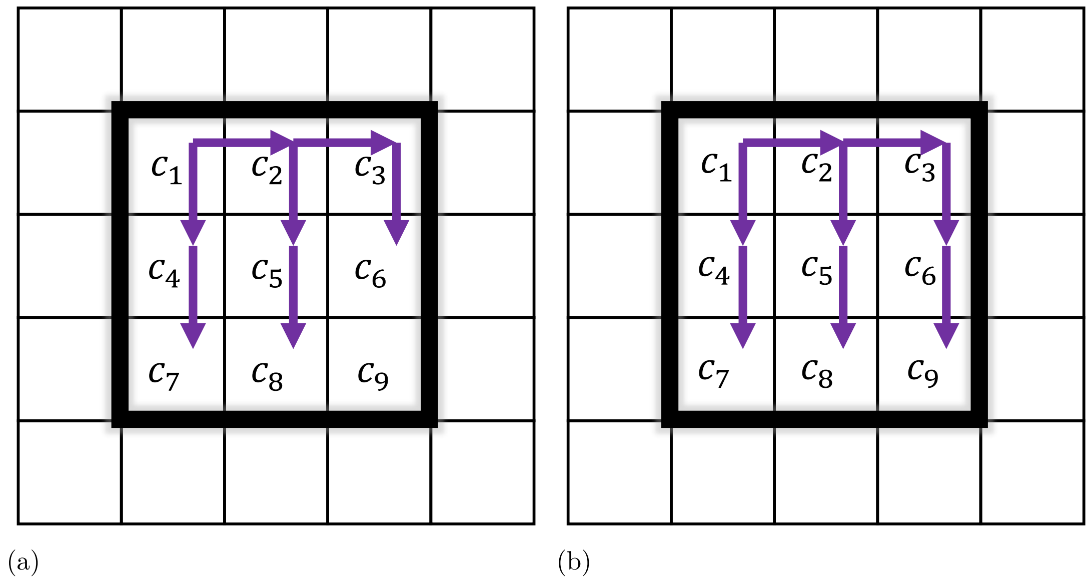
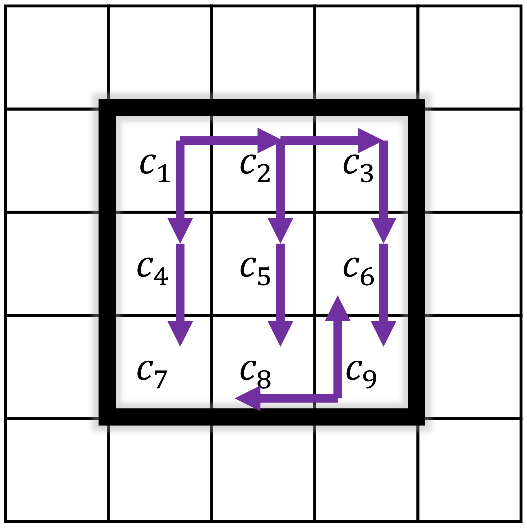
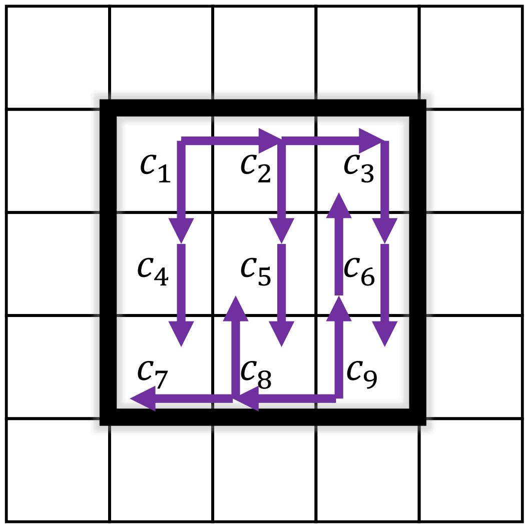


Figure 117: Steps of building S_1 (part 2). The build process continues until all conveyor modules underneath a box are connected.

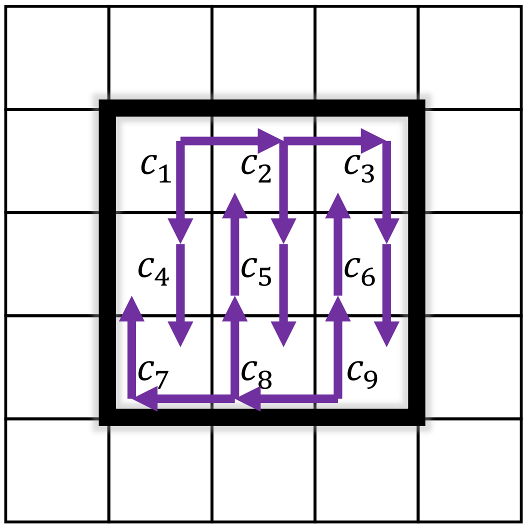


(a)

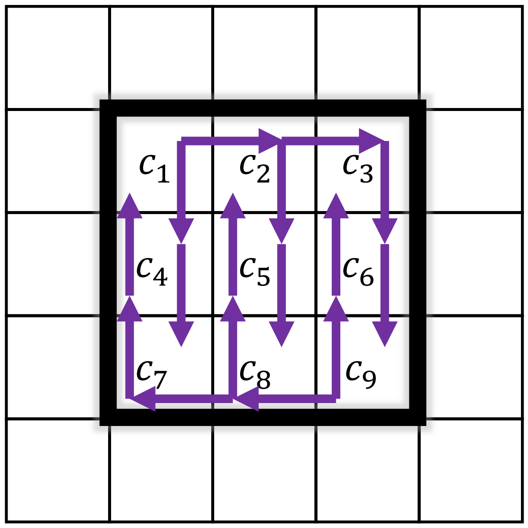


(b)

Figure 118: Steps of building S_2 (part 1). The building process starts at MDR_r.



(a)



(b)

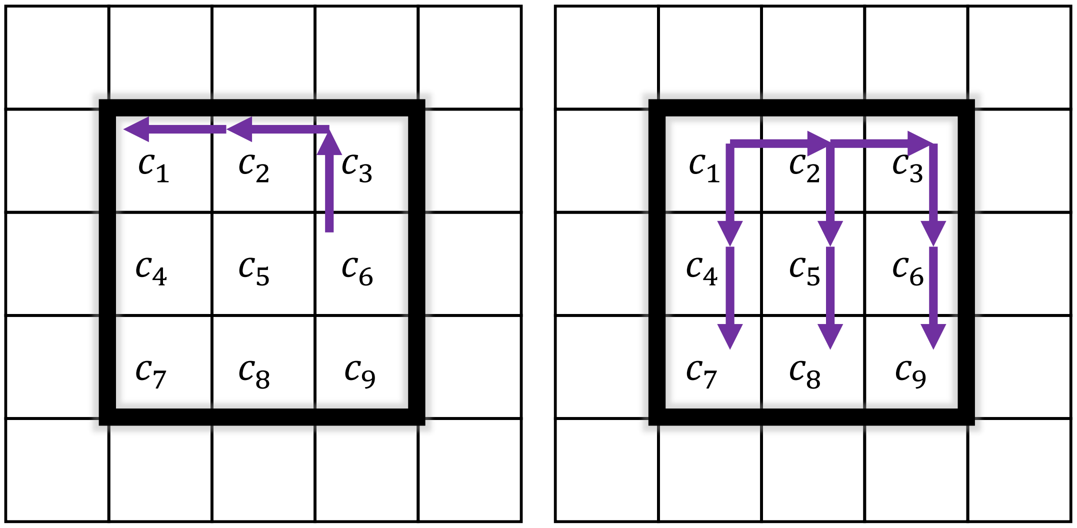
Figure 119: Steps of building S_2 (part 2).

6.1.1.3 Use cases of S_1 and S_2

S_1 and S_2 are used to share information and pass messages with the group of conveyor modules occupied by a single box.

Case 1 In this case, the group of conveyor modules is preparing to move a box. All of the box's information, which is stored in every member of the group except MUL_r, is cleared sequentially in S_1. When moving the box, MUL_r orders group members to move. MUL_r keeps all of the box's information and transfer tasks. At the new conveyor module, new relationships are built by the methods described previously.

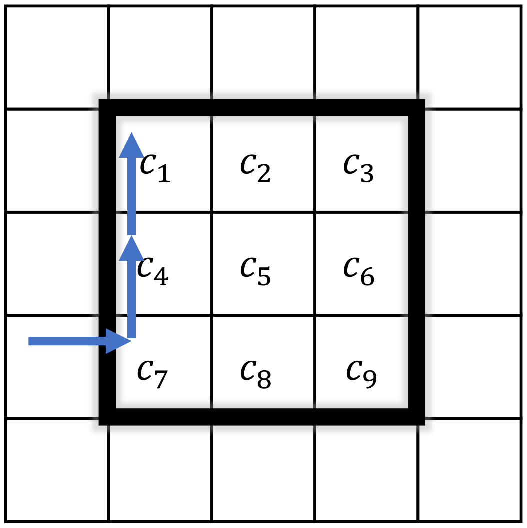
Case 2 In this case, the group of conveyor modules share their box's information or transfer tasks with the other members in the group (Figure 120a).



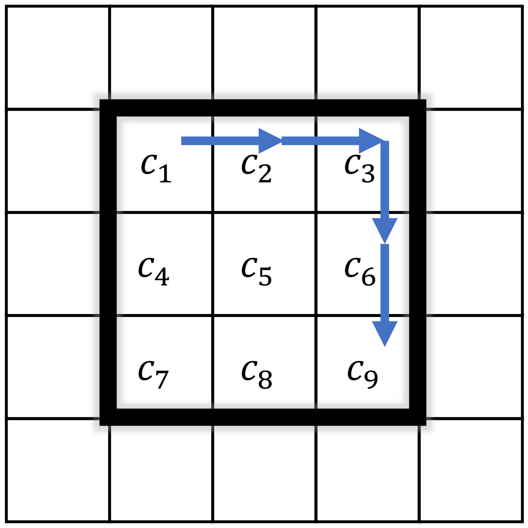
(a) Report information changes. (b) Distribute information changes.

Figure 120: Example of use case 2. When c_6 needs to make changes in the box's transfer task, c_6 reports updates via S_1 back to MUL_r along the purple arrows. After the changes is reported, MUL_r distributes the updated information to every member through S_1 along the arrows.

Case 3 In this case, the group of conveyor modules process a negotiation message (See Figure 121 and 122 for examples). We call the entire process in this example the *report-execution process.*

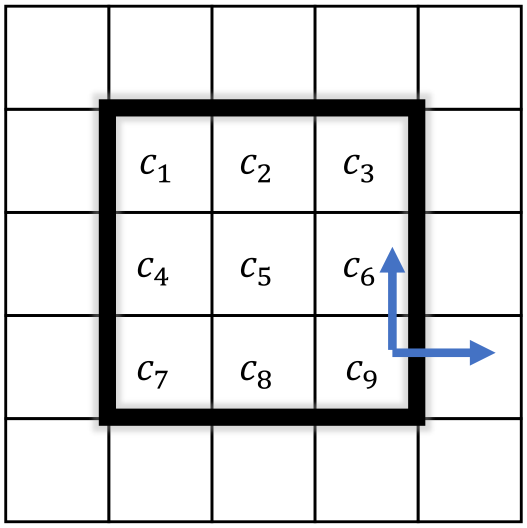


(a) Report the received negotiation message.

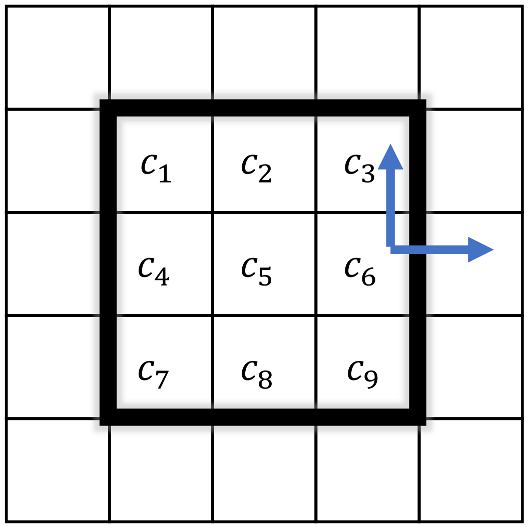


(b) Execute the decisions.

Figure 121: Example of use case 3 (report and executing). When a seek message having a right passing direction is received by c_7, it reports the message to MUL_r (c_1) through S_1. The negotiation message is processed at MUL_r. Suppose the decision is to pass the seek message, the seek message is delivered to MDR_r according to the message's passing direction.



(a) c_9 starts duplicating the activities.



(b) Continue to duplicating.

Figure 122: Example of use case 3 (duplication of passing messages). MDR_r (c_9) leads all conveyor modules in the forward face to pass the messages, which is the process of duplicating messages.

6.1.2 Adjacent types of forward faces

In the unit-sized GridHub, when two boxes are held by two neighboring conveyor modules, these two boxes face each other. A negotiation message passes from one conveyor module to its neighboring conveyor module completely, which means no other conveyor module can receive the message. In NU GridHub, the above scenario is not always true. To avoid mistakes and to simplify the negotiation procedure, we classify the case when two or more boxes are adjacency to each other. We call this *adjacency types* of a forward face.

Clear: a forward face's adjacency type is clear if it meets this condition: the corresponding side of the forward face is completely empty with no movement confirmation (Figure 123).

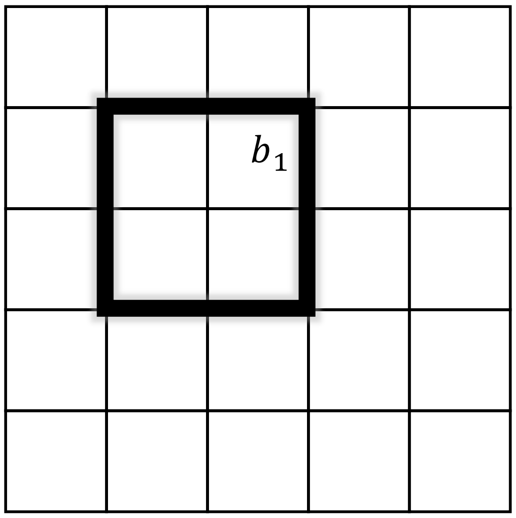


Figure 123: Clear: all forward faces' adjacency types of b_1 in are clear.

Empty confirm: a forward face's adjacency type is empty confirm if the conveyor modules in the corresponding side of a forward face are completely empty with movement confirmations in perpendicular direction(Figure 124).

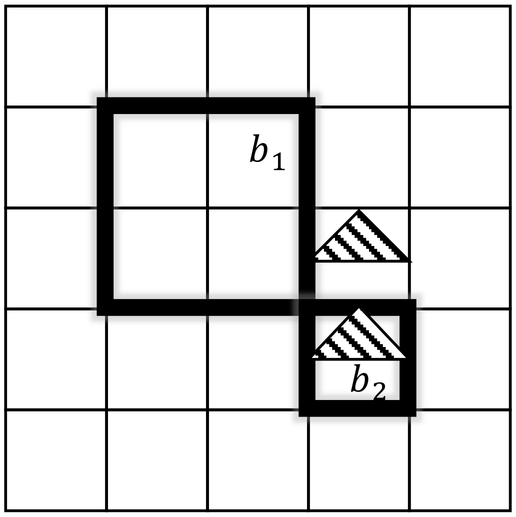


Figure 124: Empty confirm adjacent type: one module in the corresponding side of b_1's right forward face is empty, but this conveyor module has an up movement confirmation. Then, the adjacency type of b_1's right forward face is empty confirm.

Straight: a forward face's adjacency type is straight if the corresponding side of a forward face belongs completely to another group of conveyor modules which hold an entire box, and the two boxes have the same sized forward faces

(Figure 125). Furthermore, the adjacency types of the forward faces, which belongs to two neighbored unit-sized boxes, are both straight.

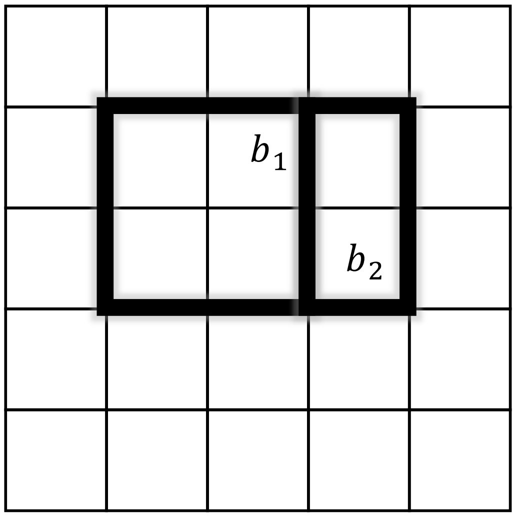


Figure 125: Straight adjacent type: the adjacency type of *b_1*'s right forward face and *b_2*'s left forward face are straight.

Semi-straight: a forward face's adjacency type is semi-straight if a pair of two contacted forward faces have different sizes, and one of forward face's neighbors belong to the same group of conveyors that carries another box (Figure 126).

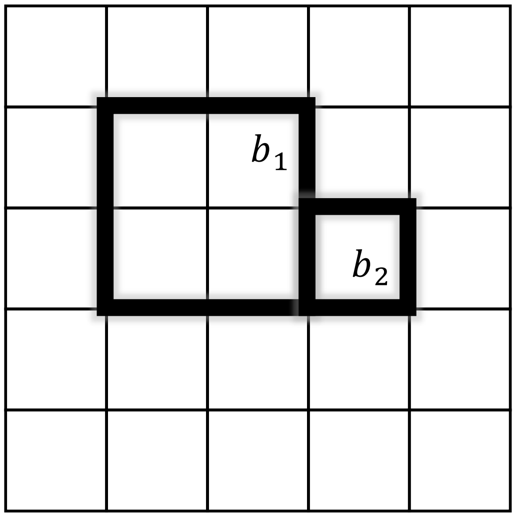


Figure 126: Semi-straight adjacent type: the adjacency type of *b_1*'s right forward face and *b_2*'s left forward face are semi-straight.

Lapped: a forward face's adjacency type is lapped if one or more of forward face's neighbors do not belong to the same group of conveyors that carries another box (Figure ??).

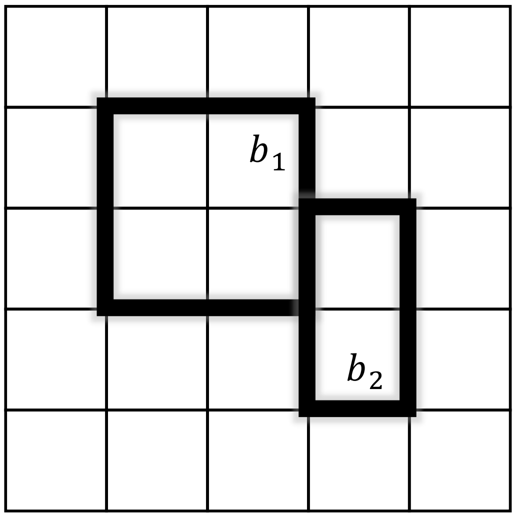


Figure 127: Lapped: the adjacency type of b_1's right forward face and b_2's left forward face are lapped.

A more complex example is in Figure 128.

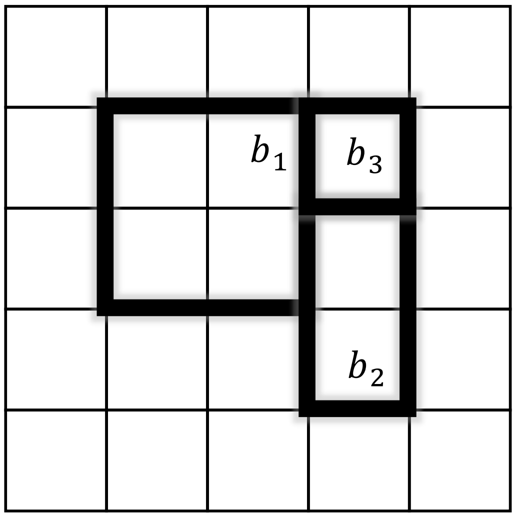
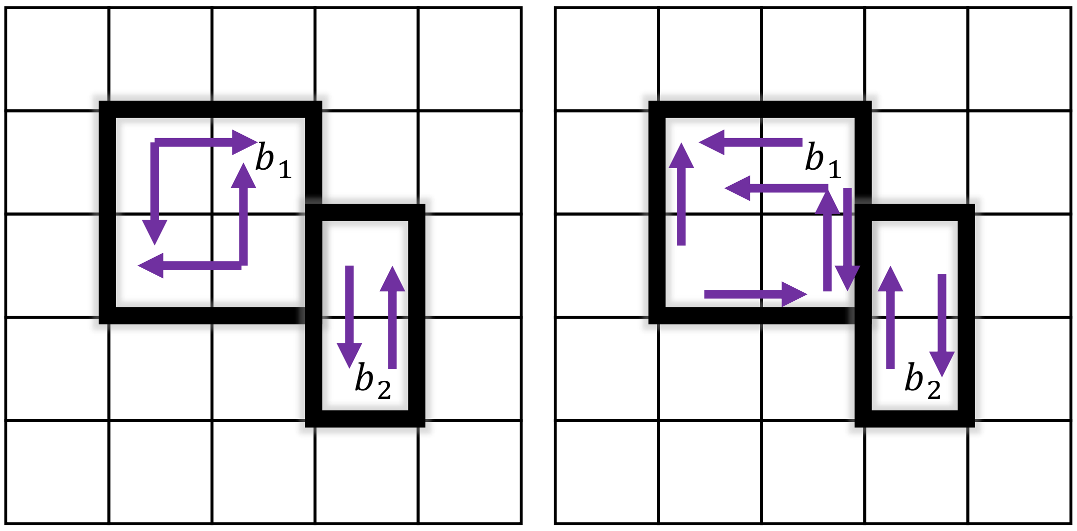


Figure 128: Mixed adjacency types: the adjacency type of b_1's right forward face and b_2's left forward face are lapped, but the adjacency type of b_3's left forward face is semi-straight.

The conveyor module on the forward face detects its adjacency type in every iteration, and the adjacency type may be updated during the negotiation. While the negotiation is in process and there is a need to change adjacency types, the conveyor module receiving the change request sends a message to the root master module which leads the forward face. When the root master receives the request, it replies with a message along the forward face, and it also reports to MUL_r using S_1.

There are many methods to detect the adjacency type of a forward face. We briefly described one method in this dissertation: First, MUL_r and MDR_r send messages along the forward faces together (see Figure 129a). Second, based on the information of the neighbors of every forward face, the messages record different "votes." Third, the module at the other end of a forward face decides the adjacency type by vote counting. Finally, the result is marked at every conveyor module of the forward face and reported to MUL_r via S_1 (see Figure 129b).



(a) Collect information. (b) Mark the adjacency type

Figure 129: Steps of one method to detect adjacency types. First, pass message to collect the related information of every member of every forward face. Then, mark the adjacency type at every conveyor, and report to MUL_r.

6.1.3 Storing and comparing of negotiation messages

In NU GridHub, conveyor modules can change the negotiation message's content while they are processing messages, and these messages may be duplicated when they are passed by the modules of a forward face. Consequently, the conveyor modules can produce different versions of messages. Processing multiple versions of messages may cause problems. Hence, the conveyor modules have to record and compare the content of negotiation messages.

To store and compare messages, the *class* of a message is defined according to the nested attempts which the message belongs. The details of a message class are in Table 15, and examples are in Figure 130, In every conveyor module, based on the classes, types (seek, confirm, and fail), and the current passing direction of messages, several buffers are established to store the content of messages being processed. Furthermore, the *current passing direction* of messages is the passing direction of a message when it is saved to a buffer.

TABLE 16: Definition of message classes.

Class	Nested Attempts
1	N_1 or $N_1([N_2, N_3, N_4])$
2	N_2 or $N_2([N_3, N_4])$
3	N_3 or $N_3(N_3)$
4	N_4

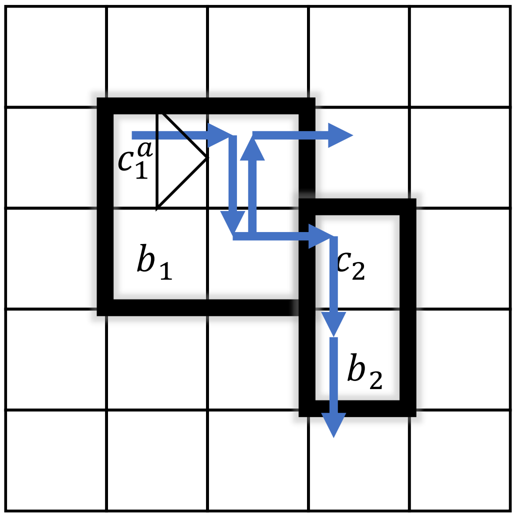


Figure 130: Example of class *1* and *2* message. when a seek message of N_2 is initiated by the group containing c_1^a, its class is *2*. When the message is processed by the group, which c_2 is in, then its class is *1* because c_2 processes $N_1(N_2)$.

Additionally, every negotiation confirmation at every conveyor module associates with the negotiation message which triggers the conveyor module to mark the confirmation, and the message is stored in a specific buffer. Besides the buffer to store the messages that mark confirmations, the following special buffers also exist in every conveyor module:

- Buffers of messages mark fake confirmations.
- Buffers of messages with a potential paths of (N_1 or $N_1([N_2, N_3, N_4])$) are overwritten.
- Buffers of messages with potential paths of (N_1) are overlapped.

All buffers at every conveyor module are cleared to empty at the end of each iteration.

When a conveyor stores a message into a buffer, a conveyor finds the buffer based on the message's class, type and current passing direction. Then, the conveyor module copies the entire content of the message into the buffer. For example, c_1^a in Figure 130 stores the seek message, which is class *2*, seek, and right, to a buffer; c_2 in Figure 130 stores the seek message, which is identifiable to a buffer as class *1*, seek, and down.

The property of a negotiation message includes the following attributes in the message's content:

- The original active conveyor module that initiates messages.
- The attempt belongs to (N_1 to N_4).
- The current and historical message passing direction. For example, the historical passing direction of the message ($N_1(N_2)$) in Figure 130 is the right (passing direction of N_2 which the $N_1(N_2)$ nests in).

In NU GridHub, the fingerprint of messages are only defined for messages belonging to class of *1*, which has the additional attributes:

- Message property

- Groups of conveyor modules which have passed this message, and every MUL_r of these groups is not in the same row or column of the MUL_r, which last passes the class *1* message. Additionally, while a class *1* message is being passed, the ID of the MUL_r, which meets the above condition, needs to be assigned to the message.
- The conveyor module group, which last processed this message.
- The group of conveyor modules, which initiate the N_1 or $N_1([N_2, N_3, N_4])$.

The purpose of comparing negotiation messages is to decide whether the same or a similar copy of a message has been processed before. The comparison is based on the messages' properties or fingerprints, and a comparison's result in a Boolean (yes or no). For example, in Figure 130, suppose another seek message is being processed by c_1^a, c_1^a finds a seek message buffer based on the message's class and current passing direction. Let us assume the message being processed has class *2*, and its current passing direction is right. Then, c_1^a compared all the stored messages to the properties of the message that is being processed. If the message being processed also belongs to attempt N_2, but the original active conveyor module is different than the message displayed in Figure 130, then they do not have the same properties.

6.1.4 Special rules of attempts

Based on the negotiation rule of the unit-sized GridHub, there are special rules for NU GridHub:

Forward face sizes and forward attempts If the active box has a forward face with a size greater than *1*, then no forward attempt is allowed (See the example in Figure 131 for details). Though we have methods to solve the confusion in in Figure 131, this confusion increases the complexity of processing messages. Hence, we disable forward attempts in this case.

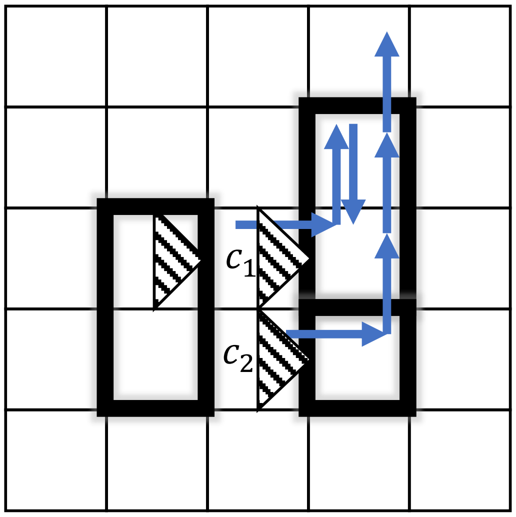


Figure 131: Example of restricting forward attempt. Let the group of conveyor modules carrying b_1 confirms a right movement in N_1. In this case, the corresponding side of the right forward face does not initiate a forward attempt. Otherwise, the negotiation messages initiated by these empty conveyor modules may have different content, and other groups of conveyor modules may be confused. For example, if c_1 and c_2 initiate two N_2 respectively, and $N_1(N_2)$ have different passing directions, then the modules that process both of these messages are confused.

N_1 and tandem movements active boxes In a unit-sized GridHub, N_1 only arranges single or tandem movements of active boxes. This is the major difference of N_1 and $N_1([N_2, N_3, N_4])$. In NU GridHub, N_1 can arrange the movements of non-active boxes. The reason to add this rule is to increase the transfer speed of active boxes or to avoid live-locks in NU GridHub. Example is in Figure 132.

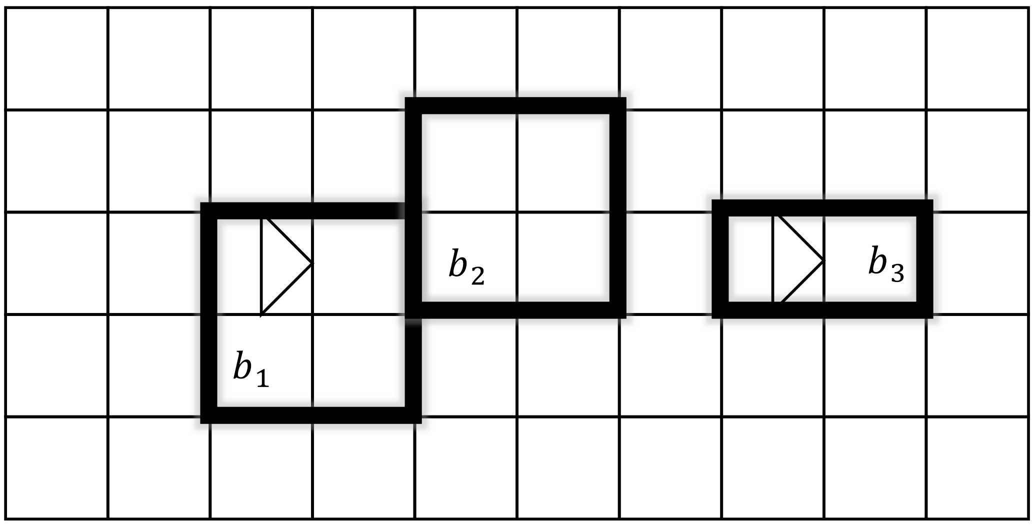
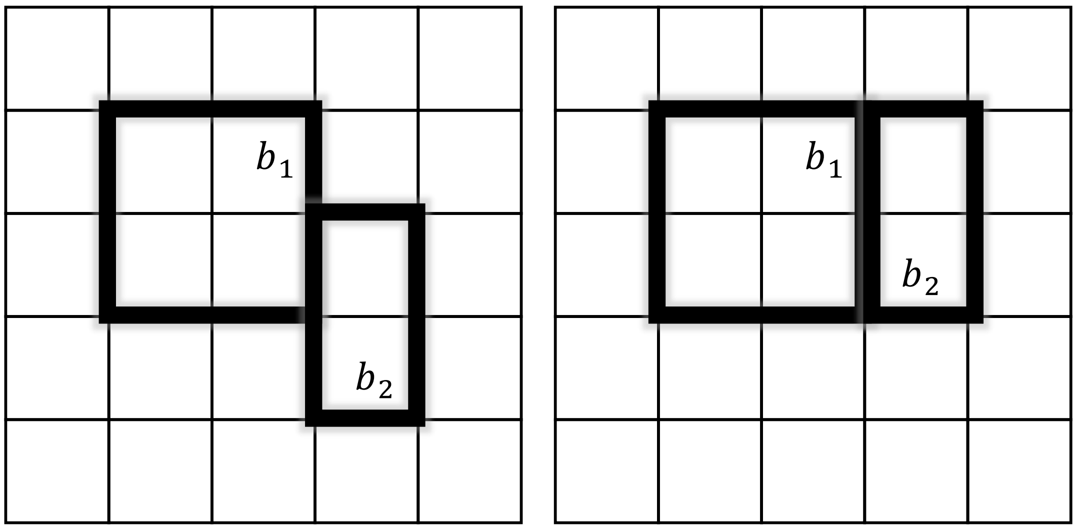


Figure 132: Bigger forward face can arrange movements of non active boxes in N_1. The active box (b_1) randomly decides whether to move the non-active box in N_1, but the 2×1 box (b_3) never allows this type of movement to right.

Movement restriction of initiating $N_1([N_2, N_3, N_4])$ To solve this problem shows in Figure 133, we recall the movement restriction to assign transfer tasks. In this case, we only check whether the box is moved from one direction, but do not try to arrange movement in the opposite direction. Additionally, this restriction is only effective when the corresponding forward face's size is greater than 1. When groups of conveyor modules pass N_2 or $N_2([N_3, N_4])$, they judge the size of the corresponding forward face and decide whether to change the content of the negotiation message. After the content of the message is changed, the conveyor module that initiates $N_1([N_2, N_3, N_4])$ knows whether to implement this rule.



(a) Layout 1 (b) Layout 2

Figure 133: The sense of cyclically moving a box in a NU GridHub. The group of conveyor modules may cyclically move b_1 in front of b_2 in a $N_1([N_2, N_3, N_4])$ (repeat switching between Layout 1 and Layout 2).

If the restriction is implemented, the box will try to move up in the next iteration instead of moving down.

Check adjacency type before passing message When passing seek messages of different attempts, the MUL_r of a conveyor module group decides whether to pass or reply fail, according to the adjacency type of the corresponding forward face.

First, when a seek message of the other attempts (N_2 to N_4) are being processed by a group of modules:

- If the corresponding forward face's adjacency type is clear, straight, or semi-straight, the conveyor module group can pass the message.
- If the corresponding forward face's adjacency type is empty with confirmation or is lapped, every member of the forward face can pass the message only when the destination of the message is occupied.

Second, when a seek message of seek message of N_1 or $N_1([N_2, N_3, N_4])$ are being processed by a group of modules:

- If the corresponding forward face's adjacency type is clear, straight, or semi-straight, the conveyor module group can pass the message.
- If the corresponding forward face's adjacency type is empty with confirmation, the conveyor module group cannot pass the message.
- If corresponding forward face's adjacency type is lapped (see Figure 134).

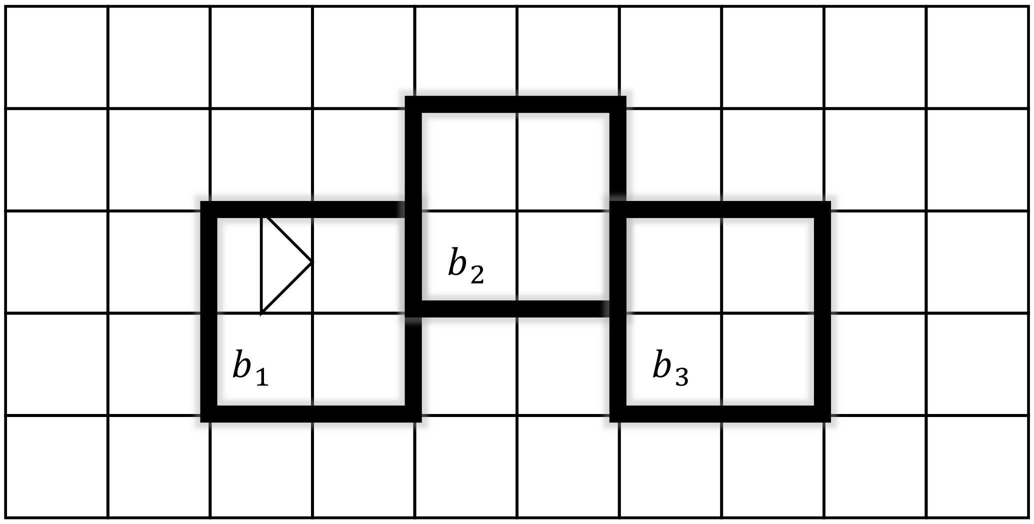


Figure 134: Example of pass seek message when the forward faces' adjacency types are lapped. Suppose b_1's group initiates a N_1 that allows tandem movements. Groups b_2 or b_3 can pass the seek message only when the vertical distance between b_2 or b_3's MUL_r and b_1's MUL_r is less than two times b_1's right forward faces. For seek messages that have different passing directions, the condition is symmetric.

Backtrack confirms or fails When a group of conveyors passes a confirm or fail message of N_1 or $N_1([N_2, N_3, N_4])$, MUL_r needs to initiate a backtrack message to change related information to the empty conveyor on the "back" of the message passing direction.

After an empty module processes a backtrack confirmation message:

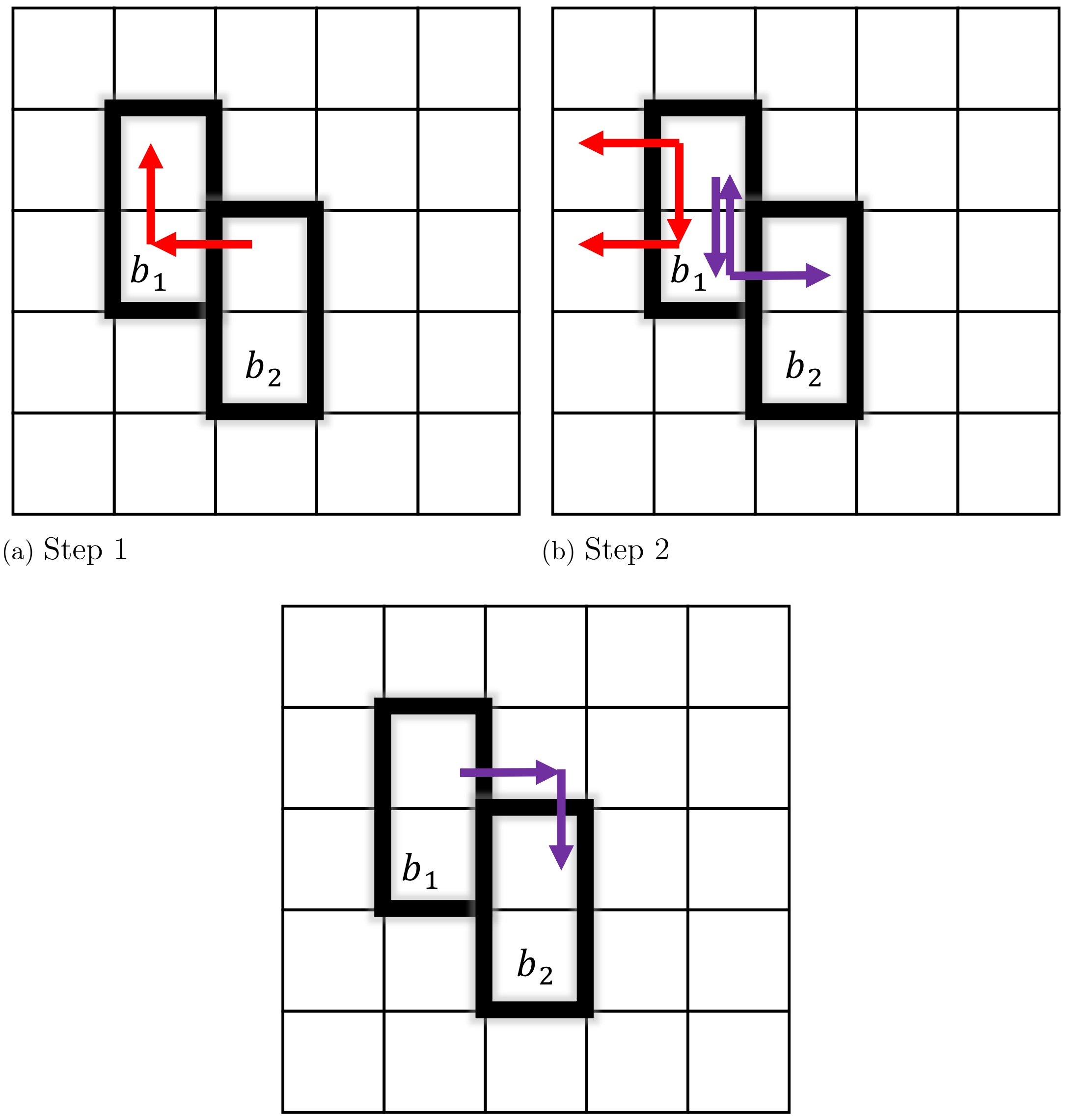
1. It checks whether the seek message in the buffer with the same property has been processed. If not, it does nothing; else,
2. it follows the same rule as the unit-sized GridHub to judge whether to mark confirmation. If no not, it does nothing; else,

3. it marks confirmation, stores the backtrack confirmation message and sends messages to the perpendicular neighbors to ask them to change their forward faces' adjacency type to "empty confirm."

After an empty module processes a backtrack fail message:

1. It checks whether the seek message in the buffer with the same property has been processed. If not, it does nothing; else,
2. it checks whether there are confirmations placed;
    - If not, it sends messages to the perpendicular neighbors to ask them to change their forward faces' adjacency types to "empty confirm."
    - If so, it performs the above actions, and it tries to remove the confirmation by comparing the property of stored confirmation messages to the backtrack messages.

Example is in Figure 135: when a fail message is passed to b_1's group, steps of backtrack fail are proceeded sequentially.



(a) Step 1 (b) Step 2

(c) Step 3

Figure 135: Steps of passing backtrack fail messages.

Remove wrong confirms When a fail message of N_1 or $N_1([N_2, N_3, N_4])$ is processed at any MUL_r, the new fail messages are created and passed back to the original group of conveyor modules. This process only occurs for the messages stored in the special buffers having the same message passing direction. Fail messages having the opposite passing directions to messages in the seeking buffers are also created for all messages.

6.1.5 Passing negotiation messages

We list the procedure of passing negotiation messages, using the concepts explained above.

6.1.5.1 Passing seek messages of $[N_1, N_1([N_2, N_3, N_4])]$

The empty module follows the same rule as the unit-sized GridHub to decide whether to reply a confirm or fail message when processing a seek message. After the empty module decides to reply with a confirm, the steps are the same as when it process a backtrack confirm message.

When a seek message is received by any member of a group conveyor module, the message is reported through S_1 to MUL_r (see use case 3 in Section 6.1.1). All of the decisions are made by MUL_r. Besides the rules defined for the unit-sized GridHub, it also needs to:

1. Check the adjacency types and movement restrictions to decide whether to pass the message. If the check result is not, it replies fail; else,
2. Check whether there are processed fail messages with the same properties. If the check result is yes, it replies fail; else,
3. Check whether the processed seek messages have the same fingerprint. If yes, it does nothing; if not, it passes the message following the execute procedure (see use case 3 in Section 6.1.1). The seek message also needs to be put into the corresponding message buffer.

6.1.5.2 Passing confirm messages of $[N_1, N_1([N_2, N_3, N_4])]$

When any member of a group of conveyor modules receives a confirm message, it reports the message through S_1 to MUL_r. Besides the rules defined for the unit-sized GridHub, it also needs to (steps also in Figure 136):

1. Check whether there are processed seek messages with the same properties. If no, it does nothing; else,
2. Check whether there are fail process messages with the same properties. If yes, it stops passing the confirm message and starts to “remove wrong confirms.” if

no, the following possibilities exist:

- Check whether there are processed confirm messages with the the same fingerprint. If yes, it does nothing. If not, it marks confirmation and stores this message as “confirm by message.” Then, it passes the confirm message, sends backtrack confirms, and also stores the message to a corresponding buffer.
- Check whether there are confirmed messages being overwritten, the process of “removing wrong confirms” also needs to be started. This confirm message is recorded by overwriting other messages.
- If the confirm message makes fake confirmations, it also needs to be recorded.

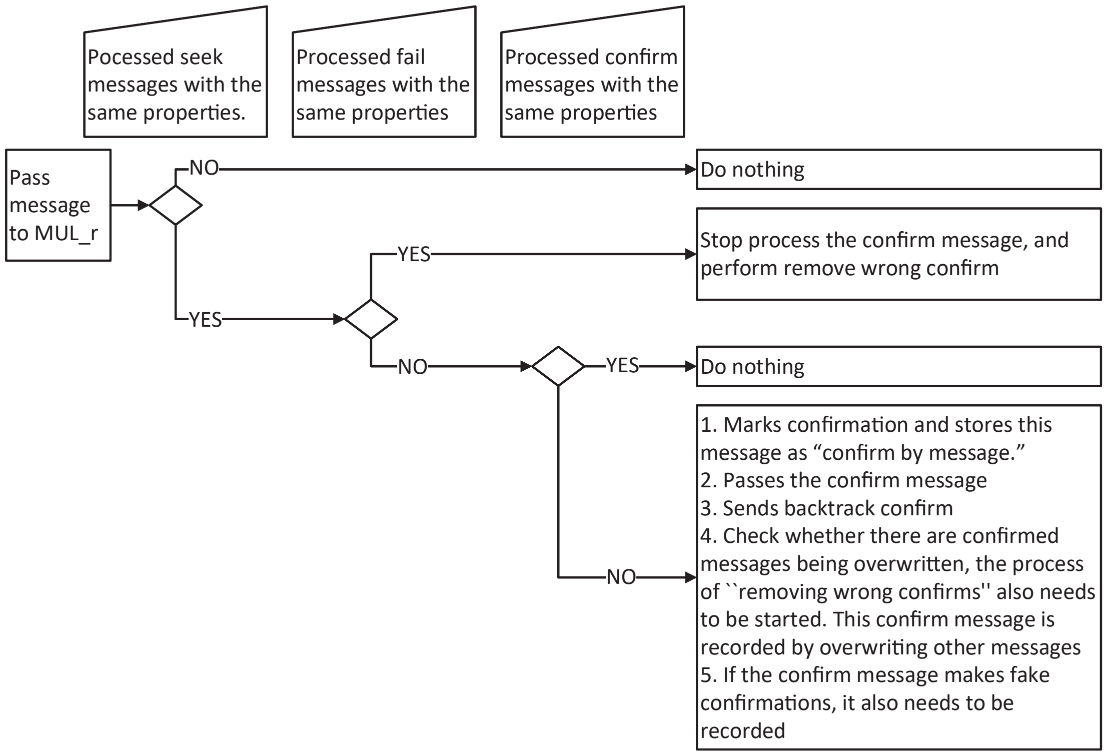

Pocessed seek messages with the same properties.
Processed fail messages with the same properties
Processed confirm messages with the same properties
Pass message to MUL_r
NO
Do nothing
YES
Stop process the confirm message, and perform remove wrong confirm
YES
NO
YES
Do nothing
NO
1. Marks confirmation and stores this message as “confirm by message.”
2. Passes the confirm message
3. Sends backtrack confirm
4. Check whether there are confirmed messages being overwritten, the process of ``removing wrong confirms'' also needs to be started. This confirm message is recorded by overwriting other messages
5. If the confirm message makes fake confirmations, it also needs to be recorded

Figure 136: Actions to passing a confirm message of $[N_1, N_1([N_2, N_3, N_4])]$.

6.1.5.3 Passing fail messages of $[N_1, N_1([N_2, N_3, N_4])]$

When a fail message is received by any member of a group (when the conveyor module is occupied), the message is reported through S_1 to MUL_r.

Besides the rule defined for the unit-sized GridHub, it also needs to:

1. Check whether there are processed seek messages with the same property. If not, it does nothing; else,
2. Check whether there are processed fail messages that have the same fingerprint. If yes, it does nothing; else,
3. Check whether there are processed confirm messages that have the the same property. If yes, it clears confirmations marked with the confirm by a message with the same property. Then, it starts the process of "remove wrong confirms," and passes the fail message; if no, it just passes the fail message.

Every conveyor module can only remove negotiation confirmations by processing fail messages. Since every confirmation mark is associated with the message which triggers the module to mark it, the conveyor module has to ensure that the fail message has the same property as the message that marks the confirmation before removing the confirmation.

6.1.5.4 Examples of passing messages of $[N_1, N_1([N_2, N_3, N_4])]$

One example of passing confirm and fail messages is in Figure 137 to Figure 139. In these examples, while fail messages are generated or sent, activities of removing wrong confirmations are performed. These message passing activities are not shown in the figures.

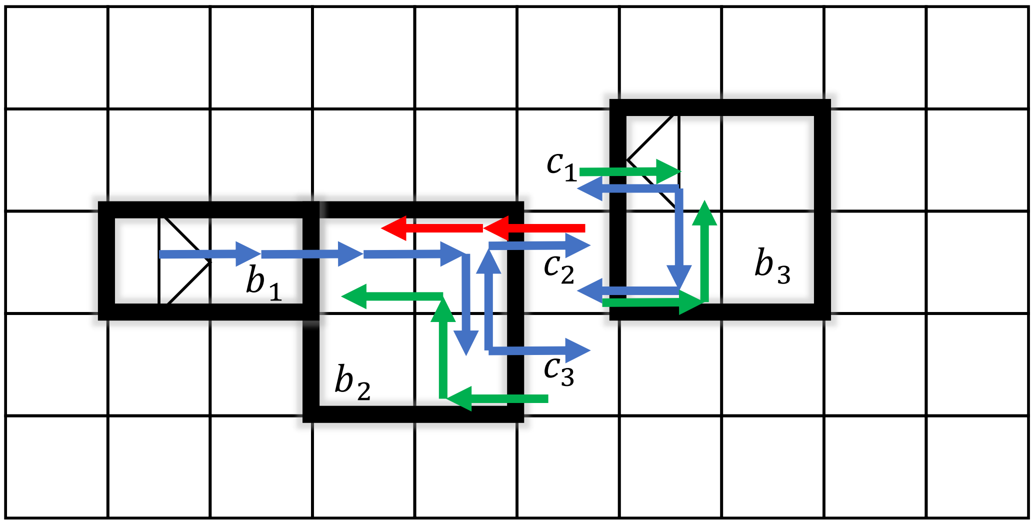


Figure 137: Example of passing seek messages. The seek message initiated by b_1 arrives at c_2 later than the seek message initiated by b_3. Hence, both the fail and confirm messages are passed to MUL_r of b_2. It is possible for MUL_r of b_2 to process either one first.

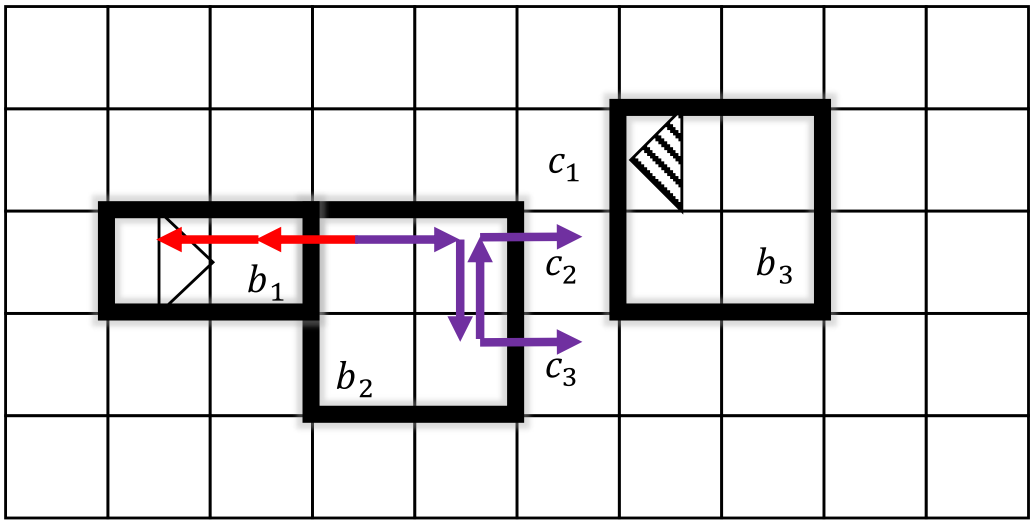


Figure 138: If process the fail message first. If the MUL_r of b_2 processes the fail message first, the confirm message will not pass because the fail message with the same properties is stored in the buffer. The MUL_r of b_2 also triggers backtrack for the fail message.

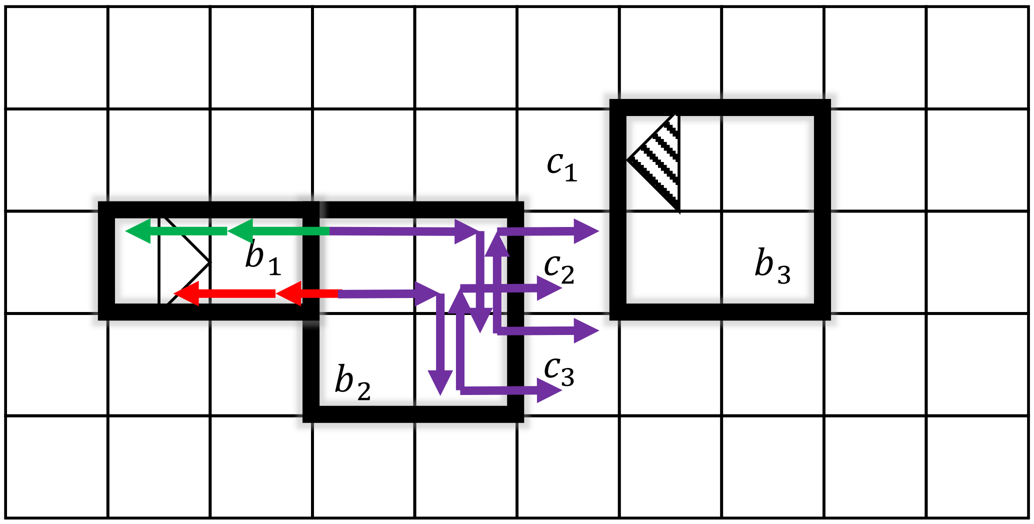


Figure 139: If process the confirm message first. If the MUL_r of b_2 processes the confirm message first, the fail message can be passed following the confirm message. The fail message also removes confirmations because the confirm messages with same properties are stored in the buffer. The MUL_r of b_2 also triggers backtrack for both confirm and fail messages.

6.1.5.5 Passing negotiation messages of other attempts

When a negotiation message is received by any occupied member of a group of conveyor modules, the message is reported through S_1 to MUL_r. Besides the rule defined for the unit-sized GridHub, it also needs to:

1. Check whether there are processed messages with the same property. If yes, it does nothing; if no,
2. It passes the message and records the message to buffer.

When the seek message is duplicated along the forward face, the rules are described above. When an empty conveyor receives a negotiation message, the rules are the same as the unit-sized GridHub's.

6.2 System Performance

The way of measuring the performance of NU GridHub is different than the method used by the unit-sized GridHub. In NU GridHub, the number of boxes is much smaller in the unit-sized GridHub. The storage density is lower too. Hence, considering the number of working boxes and empty conveyors is not helpful when exploring the system performance.

Since NU GridHub can hold boxes with different sizes, there are many combinations of box sizes. For example, there are *10* boxes in a NU GridHub. In these *10* boxes, *5* of them have size 3×1, and the other *5* boxes have size 2×2. If we consider more various sizes of boxes, the above combination have more cases, which even closes infinite. In order to measure the relationships between these two factors and the system throughput, we consider some of these combinations with different system setups.

6.2.1 Experiment setups

We use a NU GridHub with *12* rows and *12* columns (excluding the system edges). NU GridHub used for experiments is displayed in Figure 140. The gates are placed on the edges of the system. For any group of conveyor modules, if the conveyor module holds a box assigned departure information, the MUL_r faces the gate directly. For instance, the box in Figure 140 leaves at gate 4 on the down edge, and its MUL_r is in the same column of the gate.

The experiments are all run by simulations in AnyLogic. We use the "CONWIP" (Gue et al., 2014) rule to operate the system: When a box is assigned to one gate and leaves the system, another box is randomly selected to be assigned for departure information, leaving at the same gate. Each of the settings in Table 17 is run for *8000* iterations for every replication, with the first *800* iterations marked as the warm-up period. Every setting in this table is replicated *50* times.

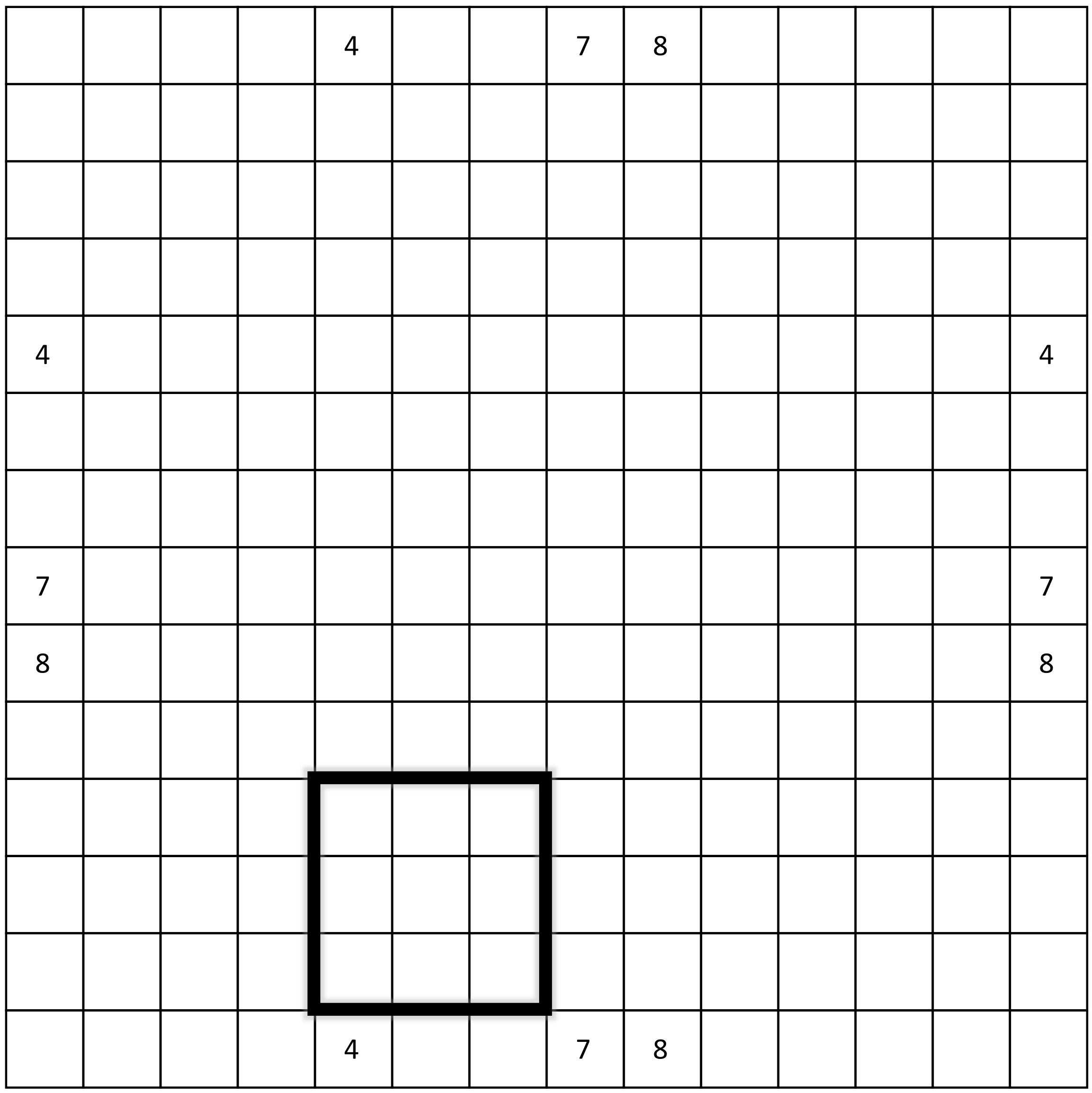


Figure 140: NU GridHub used for experiments.

TABLE 17: Experiment settings.

Name	Box sizes	Number of boxes	Density	Gates used
E1	1×2	36	0.5	4, 8
E2	2×1	36	0.5	4, 8
E3	2×2	18	0.5	4, 8
E4	2×2, 2×1, 1×2, 1×1	36	0.5625	4, 8
E5	3×2	8	0.5	4, 7
E6	2×3	8	0.5	4, 7
E7	3×3	8	0.5	4, 7
E8	3×3, 3×2, 2×3, 3×1, 1×3, 2×2, 1×2, 2×1	16	0.4861	4, 7

6.2.2 Results

The throughput (boxes released from every gate) of different settings is in Figure 141 and 142.

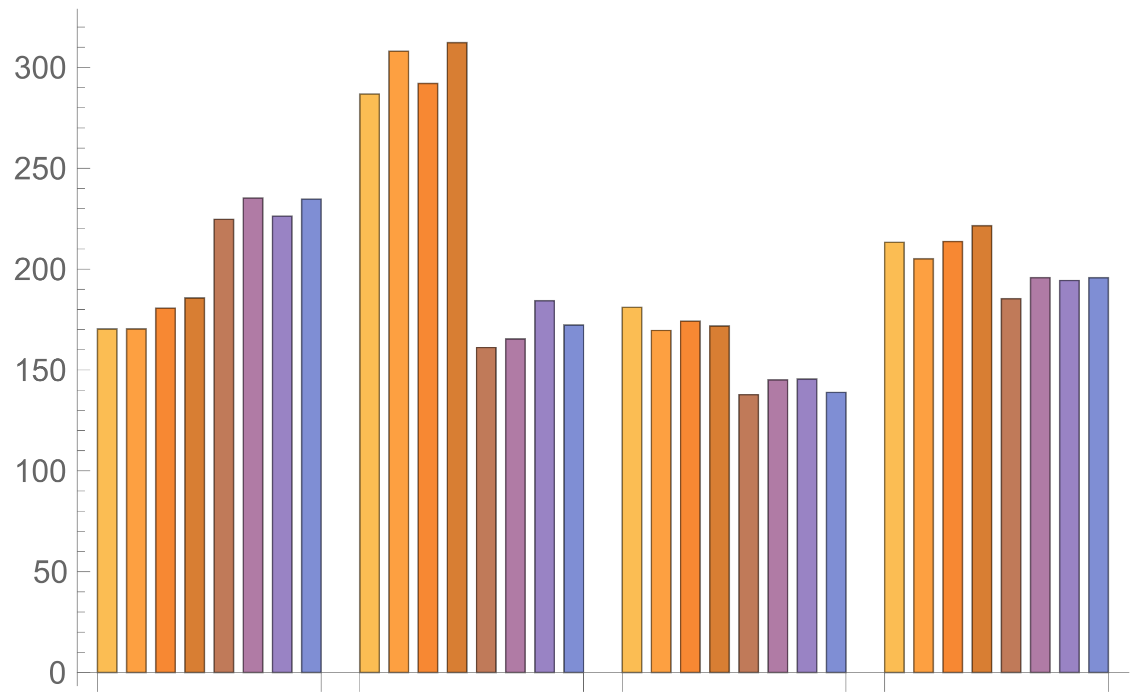


Figure 141: Throughput (the total number of boxes leave) of gates when the maximum size of boxes is 2×2. Each column represents a gate's throughput. The columns are gathers by the sizes of boxes in system (from left to right): 1×2, 2×1, 2×2, and mixed sizes (1×1, 1×2, 2×1, 2×2).

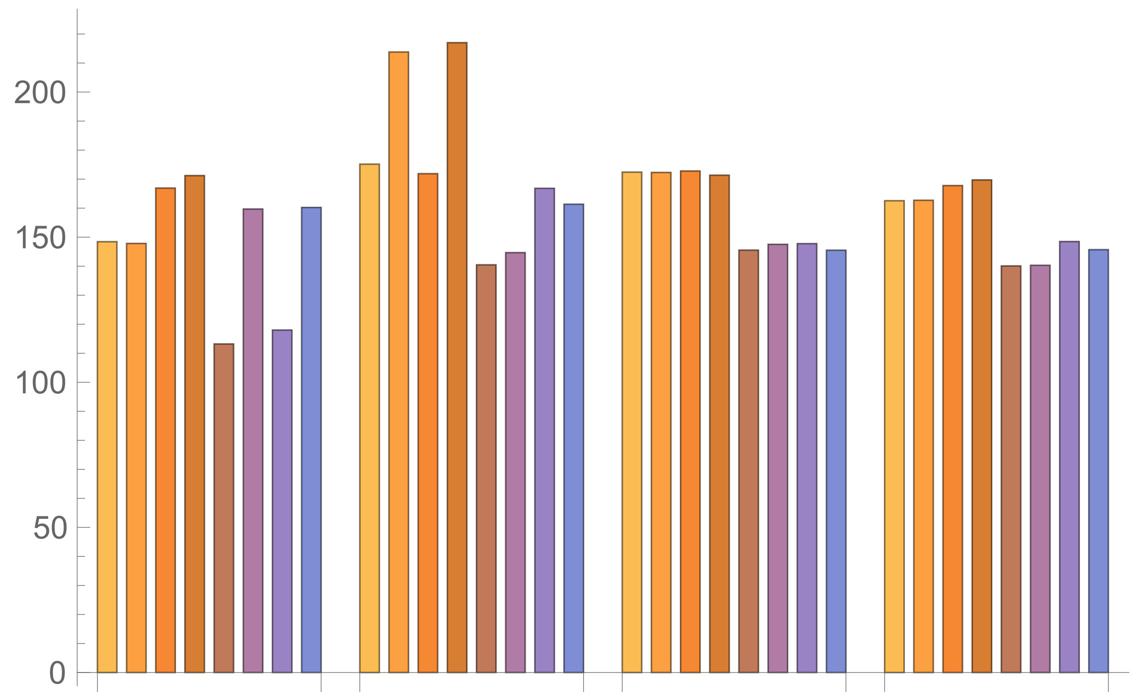


Figure 142: Throughput (the total number of boxes leave) of gates when the maximum size of boxes is 3×3. Each column represents a gate's throughput. The columns are gathers by the sizes of boxes in system (from left to right): 2×3, 3×2, 3×3, and mixed sizes (1×2, 2×1, 2×2, 1×3, 3×1, 2×3, 3×2, and 3×3).

For the mixed cases (s4 and s8), the average transferring time (measured by the number of iterations) of same sized boxes are displayed in Figure 143 and 144.

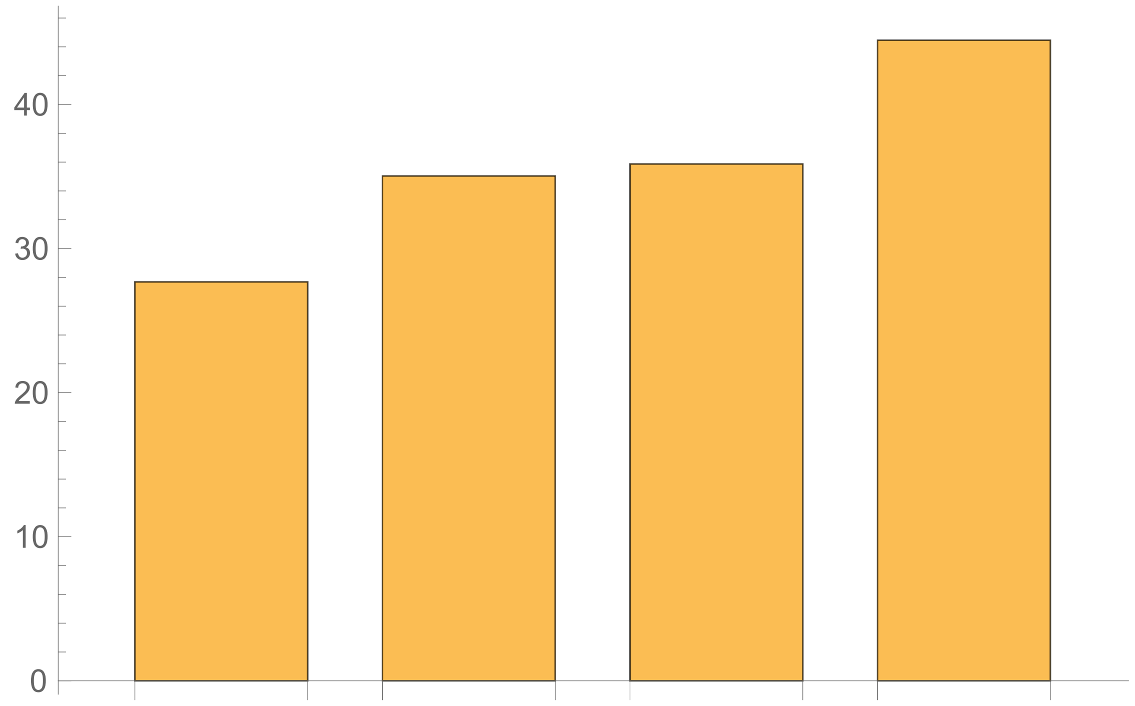


Figure 143: The average retrieval time of boxes with different sizes (part 1). Each column represents retrieval time of one box size (from left to right): 1×1, 1×2, 2×1, 2×2.

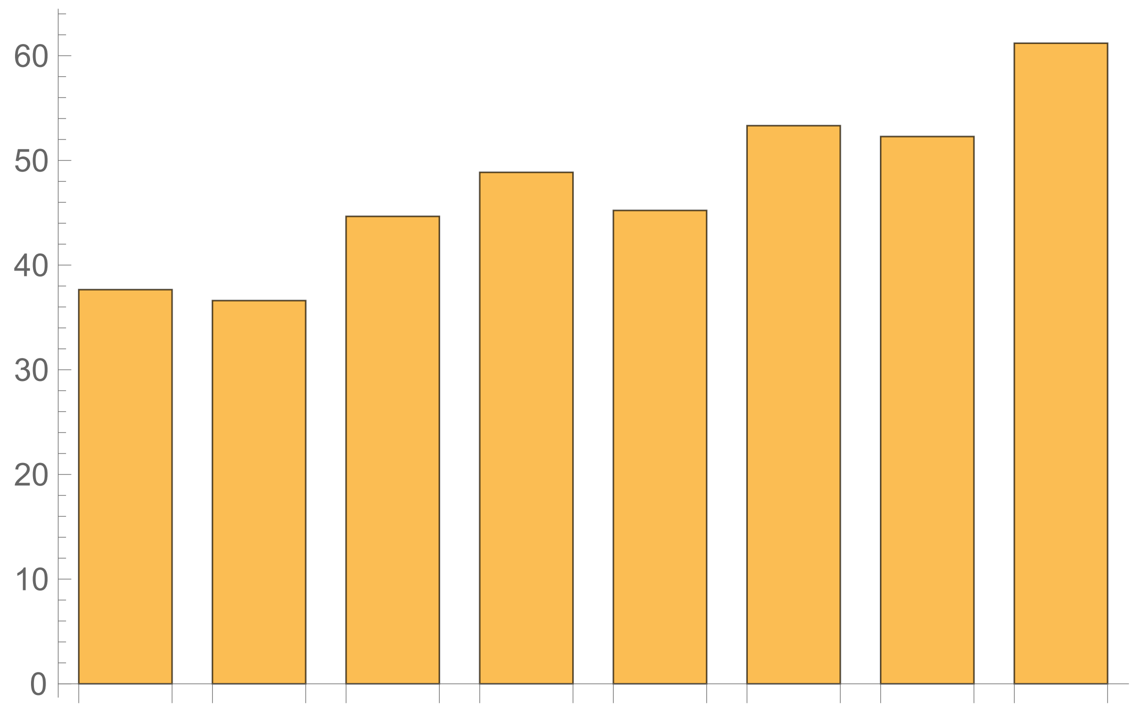


Figure 144: The average retrieval time of boxes with different sizes (part 2). Each column represents retrieval time of one box size (from left to right): 1×2, 2×1, 2×2, 1×3, 3×1, 2×3, 3×2, and 3×3.

The results suggest that the throughput of every gate is not equal. Even for a pair of gates with the same location on a pair of edges, they do not have equal throughput. For example, the gates on the left edge may have lower throughput than the gates on the right edges. The reason is the messages, which have different passing directions, take different times to reach a MUL_r in a group. Consequently, the MUL_r always processes the messages in one direction before processing the messages in the opposite direction. This effect is obvious when a group of modules process a message to start $N_1([N_2, N_3, N_4])$. One example is in Figure 145.

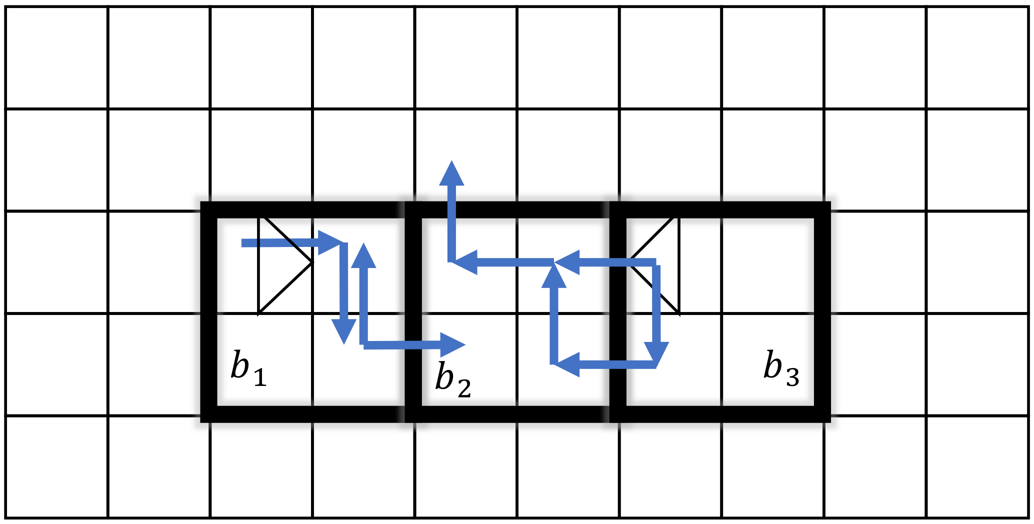


Figure 145: Message processing sequence. b_1 and b_3 are active. When both try to initiate N_2 at MUL_r of b_2, the seek message from b_3 is always received earlier than the seek message from b_1. Consequently, boxes with left active direction may move faster than the boxes with the right direction. In Figure 141, when there is only 2×2 box, then the gates on the left edges have higher throughput than the gates on the right edges.
Either b_1 or b_3 has to wait at least 2 iterations to move their immediate destinations.

The first reason the boxes move slower than unit-sized boxes is the time it takes to clear their immediate destination (see Figure 145). The second reason for slow transfer speed is that more empty conveyor modules are needed to arrange the movement of boxes (Figure 146). We summarize the following necessary conditions for moving one or multiple boxes. These two conditions are very hard to meet in NU GridHub, which reduce the throughput dramatically.

- There are no factors that can make any MUL_r reply with a fail message to a seek message of N_1 or $N_1([N_2, N_3, N_4])$.
- For the forward faces in the moving direction, the empty corresponding sides have to be available.

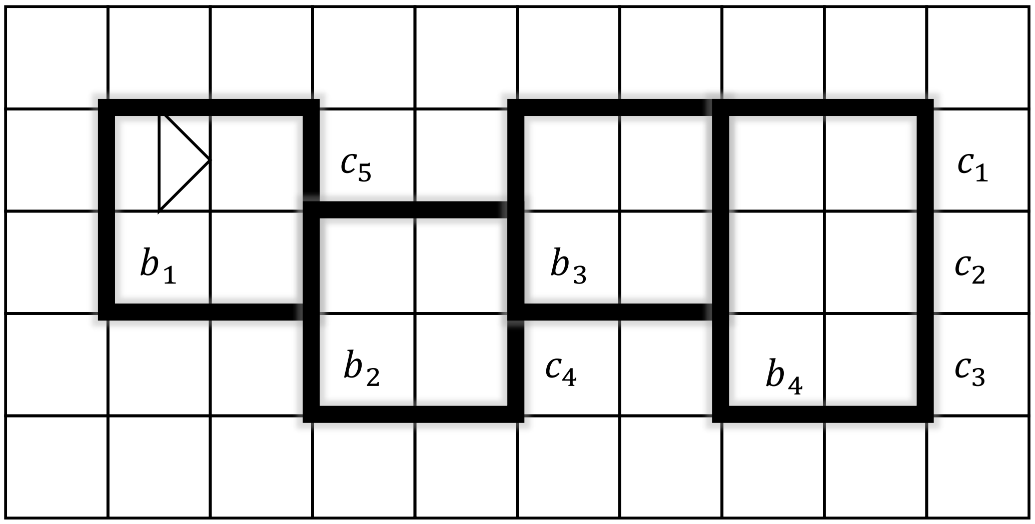


Figure 146: Examples of empty conveyors needed to move a line of boxes. To move the group of boxes, c_1 to c_5 have to be available.

Additionally, it is very difficult to find deadlock and livelock-free conditions for NU GridHub, but we have not encountered any of these problems when we run hundreds of simulation replications.

CHAPTER 7

CONCLUSIONS AND CONTRIBUTIONS

7.1 Conclusions

The unit-sized GridHub's control algorithms are message passing based. All messages are processed from the message buffer according to the conveyor's current state. To solve the competitions of empty modules among active modules, we introduce priority directions, and the sequencing of different attempts.

When an active conveyor module successfully finds an empty conveyor, through any attempt of negotiation, box movements can be arranged. The failed negotiations are summarized as blocks. We have proven that no blocks can ever persist in the GridHub, and the GridHub is deadlock free.

Conveyor modules perform other patterns of actions, and these actions are the negotiation behaviors. The negotiation behaviors can affect system performance, which include mute activities related to livelock. We examine several special cases of the GridHub and conclude that they are livelock free. For the other GridHubs, livelock risks are explained, and we list the methods to reduce the risks.

The performance of the GridHub is measured by the system throughput (boxes released per iteration), which is affected by several factors. The number of working boxes and the number of empty modules greatly affect the system throughput. We also measured the occurrence of negotiation behaviors, and show that they affect the active boxes' transferring process.

We extend the unit-sized GridHub to the NU GridHub. The NU GridHub is capable of transferring unit-sized boxes also. In the NU GridHub, the conveyor modules can build relationships to hold NU boxes, and process negotiations with additional negotiation methods.

When handling NU boxes, the performance of the NU GridHub is lower than the unit-sized GridHub, and the system throughput of every gate may differ due to the message passing sequence. More empty modules are required to move boxes which occupy the same number of conveyor modules.

7.2 Contributions

7.2.1 Moving boxes in four directions

This is the first method that enables GridHub to transfer active boxes in four directions, following the virtual aisle method.

Grid-based systems using the virtual aisle method seek empty spaces to change locations and move requested items. The GridStore, GridPick, and GridSequence were all designed to accomplish these activities. When we use this method to categorize empty modules' locations, we discover that control algorithms in GridStore and GridSequence only search and change empty modules' locations in category 1 and 2; GridPick provides another method to search and change empty modules located in category 3, but the method is limited. We develop GridHub based on the existing systems and introduce a new method for comprehensive searches in order to find empty modules in the grid. We also introduce priority directions and muting to solve conflicts that occur during the search process.

GridPick and GridSequence inherit the core algorithms of GridStore. In their control algorithms, methods were added to "mark" requested items, such as the "balancing" methods in GridPick. However, GridStore's control algorithms are still used to arrange box movements. Hence, the GridStore' control algorithms are movement management methods. We use this idea to develop GridHub's control algorithms, which are comprehensive movement management methods. This becomes the foundation for GridHub's architecture.

7.2.2 Decoupling material handling tasks and box movements

We introduce a new architecture for GridHub, which can separate the control algorithms of box movements and specific material handling tasks.

In existing grid-based systems, control algorithms are designed to address specific material handling tasks. In GridHub, control algorithms only address box movement, indicated by transfer tasks. We translate material handling tasks into transfer tasks. Consequently, transfer tasks are the only input for GridHub's control algorithms.

With this architecture, GridHub can be easily modified and extended for future use. For example, we can sort a cluster of boxes in a GridHub, while another cluster of boxes is retrieved. We can also hold the transfer task assignment for working boxes to improve transfer efficiency.

7.2.3 Moving non-unit-sized boxes

We also develop a tree-like structure to organize conveyor modules that hold a single box and provide methods to share information, pass messages, and manage box movements. In order to pass messages and manage box movements, we create methods of storing and comparing historical messages. Modules can use these methods to track details of negotiation and make precise decisions.

7.3 Potential applications

GridHub provides a framework for organizing identical modules and managing their physical movements in a grid-layout material handling system. The architecture and control algorithms of GridHub are described on an abstract level. To communicate, every module buffers and processes the messages it receives sequentially. Hence, GridHub makes proper decisions when processing messages in different sequences. This framework can be easily adapted for different material handling scenarios.

For instance, this framework can be adapted for a storage or sorting system in a warehouse. Additionally, GridHubs can be stacked to build a multi-level storage system. The NU GridHub is also an ideal solution for Physical Internet Hubs. We consider every conveyor module a space for mobile objects, such as AGVs, so this framework can manage their movements.

7.4 Limitations and future extension

7.4.1 Limitations

Limitations on a hardware-level should be noted. First, methods for synchronizing conveyor modules have to be developed. In theory, this goal has been met, but more work is required at a hardware-level. Additional hardware-level development is necessary for communication methods among modules, for instance, a “cloud” or “peer to peer” method. Third, a module’s negotiation requires message passing methods at a hardware-level.

Livelock also requires additional research. We can prove that GridHub is livelock free in certain cases, but more work is needed to understand how to avoid livelock in the general case. For the same reason, it is difficult to obtain analytical results of the box transferring process. Hence, we need different approaches to accomplish these goals.

7.4.2 Future extensions

Every research topic has endless questions. Though this work has made the above contributions, some extensions may be possible. One of the future extensions could be the job of converting working boxes to active boxes, such as when to convert a working box to an active box. Because we have shown the number of active boxes can affect system performance, limiting the conveyor modules to the assignment of transfer tasks could control the number of active boxes in the system. Thus, the system performance could be changed.

AnyLogic does not implement modern day computing power, and it only allows users to run simulations on its own platform. Thus, we need faster simulation methods and cross-platfrom libraries to simulate a multi-agent system, like GridHub.

Since we have only considered rectangular grids, the negotiation methods could also be extended to cases where a grid has arbitrary shape. For example, in warehouses, storage systems have to be built around pillars, so “holes” may exist in a grid. Negotiation methods have to be developed to find empty conveyor modules in the grids, in this case.

REFERENCES

Agile System Inc. Naval stowage and retrieval system, July 2017. URL `http://www.agilesystems.com/navstors_frame.htm`.

Eric Ballot, Benoit Montreuil, and Collin Thivierge. *Progress in Material Handling Research 2012*. MHIA, Charlote, NC, U.S.A., 2012.

K. Boutoustous, G. J. Laurent, E. Dedu, L. Matignon, J. Bourgeois, and N. Le Fort-Piat. Distributed control architecture for smart surfaces. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2018–2024, Oct 2010. doi: 10.1109/IROS.2010.5650668.

Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer, 2008. ISBN 978-1-4419-4119-0.

CLARK Associate MH. Inc. Carton flow, July 2017. URL `http://www.clarkmh.com/shelving/carton-flow.php`.

E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3:67–78, 1971.

Enrico Colla and Paul Lapoule. E-commerce: exploring the critical success factors. *International Journal of Retail and Distribution Management*, 40(11):842–864, 2012.

Colling Dominik, Seibold Zäzilia, and Furmans Kai. Gridsorter – decentralized controlled material handling system for the transport of goods of different sizes. In *Logistics Journal : Proceedings*, volume 2016, 2016.

M. B. Firvida, H. Thamer, C. Uriarte, and M. Freitag. Decentralized omnidirectional route planning and reservation for highly flexible material flow systems with small-scaled conveyor modules. In *2018 IEEE 23rd International*

Conference on Emerging Technologies and Factory Automation (ETFA), volume 1, pages 685–692, Sep. 2018. doi: 10.1109/ETFA.2018.8502655.

FRED. E-commerce retail sales as a percent of total sales, July 2019. URL `https://fred.stlouisfed.org/series/ECOMPCTSA`.

T. Fukuda, I. Takagawa, and K. Sekiyama. Autonomous recovery of global efficiency based local interaction for flexible transfer system (fts). In *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113)*, volume 2, pages 879–884, Oct 2000. doi: 10.1109/IROS.2000.893130.

Kai Furmans, Kevin R. Gue, and Zäzilia Seibold. *Optimization of Failure Behavior of a Decentralized High-Density 2D Storage System*, pages 415–425. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-35966-8. doi: 10.1007/978-3-642-35966-8_35. URL `https://doi.org/10.1007/978-3-642-35966-8_35`.

L. Gravano, G. D. Pifarre, P. E. Berman, and J. L. C. Sanz. Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1233–1251, Dec 1994. ISSN 1045-9219. doi: 10.1109/71.334898.

Kevin Gue. New paper: Gridstore—a grid-based storage and retrieval system, January 2014. URL `https://kevingue.wordpress.com/2014/01/21/new-paper-gridstore-a-grid-based-storage-and-retrieval-system/`.

Kevin Gue, Kai Furmans, and Onur Uludağ. A high-density system for carton sequencing. In *Proceedings of the 6th International BVL Symposium on Logistis*, Hamburg, Germany, 2012.

Kevin R. Gue and Kai Furmans. Decentralized control in a grid-based storage system. In *Proceedings of the 2011 Industrial Engineering Research Conference*, 2011.

Kevin R. Gue and Byung Soo Kim. Puzzle-based storage systems. *Naval Research Logistics*, 54(5):556–567, 2007.

Kevin R. Gue, Kai Furmans, Zäzilia Seibold, and Onur Uludağ. Gridstore: A puzzle-based storage system with decentralized control. *IEEE Transactions on Automation Science and Engineering*, 11(2):429–438, April 2014. ISSN 1545-5955. doi: 10.1109/TASE.2013.2278252.

Katsumi Hama, Sadayoshi Mikami, Keiji Suzuki, and Yukinori Kakazu. Motion coordination algorithm for distributed agents in the cellular warehouse problem. *Artif Life Robotics*, 6:3–10, 2002.

Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1): 7–38, January 1998. ISSN 1387-2532. doi: 10.1023/A:1010090405266. URL `http://dx.doi.org/10.1023/A:1010090405266`.

Venkat R. Kota, Don Taylor, and Kevin Gue. Retrieval time performance in puzzle-based storage systems. *Journal of Manufacturing Technology Management*, 26:582–602, 2015.

Tobias Krühn, Sascha Falkenberg, and Ludger Overmeyer. Decentralized control for small-scaled conveyor modules with cellular automata. In *2010 IEEE International Conference on Automation and Logistics*, pages 237–242, Aug 2010. doi: 10.1109/ICAL.2010.5585288.

Tobias Krühn, Mišel Radosavac, Nikita Shchekutin, and Ludger Overmeyer. Decentralized and dynamic routing for a cognitive conveyor. In *2013 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, WOLLONGONG, Australia, July 2013.

Heiner Lasi, Hans-Georg Kemper, Peter Fettke, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business and Information Systems Engineering*, 6: 239–242, 2014.

Prasad S. Mahajan and Ricki G. Ingalls. Evaluation of methods used to detect warm-up period i steady state simulation. In *Proceedings of the 2004 Winter Simulation Conference*, December 2004.

Massey Rack. Pallet racks, July 2017. URL `http://masseyrack.com/2012/06/pallet-racking/`.

Stephan H. Mayer. *Development of a completely decentralized control system for modular continuous conveyors.* PhD thesis, Institutes für Fordertechnik und Logistiksysteme der Universitat Karlsruhe (TH), Karlsruhe, Baden-Wurttemberg, Germany, April 2009.

Russell D. Meller, Benoit Montreuil, Collin Thivierge, and Zachary Montreuil. Functional design of physical internet facilities: A road-based transit center. Technical Report FSA-2013-001, CIRRELT, Montreal, Canada, March 2013.

mezzaninestoragesystems101. The advantages and disadvantages of pallet shelves, mezzanines and asrs, August 2014. URL `https://mezzaninestoragesystems101.wordpress.com/tag/mezzanine-storage/`.

Masoud Mirzaei, Nima Zaerpour, and Rene de Koster. Modelling load retrievals in puzzle-based storage systems. In *17th CEMS Saint-Louis Workshop on Logistics and Supply Chain Management*, Saint-Louis, United States, December 2014.

Averill M.Law. *Simulation Modeling and Analysis*, pages 502–505. Mc Graw Hill, 2015. ISBN 9780073401324.

Benoit Montreuil. Toward a physical internet: meeting the global logistics sustainability grand challenge. *Logist. Res.*, 3:71–87, 2011.

Ludger Overmeyer, Kai Ventz, Sascha Falkenberg, and Tobias Krühn. Interfaced multidirectional small-scaled modules for intralogistics operations. *Logistics Research*, 2(3):123–133, Decemeber 2010. ISSN 1865-0368. doi: 10.1007/s12159-010-0038-1. URL `https://doi.org/10.1007/s12159-010-0038-1`.

T. Sakao, S. Kondoh, Y. Umeda, and T. Tomiyama. The development of a cellular automatic warehouse. In *Intelligent Robots and Systems '96, IROS 96, Proceedings of the 1996 IEEE/RSJ International Conference on*, volume 1, pages 324–331 vol.1, Nov 1996. doi: 10.1109/IROS.1996.570695.

Riccardo Scattolini. Architectures for distributed and hierarchical model predictive control – a review. *Journal of Process Control*, 19(5):723 – 731, 2009. ISSN 0959-1524. doi: https://doi.org/10.1016/j.jprocont.2009.02.003. URL `http://www.sciencedirect.com/science/article/pii/S0959152409000353`.

Melanie Schwab. *A decentralized control strategy for high density material flow systems with automated guided vehicles.* PhD thesis, Karlsruher Institute of Technologie (KIT), Karlsruhe, Baden-Wurttemberg, Germany, April 2015.

Zäzilia Seibold. *Logical Time in Decentralized Controlled Material Handling Systems.* Unpublished doctoral dissertation, Karlsruher Instituts für Technologie (KIT), Karlsruhe, Baden-Wurttemberg, Germany, Feb 2015.

Zäzilia Seibold, Thomas Stoll, and Kai Furmans. Layout-optimized sorting of goods with decentralized controlled conveying modules. In *2013 IEEE International Systems Conference (SysCon)*, pages 628–633, April 2013. doi: 10.1109/SysCon.2013.6549948.

Zvi Shiller. Off-line and on-line trajectory planning, 02 2015.

Ehsan Shirazi. Flexible high-density puzzle storage systems. Master's thesis, West Virginia University, Morgantown, West Virginia, June 2018.

Peerapol Sittivijan. *Modular Warehouse Control: Simultaneous Rectilinear Movement of Multiple Objects within a Limited Free Space Environment.* PhD thesis, North Carolina State University, Raleigh, North Carolina, May 2015.

I. Takagawa, K. Sekiyama, and T. Fukuda. Coevolution of physical configuration and control strategy on self-reconfigurable flexible transfer system. In *Proceedings*

2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453), volume 3, pages 2474–2479, Oct 2003. doi: 10.1109/IROS.2003.1249241.

Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 2015. ISBN 13:978-1-292-06142-9.

Sam Toueg. Deadlock- and livelock-free packet switching networks. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, pages 94–99, New York, NY, USA, 1980. ACM. ISBN 0-89791-017-6. doi: 10.1145/800141.804656. URL `http://doi.acm.org/10.1145/800141.804656`.

Onur Uludağ. Physical system, May 2012. URL `https://onuruludag.wordpress.com/2012/05/01/physical-system/`.

Onur Uludağ. *GridPick: A High Density Puzzle Based Order Picking System with Decentralized Control*. PhD thesis, Auburn University, Auburn, Alabama, May 2014.

Vanderlande. Quickstore miniload, July 2017. URL `https://www.vanderlande.com/warehousing/innovative-systems/storage-asrs/quickstore-miniload`.

Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(2):115–152, 1995.

Altan Yalcin, Achim Koberstein, and Kai-Oliver Schocke. An optimal and a heuristic algorithm for the single-item retrieval problem in puzzle-based storage systems with multiple escorts. *International Journal of Production Research*, 57 (1):143–165, 2019. doi: 10.1080/00207543.2018.1461952. URL `https://doi.org/10.1080/00207543.2018.1461952`.

Nima Zaerpour, Yugang Yu, and Rene de Koster. Optimal configuration in a puzzle-based compact storage system. In *11th TRAIL Congress*, Deflt, Netherlands, November 2010.

Nima Zaerpour, Yugang Yu, and Rene de Koster. Storing fresh produce for fast retrieval in an automated compact cross-dock system. *Production and Operation Management*, 24(8):1266–1284, 2015.

Appendix: Additional flowcharts of processing negotiation messages

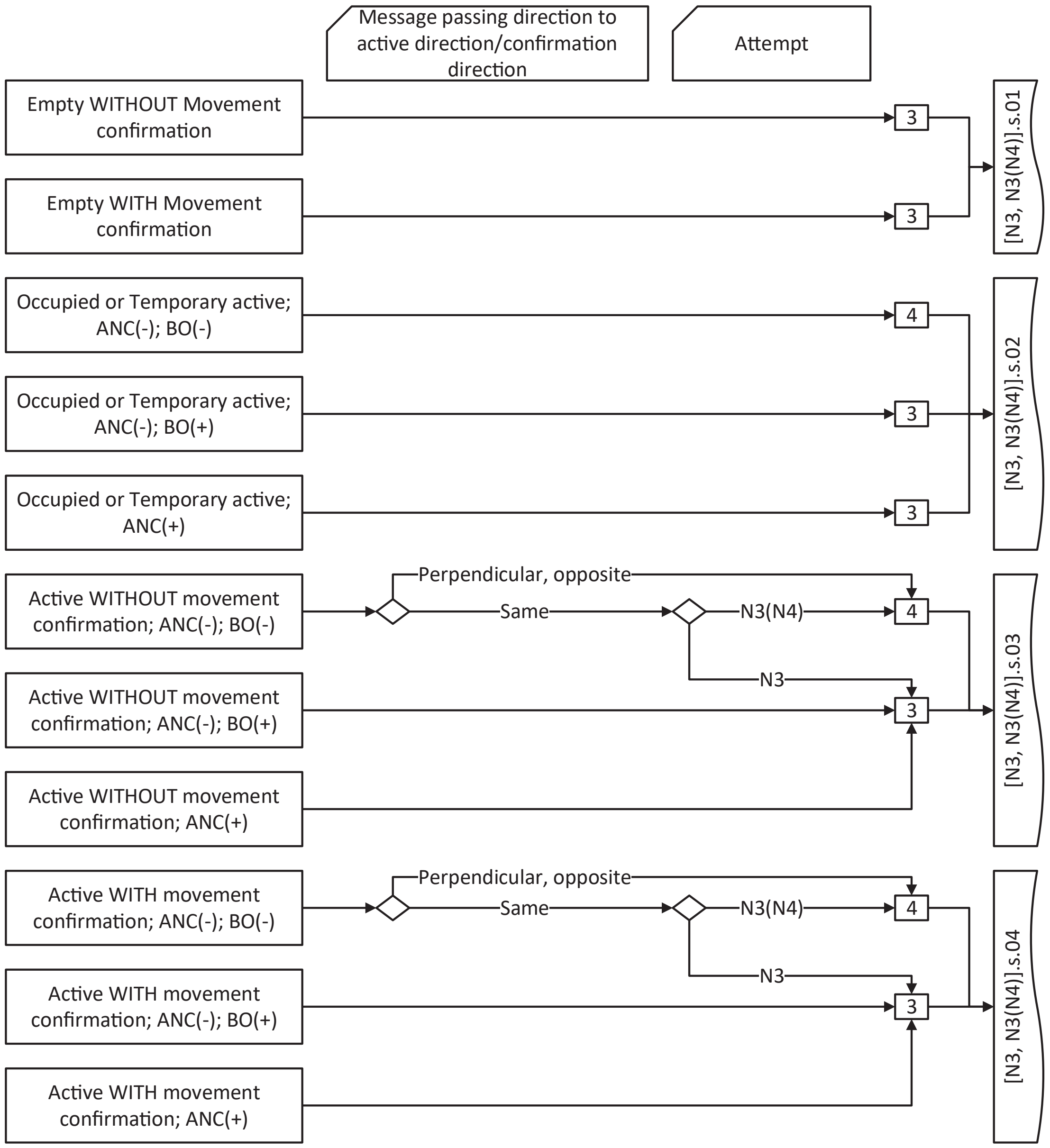

Message passing direction to active direction/confirmation direction
Attempt
Empty WITHOUT Movement confirmation
Empty WITH Movement confirmation
Occupied or Temporary active; ANC(-); BO(-)
Occupied or Temporary active; ANC(-); BO(+)
Occupied or Temporary active; ANC(+)
Active WITHOUT movement confirmation; ANC(-); BO(-)
Active WITHOUT movement confirmation; ANC(-); BO(+)
Active WITHOUT movement confirmation; ANC(+)
Active WITH movement confirmation; ANC(-); BO(-)
Active WITH movement confirmation; ANC(-); BO(+)
Active WITH movement confirmation; ANC(+)
Perpendicular, opposite
Same
N3(N4)
N3
3
4
[N3, N3(N4)].s.01
[N3, N3(N4)].s.02
[N3, N3(N4)].s.03
[N3, N3(N4)].s.04

Figure 147: Rules of processing N_3 and $N_3(N_4)$'s seek messages.

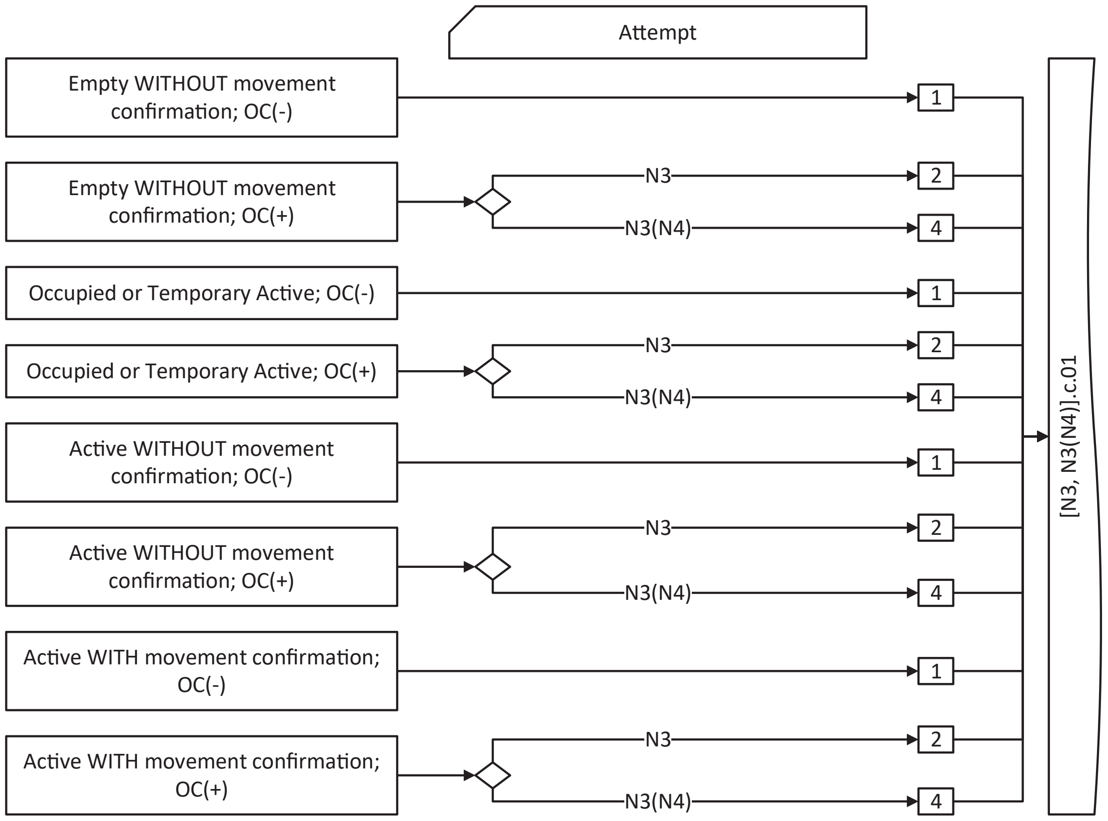

Attempt
Empty WITHOUT movement confirmation; OC(-)
Empty WITHOUT movement confirmation; OC(+)
Occupied or Temporary Active; OC(-)
Occupied or Temporary Active; OC(+)
Active WITHOUT movement confirmation; OC(-)
Active WITHOUT movement confirmation; OC(+)
Active WITH movement confirmation; OC(-)
Active WITH movement confirmation; OC(+)
N3
N3(N4)
1
2
4
[N3, N3(N4)].c.01

Figure 148: Rules of processing N_3 and $N_3(N_4)$'s confirm messages.

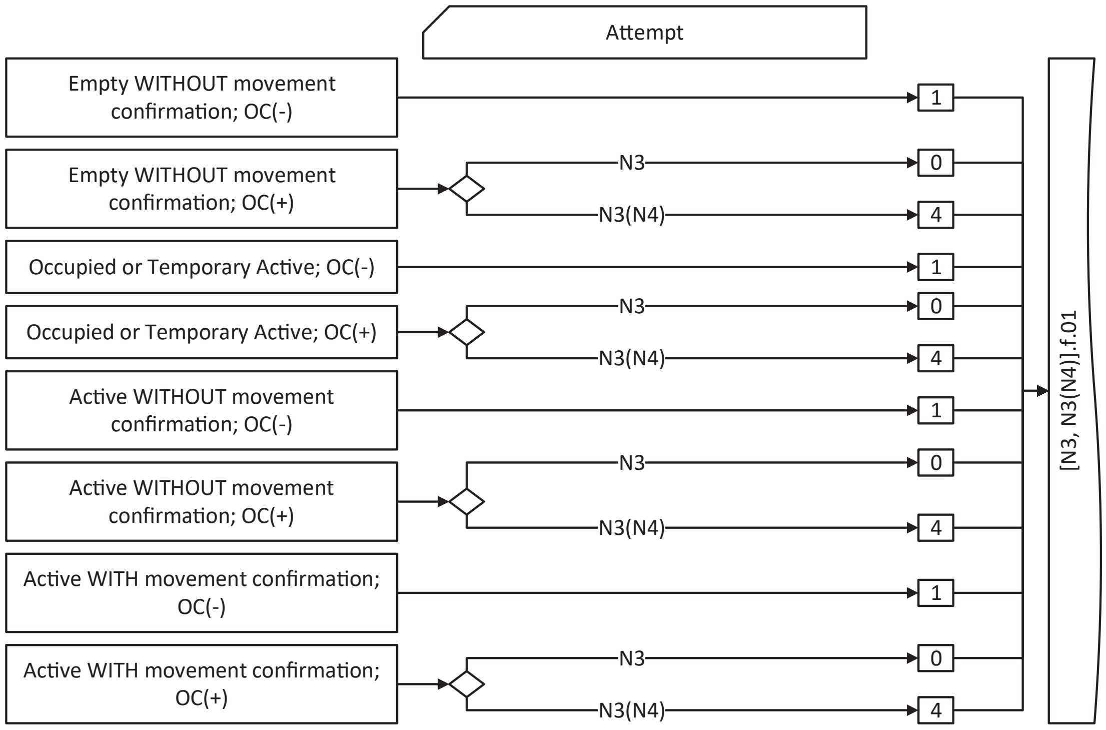

Attempt
Empty WITHOUT movement confirmation; OC(-)
Empty WITHOUT movement confirmation; OC(+)
Occupied or Temporary Active; OC(-)
Occupied or Temporary Active; OC(+)
Active WITHOUT movement confirmation; OC(-)
Active WITHOUT movement confirmation; OC(+)
Active WITH movement confirmation; OC(-)
Active WITH movement confirmation; OC(+)
N3
N3(N4)
1
0
4
[N3, N3(N4)].f.01

Figure 149: Rules of processing N_3 and $N_3(N_4)$'s fail messages.

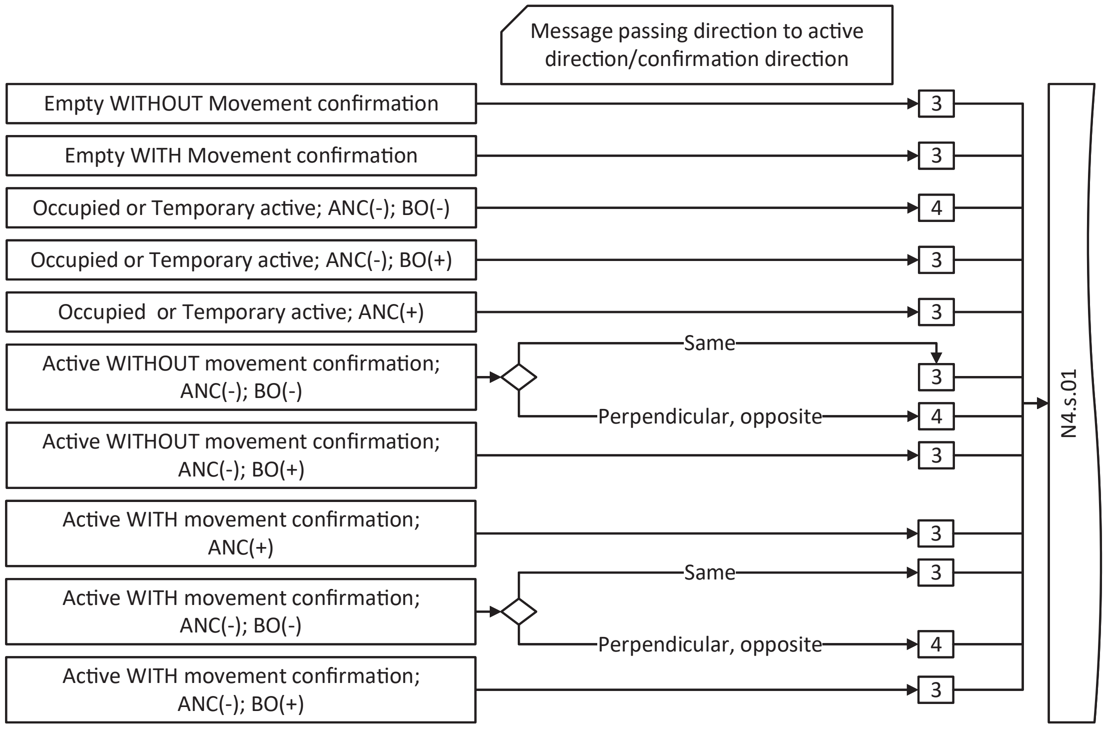

Message passing direction to active direction/confirmation direction
Empty WITHOUT Movement confirmation
Empty WITH Movement confirmation
Occupied or Temporary active; ANC(-); BO(-)
Occupied or Temporary active; ANC(-); BO(+)
Occupied or Temporary active; ANC(+)
Active WITHOUT movement confirmation; ANC(-); BO(-)
Active WITHOUT movement confirmation; ANC(-); BO(+)
Active WITH movement confirmation; ANC(+)
Active WITH movement confirmation; ANC(-); BO(-)
Active WITH movement confirmation; ANC(-); BO(+)
Same
Perpendicular, opposite
3
4
N4.s.01

Figure 150: Rules of processing of N_4's seek messages.

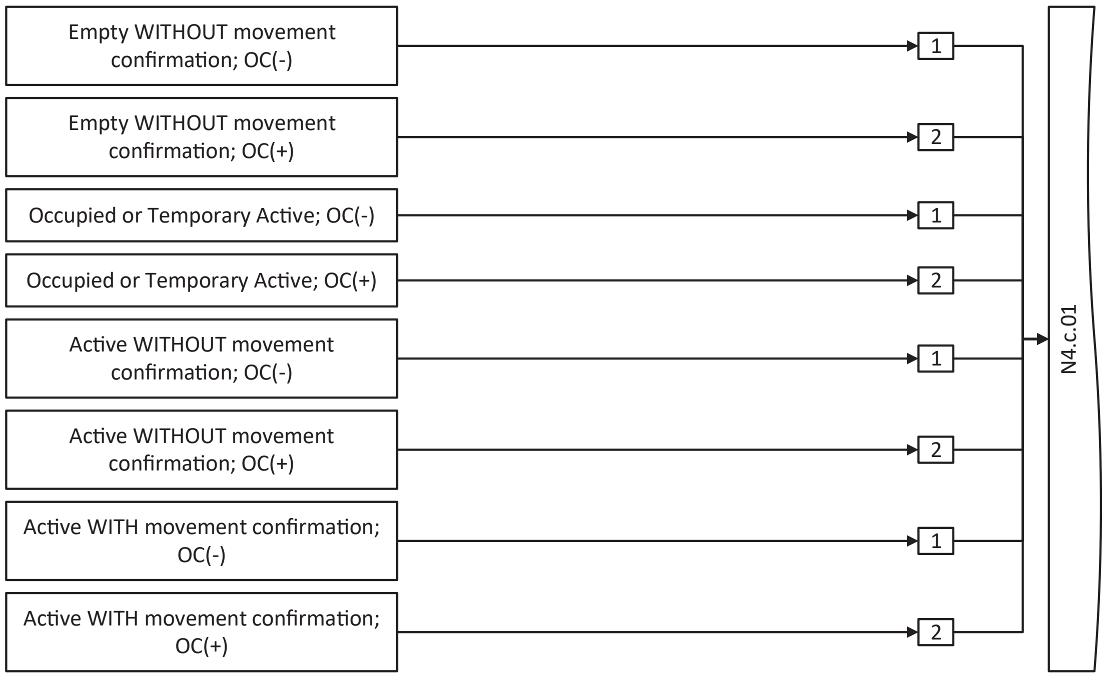

Empty WITHOUT movement confirmation; OC(-)
Empty WITHOUT movement confirmation; OC(+)
Occupied or Temporary Active; OC(-)
Occupied or Temporary Active; OC(+)
Active WITHOUT movement confirmation; OC(-)
Active WITHOUT movement confirmation; OC(+)
Active WITH movement confirmation; OC(-)
Active WITH movement confirmation; OC(+)
1
2
1
2
1
2
1
2
N4.c.01

Figure 151: Rules of processing N_4's confirm messages.

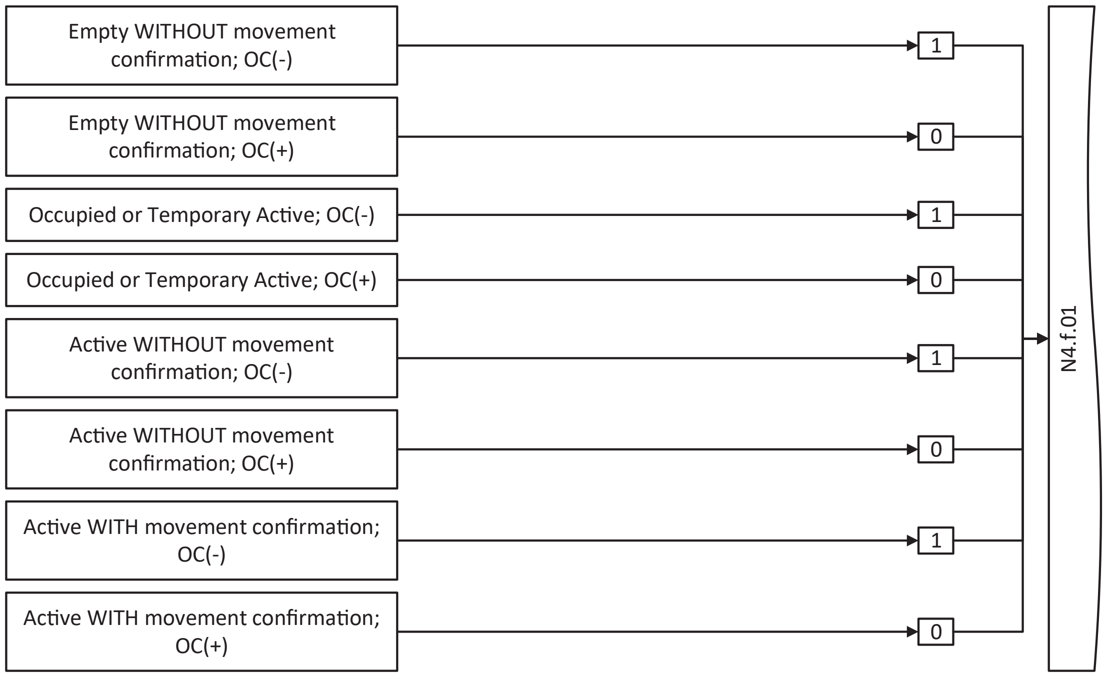

Empty WITHOUT movement confirmation; OC(-)
Empty WITHOUT movement confirmation; OC(+)
Occupied or Temporary Active; OC(-)
Occupied or Temporary Active; OC(+)
Active WITHOUT movement confirmation; OC(-)
Active WITHOUT movement confirmation; OC(+)
Active WITH movement confirmation; OC(-)
Active WITH movement confirmation; OC(+)
1
0
1
0
1
0
1
0
N4.f.01

Figure 152: Rules of processing N_4's fail messages.

Appendix: Results of ANOVA

The TukeyHSD test result does not include the interactions among different factors and levels.

```
Tukey multiple comparisons of means
95% family-wise confidence level

Fit: aov(formula = SystemThroughput ~ AspectRatio
* ExpectPath * x_wk, data = clean_final)

AspectRatio
     diff        lwr         upr            p adj
2-1  0.07602894  0.07478529  0.07727258     0
3-1  0.21497176  0.21372812  0.21621540     0
4-1  0.10304398  0.10180034  0.10428763     0
5-1  0.02938241  0.02813876  0.03062605     0
3-2  0.13894282  0.13769918  0.14018647     0
4-2  0.02701505  0.02577140  0.02825869     0
5-2 -0.04664653 -0.04789017 -0.04540288     0
4-3 -0.11192778 -0.11317142 -0.11068413     0
5-3 -0.18558935 -0.18683300 -0.18434571     0
5-4 -0.07366157 -0.07490522 -0.07241793     0

ExpectPath
     diff         lwr           upr          p adj
2-1  0.016766852  0.0157192874  1.781442e-02 0.0000000
3-1  0.018224630  0.0171770651  1.927219e-02 0.0000000
4-1 -0.001080926 -0.0021284904 -3.336145e-05 0.0400988
3-2  0.001457778  0.0004102133  2.505342e-03 0.0020010
4-2 -0.017847778 -0.0188953423 -1.680021e-02 0.0000000
4-3 -0.019305556 -0.0203531200 -1.825799e-02 0.0000000

x_wk
    diff       lwr        upr            p adj
2-1 0.16125514 0.16042755 0.16208273     0
3-1 0.25086306 0.25003546 0.25169065     0
3-2 0.08960792 0.08878032 0.09043551     0
```

CURRICULUM VITAE

Gang Hao

Department of Industrial Engineering

University of Louisville, Louisville, KY 40292

M.S., University of Nebraska-Lincoln, 2011

B.Eng., Nanjing Forestry University, 2008