Design, verification and analysis of the Highly Dynamic Storage system

by
D.F.J. Berendse

February 5, 2010

Contents

Preface

Over the years systems kept on growing and their complexity became more complicated then ever before. Failure or unwanted behavior of such systems is an increasing problem as it becomes more difficult to create a flawless system, or at the very least a system in which the most critical parts (such as safety requirements) are guaranteed to hold under all conditions. The need for formal analysis of these system becomes evident: pure intuition and experience are no longer sufficient. One way of analyzing the behavior of a system is by means of process theory which describes a system in terms of processes. Such a specification can be used to prove desired properties by several methods, such as model inspection of the part that is of interest or by automated verification. This thesis details the process of specifying an efficient storage device of which desired properties are proven and of which performance is measured and analyzed. Both aspects are investigated with the use of the mCRL2 toolset, which presents us with the interesting result that simulations can be performed on a proven system.

Acknowledgments

First and foremost, I would like to thank my supervisors Jan Friso Groote and Roelof Hamberg for their support and help in this graduation project. We regularly discussed the current results of the project and reflected upon the status of the work, which really aided me in this project. I would also like to thank Rink Springer for his constructive criticism and discussions while working on this project, as well as everyone else who read my thesis and provided useful comments. Last but not least, I would like to thank the mCRL2 development team that quickly helped me with mCRL2 issues that I encountered during this project.

Eindhoven, January 2010
Dwight Berendse

Abstract

History shows that over the years software became more difficult to truly and completely understand, as software becomes more complex due to the increasing demands and expectation on software. This complexity of software presents itself in a Programmable Logic Controller (PLC) that controls a storage device which shows undesired behavior. This undesired behavior includes inefficient functioning and unexpected system functioning. The software for the controller is redesigned as a mCRL2 model with a modular approach that divides the tasks of the system into several processes. Safety and behavioral properties, expressed as modal formulas in the μ-calculus language, are verified on the model to prove that the system works as intended under all possible conditions that it can reside in. An approach is presented that allows us to write modal formulas that respect priority bound actions that are used in the controllers model. Such formulas tend to become large expressions but in practice their validity is calculated quite efficiently.

Next to the verification of the systems properties we also investigate the performance of the system for which we introduced an *M/D/1* and an *M/D/2* system. The performance of these systems are verified against analytical approximations. The *M/D/2* model is used as the core of the systems model that we are investigating in this research: the Highly Dynamic Storage system. The results of these simulations include the average process time of an object by the system, the average queue length of objects waiting to be processed by the system and its throughput.

In this research we have shown that it is possible to perform simulations with an mCRL2 model and also verify properties of the system that hold for all situations of the system on almost the same model. Some adaptation to the mCRL2 model was necessary to verify the model, as otherwise too much computer memory would be needed.

Chapter 1

Introduction

Vanderlande Industries, *VI* for short, is a large organization that offers a wide range of technologies for customized parcel handling and documents, baggage handling systems for airports of all sizes and automated logistics system for warehouses and distribution centers. The Embedded System Institute, *ESI*, cooperates with VI in projects that are aimed at the development of existing and new systems that are often aimed at efficiency improvement. A number of these projects are distributed by the ESI to students at the University of Technology Eindhoven; one of these project is described in this thesis as a final graduation project.

The Highly Dynamic Storage system is a VI project that is aimed at increasing the amount of stored items per m^2 of surface by stacking them above each other. This means that the items must be lifted to the correct height which takes time and as such influences the throughput of these items. The Highly Dynamic Storage system, or *HDS* for short, is managed by a PLC industrial PC of which the software development is outsourced to a third party. During live tests at the VI test facilities they noticed that the PLC code was below their quality standard in terms of strategy, where the HDS system often showed unexpected behavior that reduced the throughput performance. The actual specification and implementation are not released, but VI learned from on the spot debugging session (in cooperation with the third party) that the code seemed created in a ad-hoc way with very little structure overall. Furthermore it is not guaranteed that such ad-hoc patches do not introduce new problems in other parts of the system. This approach produces PLC source code that becomes unmaintainable and hard to understand because of the single problem fixes that are introduced on top of the ad-hoc design. VI wants a machine controller that is modular in its design and where system properties are proven to be true under all possible conditions. Furthermore they would like to know the performance of the system in terms of throughput and the time it takes to process totes.

VI has described an algorithm that specifies the desired properties of the system on the methods of handling items with the HDS system. This algorithm does not represent the actual implementation of the HDS controller; actually it is unknown if it does. The HDS hardware subsystems are described in Chapter 2 followed by its algorithms, as defined by VI. The safety of these hardware systems must be guaranteed in such a way that the code never performs actions that result in physical damage being done to system; for example: totes should never never be placed on locations where a tote already resides. The properties that provide such safety demands, in terms of damage to the system, are described in chapter 3. The properties that validate the behavior of the HDS systems model is included as well.

To perform simulations with the mCRL2 model of the HDS system, we first designed two relatively simple and easy to understand systems that have a close resemblance with the HDS system. With these systems, we tested our method to perform a simulation of a mCRL2 model by validating the correctness of the simulation results. This validation is done by comparing the results against analytical simulation approximations. Chapter 4 handles both the design and verification of the simulation approach.

The model of the HDS system itself is described in Chapter 5 which begins with a short introduction of the mCRL2 model language for those who are unfamiliar with it. Subsequently a model is

presented in which the simulation aspects are removed such that we can validate the properties that are stated in Chapter 3 on a model in which all possibilities are explored. Validating these properties against the simulation model verifies just one *possible* situation in which the HDS system could reside: exactly the issue that we want to overcome with toolsets such as mCRL2. To do so we created a translation from the simulation version to a version where time is replaced by priorities. This translation is explained in Section 5.3. At the end of the chapter we discuss the simulation results of HDS system and analyze the performance of the system.

The functional and behavior requirements from Chapter 3 first have to be expressed as modal formulas before we they can be automatically verified on the model of the HDS system. Chapter 6 gives a short description of the modal μ-calculus language (for those who are unfamiliar with it), followed by the modal formulas of the requirements. However the 'normal' way of writing modal formulas for a mCRL2 model is not sufficient in our case because we deal with priority bound actions which allows actions to have precedence over others. The translations from a 'normal' modal formula to a prioritized modal formula is provided in Section 6.3 and 6.4, where a simple example is provided and the deadlock freedom property is written out completely for the HDS systems model. All properties have been translated to the prioritized version and are verified with the mCRL2 toolset.

Chapter 2

Overview

This chapter is devoted to the description of the *Highly Dynamic Storage* system, *HDS* for short, in which the hardware of the system is explained. After the hardware description we discuss the strategies in place that control the behavior of the HDS system.

2.1 The Highly Dynamic Storage system

The *Highly Dynamic Storage system*, as globally depicted in Figure 2.1, is an automated high performance storage facility that is specialized in the storage and retrieval of totes to and from a large amount of storage locations. At one side totes arrive from an arbitrary external system and are stored at a storage location in one of the two large racks of the HDS system. The HDS system also handles requests to retrieve totes from the racks, which are delivered to an arbitrary external system. The physical interfaces connections to the external systems used at Vanderlande Industry are two conveyor belts, one serving as input and one as output of the HDS system. In the next sections we look at the different systems that constitute the HDS system and discuss their purpose.

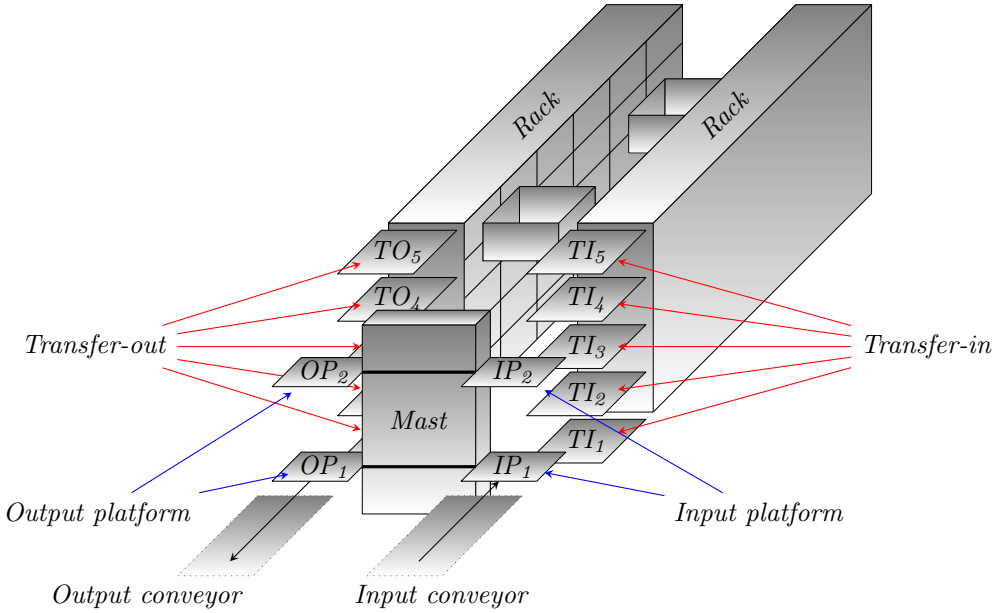


Figure 2.1: Highly Dynamic Storage system

2.1.1 Shuttle system

The HDS system has two racks which provide a large amount of storage locations where totes are stored by the system. Each rack is divided in a number of aisles stacked on top of each other. Each one of these aisles has a number of storage locations placed next to each other. Shuttles travel alongside these two racks to store and retrieve totes from these storage locations. Each shuttle is assigned to a number of aisles that it is allowed to utilize, spreading the workload of each shuttle over an equal number of aisles. The shuttles can store and retrieve totes from these aisles and can also retrieve and store them at different heights from transfer-points which make up the buffer system.

2.1.2 The buffer system

A buffer system is situated in front of the rack setup. This buffer system has two columns, each one with a number of temporary storage locations that are positioned above each other. We will refer to these locations as *transfer points*. There are two types of transfer points: *transfer-in* and *transfer-out* points. These are respectively denoted as TI_n and TO_n, for $1 \leq n \leq 5$; throughout this system we use $n = 5$, however any value satisfying $n \geq 3$ is allowed. An elevator system, which is discussed later in this section, places totes that need to be stored on the transfer-in points. A shuttle will complete the storage of the tote by retrieving it from the transfer-in point and placing it in one of the storage locations in the racks. Totes that have to be retrieved from the HDS system to the external output system are placed on the transfer-out points by the shuttles. Of course the shuttles will first get the requested tote from the storage location in the rack before delivering it to a transfer-out point. The elevator system takes totes from the transfer-out points to complete their retrieval.

2.1.3 The elevator system

The elevator system is situated in front of the buffer system and consists of a mast to which two platforms are connected. Each platform consists of two temporary storage locations: one location for totes going into the system and one for totes leaving the system. The ingoing locations of the platform are denoted as IP_1 and IP_2 and carry totes from the input conveyor to one of the transfer-in points: $TI_{1...5}$. The outgoing locations of the platform are denoted as OP_1 and OP_2 and carry totes from one of the transfer-out points $TO_{1...5}$ to the output conveyor. Note that these platforms can never pass each other since they are positioned above one another. This also implies that the lower platform can never reach the top transfer-points TO_5 and TI_5 and the upper platform can never reach the bottom transfer-points TO_1 and TI_1.

These platforms move alongside the mast and are both powered by a dedicated stepper motor. Each stepper motor has a controller which accepts commands that tell it where to move the platform to. The controller sends out a signal once the platform reaches the requested destination and does not accept new commands while it is still executing the previous command.

2.2 Performance strategy

2.2.1 Elevator system

At first sight the buffers in between the rack and elevator system seem quite superfluous, so why are they considered a fundamental part of the HDS system? It could be argued that the transfer of a tote from the shuttle directly to a platform greatly simplifies the system. However the main reason to opt for the buffer system is related to the difference of processing speed of the elevator system and the shuttle system. Shuttles would be kept waiting for a platform to arrive before the exchange of totes can take place because this system is faster than the elevator system. The introduction of the buffer system eliminates such waiting times and increases the overall performance of the HDS system.

The HDS system utilizes a strategy which aims to efficiently store and retrieve totes between the buffer and the elevator system. Both platforms apply this strategy to determine their next course of action. The strategy results in a total of four actions (see the next two paragraphs) that a platform can perform depending on its current location and the status of the buffer system. Note that moving to a transfer point or conveyor belt, including the exchange of a tote between two systems, is considered to be an *atomic action* that cannot be interrupted.

The two basic actions that a platform can perform are the transportation of a tote from a transfer-out point to the output conveyor, and the transportation of a tote from the input conveyor to a transfer-in point. From now on we will respectively refer to the actions as the *retrieval-action* and the *storage-action*. The strategy will select one of these actions if totes are waiting at transfer-out points or if totes are waiting at the input conveyor. It also verifies that a free transfer-in point is available before the storage-action is selected, otherwise this action is not taken. Note that this strategy implies that totes can always be moved to the output conveyor; it is never unavailable.

The last two actions of the four are combinations of the retrieval-action and storage-action. These two actions are possible under the condition that both retrieval-action and storage-action are possible at the same time. First the strategy will check if the storage tote can be transported to the transfer-in point that is positioned next to the transfer-out point where tote retrieval takes place (e.g: TO_4 and TI_4). If so then it will get the tote from the input conveyor and travel to the transfer-points where it will simultaneously transfer the two totes: one tote going from the platform to the transfer-in point, and one tote going from the transfer-out point to the platform (e.g: IP_2 to TI_4 and TO_4 to OP_2). After these transfers are completed the platform will deliver the retrieved tote to the output conveyor. However if the case arises that the transfer-in point next to the transfer-out point is not available then the strategy will select another free transfer-in point. It will then execute the storage-action first, directly followed by the retrieval-action (sequentially). If none of the transfer-in points is available then the strategy will fall back to the retrieval-action. We will refer to these two actions as the *simultaneous-action* and the *sequential-action*.

2.2.2 Shuttle system

Even though the handling strategy of the shuttle system is unknown, we can still describe several properties based on the observations made during the live demo of the HDS system at Vanderlande Industries. Each shuttle can transfer exactly one tote at a time which travels between some designated storage location in the rack to one of the transfer points. The retrieval requests are most likely processed in a FIFO order, for this the shuttle will first get the tote from the storage location in the rack, after which it delivers it to one of the transfer-out points: $TO_{1...5}$. Totes from the transfer-in points $TI_{1...5}$ are also most likely picked up in FIFO order after which they are stored on a storage location in one of the racks. During our analysis we do not rely on this FIFO property but assume that any sequence is possible.

However we have also seen that the shuttles try to optimize this process by putting retrieval totes at a transfer-out point at which a transfer-in tote is also waiting. This way it can immediately process the storage tote after delivering the retrieval tote without traveling to another transfer-in point. Thus we focus our research on the elevator system in such an abstract way, that the shuttle system's strategies are unimportant.

2.3 Tote selection

The strategy from the previous section determines the next course of action for a platform based on the availability of totes. In this strategy we mentioned that a tote is available at some transfer-out point or at the input conveyor. In this section we will see how tote availability is determined for both storage and retrieval totes.

2.3.1 Storage totes

Totes that have to be stored in the HDS system are delivered per conveyor belt. A number of totes can be queued on this conveyor in case a burst of tote arrivals occur, meaning that the tote arrival rate is momentarily beyond the process capacity of the elevator system. The conveyor belt pushes the first tote in the queue into the elevator system which triggers a sensor that notifies the HDS system of the new tote that has just entered the system. The tote is then scanned such that the HDS system can decide at which storage location in the rack the tote will be stored to provide an equal division over the aisles. It does this in cooperation with an external database, referred to as a *logic business backend*, that can dictate specific requirements for totes with specific contents. For example dangerous or heavy weighing content should be stored close to the ground for safety reasons. The HDS system has complete control over the selection of the storage location in the rack if the logic business backend dictates no restrictions. The sequence in which totes are delivered by conveyor is beyond the scope and control of the HDS system.

2.3.2 Retrieval totes

Retrieving a tote from the HDS system starts with an external request (by some external system) to retrieve a particular tote based on its identifier, which can be resolved to a storage location in the rack. A shuttle is then dispatched to retrieve this tote from the rack and place it on a transfer-out point of the buffer system. Totes are then processed in a certain order from these buffers, which is determined by an algorithm of VI. The lift platforms will eventually retrieve the totes from the buffers and transport them to the output conveyor belt.

2.4 The priority game

As the elevator system is processing orders one will find that a platform can be in the way of the other one, preventing it from doing its job. A *priority status* is introduced to the elevator system which results in one platform having precedence over the other. Whenever a platform intends to perform an atomic action (see Section 2.2.1) which would result in the two platforms crossing each other, then the platform with the priority status gets precedence. After completion of the atomic action it transfers the priority status to the other platform. If a platform has priority and the other platform is positioned somewhere in the path to its destination then the other platform moves to the closest possible location that clears the path for the platform with a priority status. The priority status is not taken into account if the offending platform is idle, meaning that it does not have any active orders.

Figure 2.2 shows an example where both platforms have an active order and the priority status is assigned to the lower platform. The upper platform's destination is level 2 and the lower platform's destination is level 4. The start situation is depicted in 2.2-a where the upper platform is blocking the path the level 4 for the other platform. Since the lower platform has precedence we will order the upper platform to move out of the way traveling as little distance as possible which is to level 5 just above level 4. The two platforms will travel to theses destinations which results in the situation depicted in 2.2-b after which the upper platform is assigned the priority status. Next we assume that this point the lower platform has processed all active orders and becomes idle. The upper platform now wants to continue which means that the lower platform has to the move to level 1, just below level 2. Both platform now travel to these locations which results in the situation in 2.2-c but it will not sign over the priority status since the lower platform was idle.

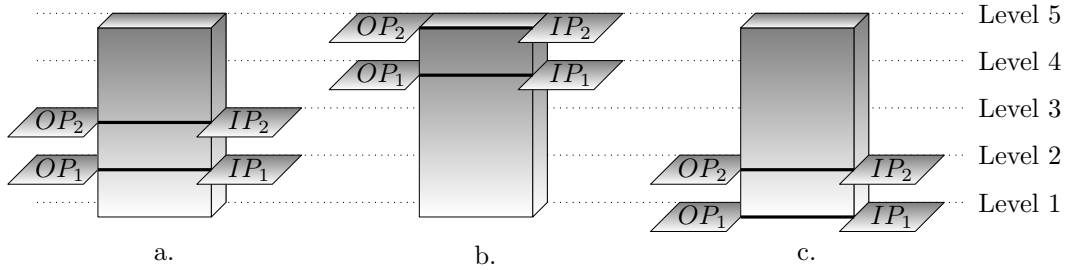


Figure 2.2: Lift priority system

Chapter 3

Requirements

The previous chapter specifies how the HDS system functions of which a mCRL2 model is created as is later on described in Chapter 5. This chapter illustrates the desired properties of the HDS system that we verify against the model of the HDS system. These properties make sure that the model of the system behaves correctly in terms of expected behavior and the safety of the hardware.

3.1 Performance goal

Efficiency is a key point of the HDS system besides meeting additional constraints (e.g. safety) as well. The efficiency of the HDS system is reflected by the performance requirements whereas the functional requirements reflect the behavioral boundaries of the system. The performance requirements are listed in table 3.1.

Table 3.1: Overview performance requirements

Identifier	Requirement
PR.1	Maximize the throughput of the HDS system which is measured in the total number of totes processed per hour.
PR.2	Minimize the average time required to process a single tote, which is measured in seconds.

The performance will be measured by incorporating time attributes into the mCRL2 model as will be explained in Chapter 4.

3.2 Functional requirements

The functional requirements can be divided into hardware safety demands on the one hand and behavioral demands on the other hand. The hardware demands guarantee that the two lift platforms never collide. Furthermore the system may never put totes on buffers or platforms when a tote still resides on such locations. These two demands imply that no physical damage is ever done to the system. The remaining properties of the system are described by the behavior demands, such as starting a motor to move a lift platform to a new destination when required. Table 3.2 lists all the functional requirements that we use to verify our mCRL2 model of the system.

These requirements of the HDS system are verified by checking if the mentioned properties hold on the entire statespace of the HDS system's model. To do so we have to translate the requirements into properties that are expressed using the set of actions that are possible in the mCRL2 model. If a sequence of these actions (which denote a given property) holds for the

Table 3.2: Overview functional requirements

Identifier	Requirement	Description
FR.1	The two lift platforms of the elevator system never collide against each other.	The controller only orders a lift platform to move to locations in which the other lift platform does not block the path or it will move the offending lift platform out of the way.
FR.2	Totes are always safely transported onto the lift platforms.	This requirement consists of two parts because a lift platform has two temporary storage locations, one for input and one for output. The controller firstly only orders a lift platform to retrieve a tote from the input conveyor if the lift platform input storage location is empty. Secondly the controller only orders a retrieval of a tote from a transfer-out point if the lift platform is output storage location empty.
FR.3	Totes are always safely transported onto the transfer points.	This requirement also consists of two parts because both the lift platforms and the shuttles are involved. Firstly, the controller only orders a lift platform to store a tote at an empty transfer-in point. Secondly, the controller only orders a lift platform to store a tote at an empty transfer-out point.
FR.4	The totes that arrive at the input conveyor are eventually retrieved by either lift platform.	After the sensor at the input conveyor signals that a tote has arrived then a lift platform will eventually retrieve this tote.
FR.5	If a motor processes a command then it will eventually signal that it has performed that command.	A motor is started to move a platform to a destination location. The motor will eventually signal that the lift platform has reached the destination location.
FR.6	Commands from the lift agent are eventually performed.	After a lift agent sends the next command to the mast then the lift agent will eventually be notified that the command is processed.
FR.7	Commands from the mast layer are eventually performed.	After the mast sends an command to the motion manager then it will eventually be notified that the command is processed.
FR.8	Commands from the motion manager are eventually performed.	After the motion manager commands a lift platform to perform the command then it will eventually be notified that the command is processed.

complete statespace of the system, then we may conclude that a property is valid for the model. Such a sequence of actions can be described with modal formulas. Modal calculus is explained in Chapter 6, in which also the properties of Table 3.2 are explained and described as modal formulas.

Chapter 4

Modeling performance

The functional requirements from Section 3 are used to validate the correct behavior of the HDS system controller, yet we also require a method to determine the performance of the very same model. The mCRL2 toolset does not offer direct means to determine the performance of a model. However this was expected as the mCRL2 language is not intended as a performance measure tool. Sections 4.2 and 4.3 will elaborate on the method that is developed to measure performance, which is realized by incorporating elapsed time into the model specification without changing the behavior of the system. This method will eventually be applied in the HDS system model, to evaluate the performance and properties of several process scheduling policies.

4.1 Server queuing system

The two elevator systems that are described in the two upcoming Section 4.2 and 4.3, lift totes that arrive at the system to a higher location. Totes arrive on a conveyor belt for both these systems but the system of Section 4.2 can handle one tote at a time whereas the system of Section 4.3 can handle two totes at once. These systems can be viewed as server queuing systems for which the performance can be approximated with the theory described in this section. Such a system can be denoted as a *M/D/c* model (Kendall's notation), which we compare against the models simulations to verify their correctness. The following approximation for the flowtime φ holds (see [6] pp. 332-347):

$$\varphi = \frac{1}{2}\left\{1 + (1-u)(c-1)\frac{\sqrt{4+5c}-2}{16uc}\right\}\frac{(uc)^c}{c!(1-u)^2}\left\{\sum_{k=0}^{c-1}\frac{(cu)^k}{k!} + \frac{(cu)^c}{c!(1-u)}\right\}^{-1}\frac{\delta}{c} + \frac{\delta}{2} \quad (4.1)$$

In this formula, φ is the average cycle time (flowtime), u is the utilization of the system, c is the number of parallel servers, and δ is the deterministic server time. In comparison to [6], one factor u in the first term has disappeared due to conversion of the arrival rate in the queue to the average server time. The second term is the time spent in the server itself. These terms are respectively denoted in Figure 4.1 as *E(Q)* (the time waiting in queue) and *E(S)* (the time spent in the server). In our case the latter is half of the server time δ: the time spent to transfer the tote from the input to the output. The second half of the server time is spent returning the platform to its initial input position, which does contribute to the cycle time of the totes, but is seen as part of the server time for all the totes in the waiting queue.

For the elevator system described in Section 4.2 with a single server, thus $c = 1$, formula 4.1 reduces to:

$$\begin{aligned}\varphi &= \frac{1}{2}\frac{u}{(1-u)^2}\left\{1 + \frac{u}{1-u}\right\}^{-1}\delta + \delta - \frac{\delta}{2} \\ &= \frac{1}{1-u}\left(1 - \frac{1}{2}u\right)\delta - \frac{1}{2}\delta \end{aligned} \quad (4.2)$$

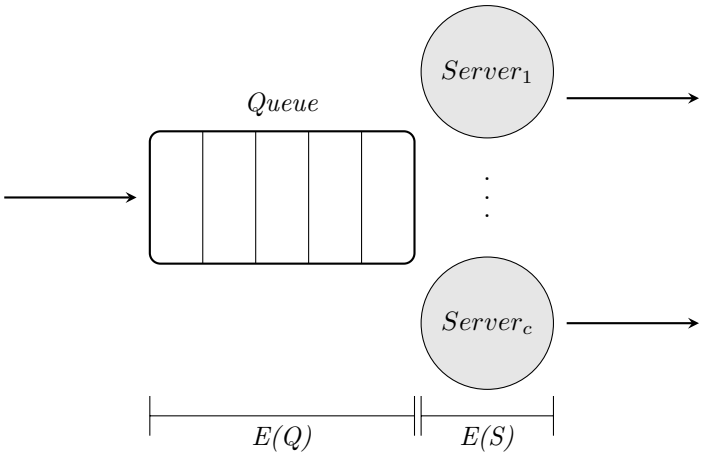


Figure 4.1: M/D/c model

For the elevator system described in Section 4.3 with two servers, thus $c = 2$, formula 4.1 reduces to:

$$\begin{aligned}
\varphi &= \frac{1}{2}\left\{1 + (1-u)\frac{\sqrt{14}-2}{32u}\right\}\frac{u^2}{(1-u)^2}\left\{1 + 2u + \frac{2u^2}{1-u}\right\}^{-1}\delta + \frac{1}{2}\delta \\
&= \frac{1}{2}\left\{1 + (1-u)\frac{\sqrt{14}-2}{32u}\right\}\frac{u^2}{(1+u)(1-u)}\delta + \frac{1}{2}\delta \\
&= \frac{1}{2}\left\{1 + (1-u)\frac{\sqrt{14}-2}{32u}\right\}\frac{u^2}{1-u^2}\delta + \frac{1}{2}\delta
\end{aligned} \tag{4.3}$$

The throughput of these systems is defined as the number of objects in the queue that receive service in a given period of time. For the throughput ψ the following formula holds:

$$\begin{aligned}
\psi &= \frac{1}{(\delta/c)}u \\
&= c\frac{u}{\delta}
\end{aligned} \tag{4.4}$$

In this formula ψ is the throughput and $\frac{\delta}{c}$ is the average service time. So doubling the number of servers doubles the throughput, which is according to expectation.

The work in progress (ξ) denotes the average number of totes that are in scope of the server system, i.e., either in the queue or in one of the c servers. This relation can be determined using Little's Law (see [6], pp. 262), for which the following formula holds:

$$\xi = \varphi \cdot \psi \tag{4.5}$$

The stochastic probability theory that is discussed in this section is used to validate the correctness of the models of the elevator systems, which are discussed in the next two sections.

4.2 Single-platform elevator system

The complexity of the HDS system controller makes it difficult to determine a method to keep track of time immediately. We first introduce a basic and easy to comprehend elevator system. This system will behave completely deterministically and refrains from using retrieval and storage strategies as those used by the HDS system. This simple system also allows verification of simulation results by comparing it to an analytical approximation.

4.2.1 Problem statement

Processing totes fast and efficiently is the main objective of the HDS system under the conditions that are stated in Table 3.1 as the requirements of the system. The performance of the live HDS system by Vanderlande is expressed in the number of totes that are processed by the system in a given time-unit, normally an hour. A second important factor is the average time that is spent before a single tote is handled. This is the total time between the tote entering the system until the moment it leaves the system. Finally we want to know the work in progress, the average number of totes in the system, to determine that the input queue does not grow too large.

The time that is needed to process the totes on the live HDS system is measured by a timer, keeping track of time as the system is physically running. However, computer simulations typically don't execute at the same speed as the physical system, because mechanical actions such as transportation are not actually performed. The time that is required for such actions must be extracted from the actual system. An example is the time it takes to move an object between two static locations, which needs to be taken into account in the simulation.

4.2.2 Approach

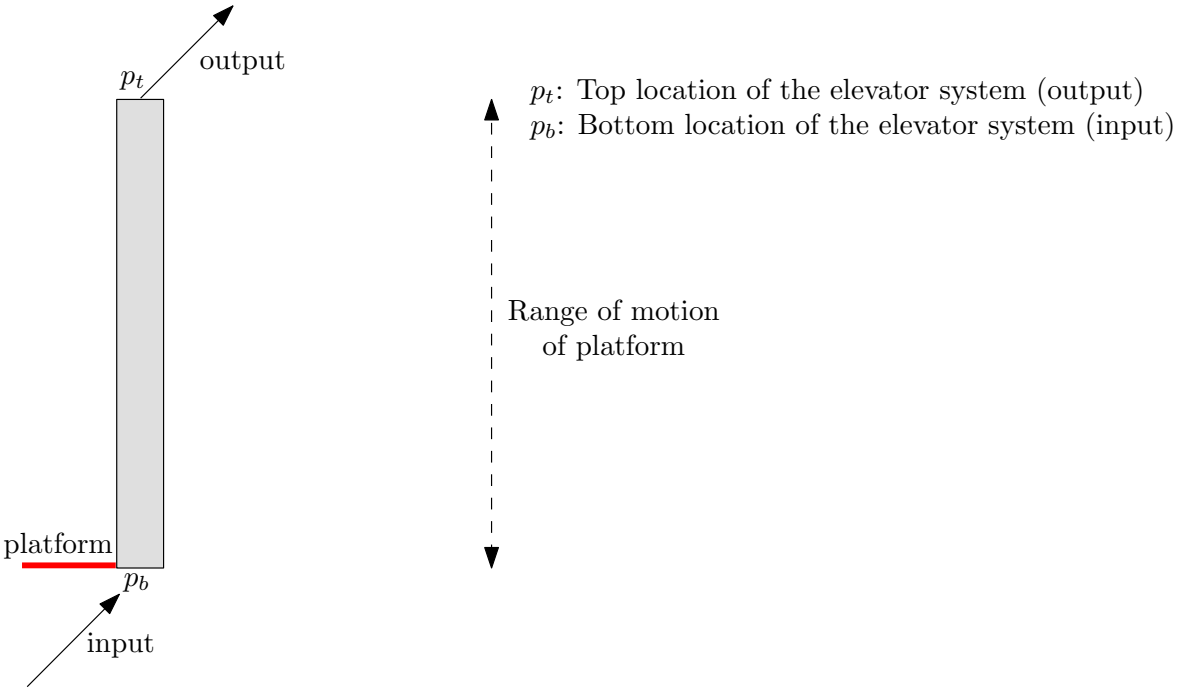


Figure 4.2: Single-platform elevator system

As discussed in Section 4.2 we first focus on a system consisting of one platform connected to a mast, depicted in Figure 4.2. We will refer to it as the *single-platform system* from now on. Totes arrive at the input queue at the bottom of the mast and are transported to the output queue at the top of the mast. The system does so by loading the first tote at the input queue onto the platform and transporting it to the output queue where it is then unloaded from the platform. Afterward it will travel back to the input queue and process the next tote if there is any available. Otherwise it will idle until the next tote enters the queue.

We assume that a stream of totes arrives at the input with inter-arrival times that follow a certain pattern: random, sporadic or any other imaginable pattern. We introduce a delivery process in our model, to keep track of the absolute arrival times of the totes. This process is addressed by the elevator once the platform reaches the bottom, as to determine when the next tote is ready. For this model we selected a Poisson process as arrival pattern, where totes arrive continuously and independently of the previous one. The arrival times between two totes for a Poisson process is expressed by an exponential distribution. Before each simulation of this system we generate an

exponential distribution (for a given mean arrival time) and inject this into the mCRL2 models delivery process. Through this approach the single-platform elevator system sometimes has to wait for totes to arrive and sometimes creates a queue at the input when the totes arrive at a higher rate than they can be processed.

The system has a certain amount of work to perform depending on the mean of the inter-arrival times, where it might be continuously active (processing totes) or it is idle for a certain amount of time (waiting for new totes to arrive). This workload is expressed as the utilization of the system. The utilization of a simulation can be approximated given a mean inter-arrival time of totes. Let δ_a be the mean inter-arrival time, u_{app} denote the utilization, c denote the number of servers and δ be the server time. Then the following holds:

$$u_{app} = \frac{1}{c} \cdot \frac{\delta}{\delta_a} \tag{4.6}$$

A utilization of 0 indicates that there is no work at all, whereas a utilization of 1 indicates that the system is continuously at work.

The single-platform system contains two factors that influence the determination of total elapsed time. The first is the fact that it takes time to physically move the lift platform between the bottom and the top location. Secondly, once the lift platform has reached the bottom it might have to wait an amount of time until a tote is available at the input queue. The combination of both factors, together with the arrival times of totes, have to be taken into account to determine the time used by the system to process a stream of totes.

We introduce the notion of the last action time (lat) into our model to denote the last moment in time that an action occurred. The lat must be updated every time a time related action is performed. Assume it takes $\frac{\delta}{2}$ seconds for the platform to travel between the top and bottom location of the mast, including (un)loading totes. The model consists of two processes: a delivery process and a lift process. The delivery process communicates the arrival time of the first tote in the queue to the lift process. Let this time be t_{in}. The lift process keeps track of the platform's location on the mast and the last action time of the system. This model assumes that the platform is initially located at the top of the mast and ready to travel to the bottom to process the next tote. Refer to Appendix A.1 for the complete mCRL2 model of the single-platform system and its details.

After the lift process handled a tote it needs to determine the new lat, lets call this lat'. Figure 4.3 displays both cases that can occur in this system while processing a tote, where it takes $\frac{\delta}{2}$ time to travel between the bottom to top with a platform. The first case shows that the lift can continue right away as there is a tote available once the platform has traveled to the bottom, thus $t_{in} \leq lat + \frac{\delta}{2}$. In the second case the platform arrives at the bottom but has to wait until a new tote is available, thus $t_{in} > lat + \frac{\delta}{2}$. As the platform must travel down in both cases we know that lat' is at least $lat + \frac{\delta}{2}$.

Next we determine how long it takes for the platform to reach the output again. The first case is simple since the system can continue immediately, which means that the cost of the platform traveling back to the output is the only cost that needs to be added ($\frac{\delta}{2}$). Thus in this case $lat' = lat + \delta$. In the second case we arrive at the bottom of the mast without any work to perform and we must wait until t_{in}, when the next tote arrives for processing. From that moment in time it must still travel to the output thus $lat' = t_{in} + \frac{\delta}{2}$. Combining these two cases we can calculate lat' by taking the earliest time a tote is ready and add the cost of the platform traveling back to the output, thus $lat' = max(t_{in}, lat + \frac{\delta}{2}) + \frac{\delta}{2}$. Note that lat' always denotes the time at which the last tote was delivered at the output.

The single-platform system must also keep track of the total time that each tote resides in the system, from the moment they enter the input queue until they are put on the output queue. This is known as the cycle-time of a tote. Using the last action time and the knowledge of the arrival times of each tote we are able to keep track of the time spent in the system per tote while they are being processed. These times are accumulated in a single variable in the lift process. The model will also count the total number of totes that are processed even though these are known beforehand. This counter is added to verify that all totes are processed when the model

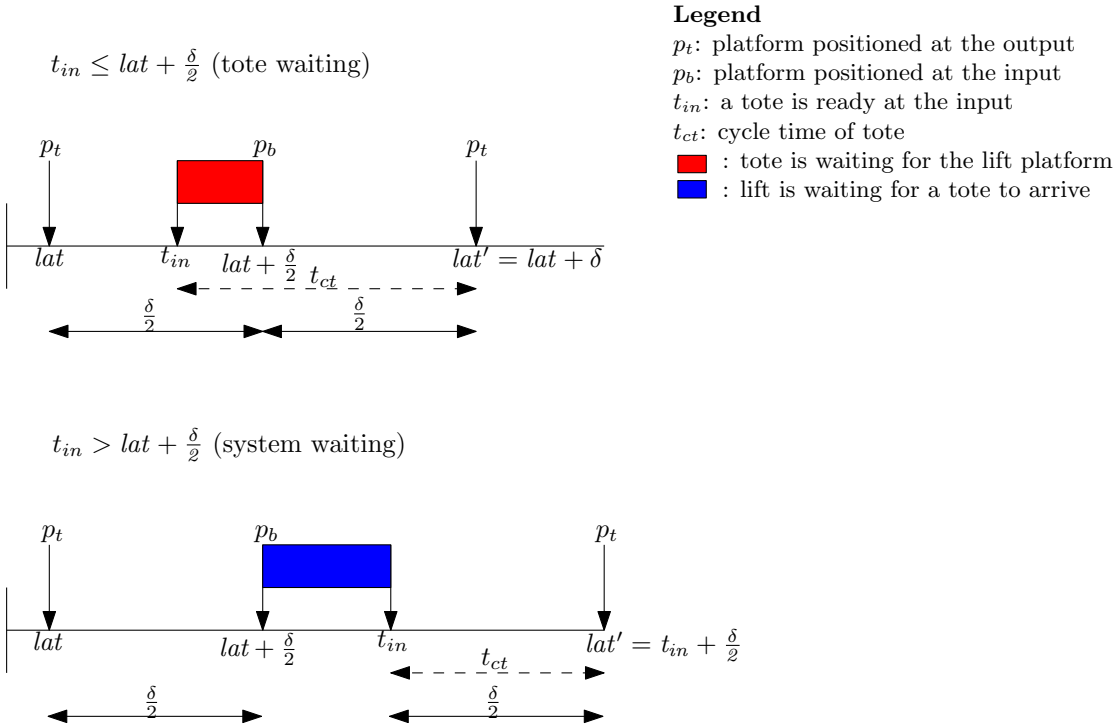


Figure 4.3: Platform arrival cases and cycle time

terminates with a result action, as to rule out any mistakes that are present in the model itself (firing the result actions prematurely). The cycle-time for both cases is displayed in Figure 4.3, indicated by t_{ct}. In the model this is calculated with the formula: $t_{ct} = lat' - t_{in}$.

Recall that the utilization is approximated and might deviate from the actual utilization of a simulation. The total idle time is also tracked by the model, to calculate the true utilization of a simulation. Idle time has to be calculated if the platform is waiting at the bottom of the mast for the next tote ($t_{in} > lat + \frac{\delta}{2}$), which is idle for $t_{in} - (lat + \frac{\delta}{2})$ time units. Otherwise it is zero as the system would continue right away.

The simulation is performed with the `lps2lts` tool that generates the entire statespace of the model. We supply the tool with the option to display the action 'result' which carries the accumulated cycle-times, the last action time, the total idle time and the number of the totes processed as parameters. This model will only have one result action as the whole system is completely deterministic in its behavior. These values are all that is required to calculate the utilization, throughput, cycle-time and work in progress of the simulation. The details are explained in the next section.

4.2.3 Simulation verification and results

The single-platform system can be denoted as an *M/D/1* model. Such a model consists of a server and a queue of objects waiting to be serviced. In this case one server (i.e., the single platform) handles transportation of totes along the mast with the queue being a conveyor with totes lining up in a FIFO order. The service is deterministic as it is constant: it always takes the same amount of time to move the platform along the mast. For this system we have a server time of 10 seconds which is *2C* in the mCRL2 model (see Appendix A.1). The performance of this system can be described with stochastic probability theories as explained in Section 4.1. For the single-platform system we have $c = 1$ and $\delta = 10$ for which the following holds:

$$\begin{aligned}\varphi &= \frac{1}{1-u}\left\{1-\frac{1}{2}u\right\}\delta-\frac{1}{2}\delta \\ &= \frac{1}{1-u}\left\{1-\frac{1}{2}u\right\}10-\frac{1}{2}\cdot 10 \\ &= \frac{1}{1-u}\left\{1-\frac{1}{2}u\right\}10-5 \ (s)\end{aligned} \tag{4.7}$$

$$\begin{aligned}\psi &= c\cdot\frac{u}{\delta} \\ &= \frac{u}{10} \ (totes/s)\end{aligned} \tag{4.8}$$

$$\begin{aligned}\xi &= \varphi\cdot\psi \\ &= \left\{\frac{1}{1-u}\left\{1-\frac{1}{2}u\right\}10-5\right\}\frac{u}{10} \ (totes)\end{aligned} \tag{4.9}$$

Here φ is the cycle-time, ψ is the throughput, ξ denotes the work in progress and $0 \leq u < 1$ is the utilization.

We performed a total of 20 simulations of the single-platform system, each with an input sequence length of 100.000 totes. Each simulation is performed by generating the statespace of the mCRL2 model with `lps2lts` (as discussed in the previous section) of which we store the result in a file. The inter-arrival times determine the utilization of the system and are determined with the utilization approximation Formula 4.6 rewritten to (with $\delta = 10$ and $c = 1$):

$$\begin{aligned}\delta_a &= \frac{1}{c}\cdot\frac{\delta}{u_{app}} \\ &= \frac{1}{1}\cdot\frac{10}{u_{app}} \\ &= \frac{10}{u_{app}} \ (s)\end{aligned} \tag{4.10}$$

We take u_{app} to be the desired utilization for a single simulation and use Formula 4.10 to determine δ_a with which we generate the exponential distribution sequence.

The values of the distribution require a minimal amount of time-precision for a realistic simulation. Simulation tests have shown that a time precision of 100^{th} of a second is sufficient whereas using plainly seconds distorts the results. As time is unit-less within the mCRL2 model we can scale time by multiplying the units by 100 to match the precision of the distributed sequences of inter-arrival times.

Recall that the result action contains the accumulated cycle-time, the last action time, total idle time and the total number totes processed. Let ct_{meas}, lat_{meas}, $idle_{meas}$ and cnt_{meas} denote these respectively. For this simulation of this system the following holds:

$$u_{meas} = 1-\frac{idle_{meas}}{lat_{meas}} \tag{4.11}$$

$$\varphi_{meas} = \frac{accu_{meas}}{100cnt_{meas}} \ (s) \tag{4.12}$$

$$\psi_{meas} = \frac{100cnt_{meas}}{lat_{meas}} \ (totes/s) \tag{4.13}$$

Here φ_{meas} is the average cycle-time per tote in seconds, ψ_{meas} is the throughput and u_{meas} is the utilization, all as measured in our simulation. The factor 100 is introduced to convert the results to seconds with a two digit precision. Note that the work in progress (ξ) is not measured but calculated with Little's Law (see Formula 4.9).

Figures 4.4, 4.5 and 4.6 display the results of our simulations which match the analytical model very well. Each of these figures plot one of the aspects to measure against the utilization (u_{meas}). Figure 4.4 displays the cycle-time of totes: the duration of transporting a tote from the moment it arrives at the input until it is transported to the output (including queuing time). Once the utilization approaches zero we see that the average cycle-time of a tote becomes five seconds. This makes sense as the inter-arrival times are large which means that the system is almost always ready to transport the tote as they arrive in the queue: the platform always has enough time to travel down to input before a new tote arrives (where it might even be waiting). As utilization increases we see that the average cycle-time increases because totes have to wait longer in the queue before it is their turn, which adds to their cycle-time. This also makes sense as inter-arrival times become small and cause a longer queue since there is less time for the platform to travel back down to input, before another tote arrives in the queue.

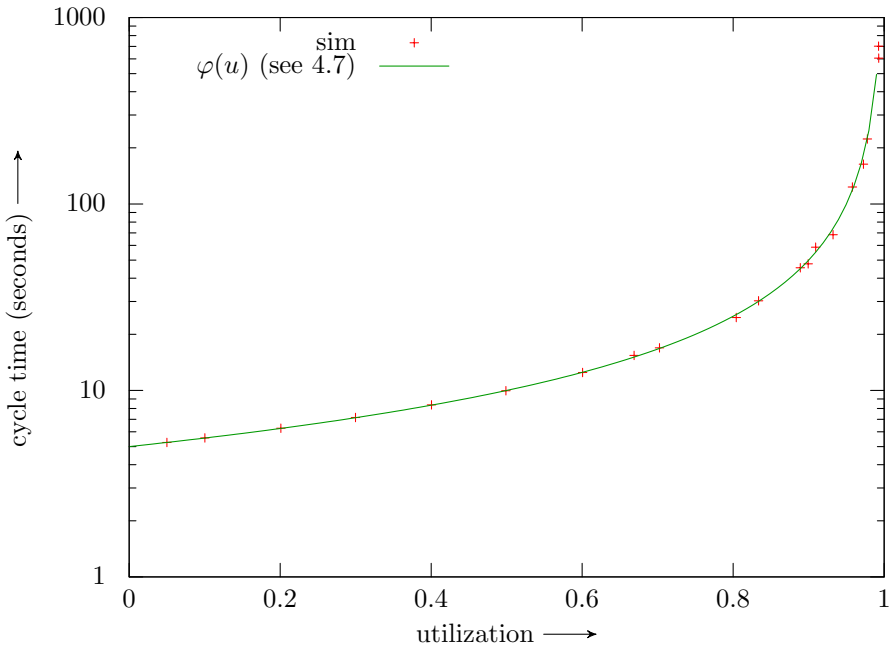


Figure 4.4: Single-platform system cycle time

Figure 4.5 displays the throughput of the system, expressed in the number of totes processed per minute. The throughput is linear with the utilization, which is expected because of conservation of the number of totes (they don't disappear). For $u = 1$ the throughput is 6 totes per minute, because the server time for handling a tote is 10 seconds.

The final aspect is work in progress (ξ), which shows the average number of totes that are waiting in queue for service, including the one being transported by the system. Figure 4.6 shows the work in progress which starts to increase exponentially once the totes' inter-arrival times become too small, since this means that totes get stacked up in the queue. This phenomenon occurs because the system has too little time to process each tote on average. Work in progress is equal to the product of the throughput and the cycle-time as denoted by Little's Law.

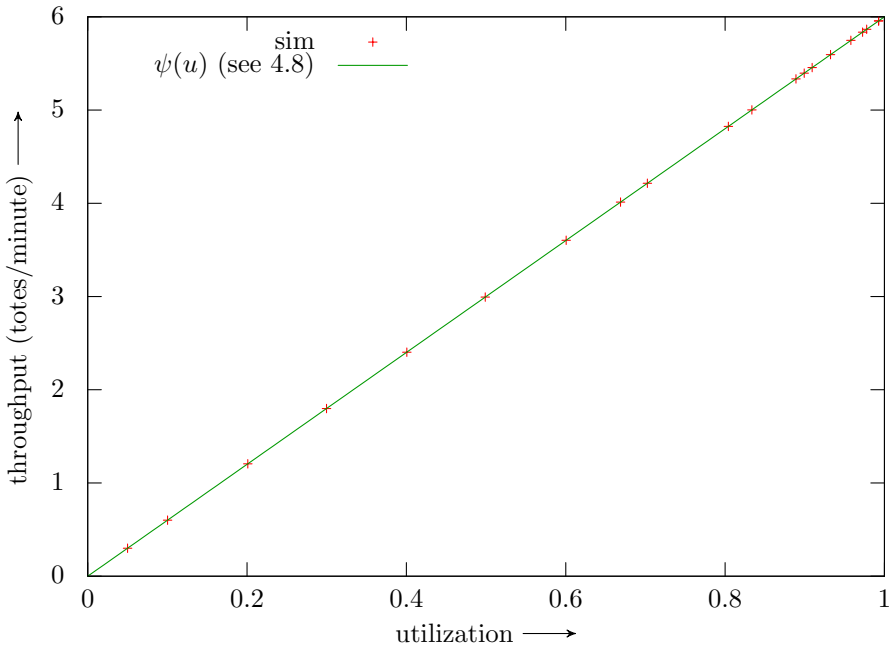


Figure 4.5: Single-platform system throughput

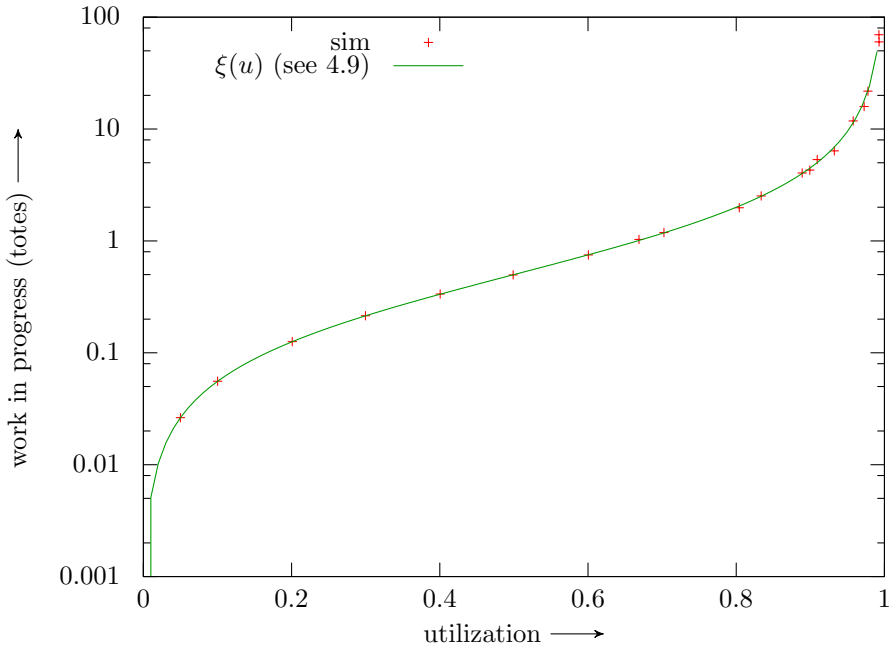


Figure 4.6: Single-platform system work in progress

4.3 Parallel-platform elevator system

In the previous section we introduced a method to determine the performance of the single-platform elevator system using the notion of last action time, described as a mCRL2 model. In this section we extend this work to the use of two platforms which operate parallel to each other. This extension is designed with the required properties for the HDS system controller model in mind so that we can use this model as a part of the HDS system.

4.3.1 Problem description

The complexity of the parallel-platform system introduces new problems as how to keep track of elapsed time as many processes run intertwined. The way time is tracked must be adapted such that each process knows the global time and can inform other processes that time has progressed at all times. Furthermore we must make sure that actions that execute at a certain point in time are performed before actions that occur at a later point in time. The simulation results must once again produce values for the same aspects as in the single-platform system namely: the utilization, the cycle-time, the total idle time, and the throughput of the system.

4.3.2 Approach

The parallel-platform elevator system setup is basically the same as the single-platform elevator system with the exception that there are two platforms (instead of one) that service totes independent of each other as depicted in Figure 4.7.

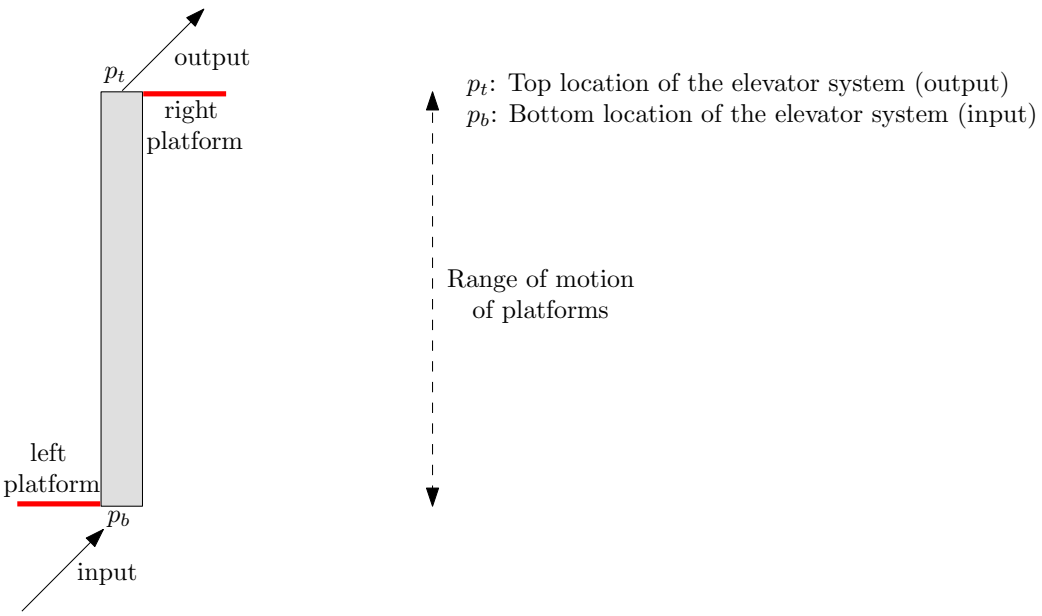


Figure 4.7: Parallel-platform elevator system

The HDS system setup also makes use of two platforms but these are located above each other which makes it impossible to pass each other. Therefore they are not truly working in parallel as they might block each others path to their destination. However, this behavior makes verification of our simulation results with an analytical approximation impossible; hence we remove this restriction here in the parallel-platform system. In the HDS model these restriction are not removed as they are part of the requirements of the system.

The single-platform system has exactly one action enabled in all possible states which allows us to store and determine all simulation values in one process once this action is performed. The parallel-platform system has multiple processes running in parallel and the possibility that multiple actions might be enabled simultaneously. A monitor process is introduced that will act as a timer to which processes can request the global current time before performing an action

and report their new local time after performing the action (assuming that such an action takes time). This way we are able to keep track of the last action time even though many processes are running in parallel.

We must also prioritize all actions such that those that occur at the earliest moment in time are performed before actions that occur later on. This is necessary, because time has been introduced as a simple parameter in the model without any additionally enforced behavior of the model. For this we use the *prioritized walk* option of the `lps2lts` tool, which adds a filter over the enabled actions for each state. This option will check the first argument of each action. If this argument is of the natural number type then it will only be enabled if and only if the value is the smallest number of all other actions which also have a natural number type as their first argument. Actions without such an argument are always selected as part of the enabled actions set. For example say that the actions $a(1)$, $b(1)$, $b(2)$ and $c(true)$ could occur. Without prioritized walk all these actions would remain enabled, otherwise $b(2)$ is eliminated since $2 > 1$. Note that actions without a natural number as the first argument can be performed at any time.

The flow of time is controlled by introducing such an additional argument to every action that takes time to perform it physically. This argument will be set to the time at which an action has to be performed. This will ensure that time cannot progress as long as an action with an earlier timestamp is enabled. There are four processes that make up the mCRL2 model: a monitor, a delivery, a lift platform and a stepper process. The delivery process keeps track of the arrival times of the totes. The lift platform process keeps track of the current position and physical state of a platform (moving, idle). Two of such processes are initiated. The stepper process represents the motors which drive the platforms and performs the steps the motors take. Each step will take one time unit in this model but the stepper process also makes sure that two simultaneous steps take a single time unit. The control process does all the bookkeeping regarding the performance values we need. The delivery process will push the next arrival time to the control process once the global timer has reached that time. The control process will assign a platform to service the tote. The platforms and stepper processes will perform the actual motion that is required to service the tote. Once a platform has delivered its tote to the output it will inform the control process of the time at it was delivered. The control process can then determine the cycle-time of the tote with both these times. It also knows the last time a platform has arrived at the input. It will use this value in combination with the next tote arrival time to determine how long it was idle (i.e. the queue was empty).

The delivery process advances the global time once all other timed actions are disabled. This is regulated by assigning a large constant value to the first argument of the 'advance time'-action such that it is always selected as the final action. Advancing time occurs when the system is completely idle and is waiting for the next tote. In this situation the global time will not progress as no time consuming actions take place anymore. The delivery process will then set the global time to next arrival time of the first upcoming tote. A more detailed description of this model is given in the HDS system controller model where all actions, processes and communications are defined and explained in detail (see Section 5.1). Refer to Appendix A.2 for the complete mCRL2 model of the parallel-platform system.

4.3.3 Simulation verification and results

The parallel-platform system can be denoted as an *M/D/2* model. The service time is still as it remains constant for both servers: the two platforms that transport the totes. The performance of this system can also be described with stochastic probability theories from Section 4.1. For this parallel-platform system we have $c = 2$ and $\delta = 10$ for which the following holds:

$$\begin{aligned} \varphi &= \frac{1}{2}\left\{1 + (1-u)\frac{\sqrt{14}-2}{32u}\right\}\frac{u^2}{1-u^2}10 + \frac{1}{2}10 \\ &= \frac{1}{2}\left\{1 + (1-u)\frac{\sqrt{14}-2}{32u}\right\}\frac{u^2}{1-u^2}10 + 5 \ (s) \end{aligned} \tag{4.14}$$

$$\begin{aligned} \psi &= c \cdot \frac{u}{(\delta)} \\ &= 2 \cdot \frac{u}{10} \\ &= \frac{u}{5} \ \textit{(totes/s)} \end{aligned} \tag{4.15}$$

$$\begin{aligned} \xi &= \varphi \cdot \psi \\ &= \left(\frac{1}{2} \left\{ 1 + (1-u) \frac{\sqrt{14}-2}{32u} \right\} \frac{u^2}{1-u^2} 10 + 5 \right) \frac{u}{5} \ \textit{(totes)} \end{aligned} \tag{4.16}$$

Again here φ is the cycle-time, ψ is the throughput, ξ denotes the work in progress and $0 \leq u < 1$ is the utilization.

For our simulations of the parallel-platform system we still have $\delta = 10$ (server time to travel between the top and bottom of the mast to service one tote). The mean inter-arrival time (to generate an exponential distribution sequence for our simulations) is determined with the utilization approximation Formula 4.6 (with $\delta = 10$ and $c = 2$):

$$\begin{aligned} \delta_a &= \frac{1}{c} \cdot \frac{\delta}{u_{app}} \\ &= \frac{1}{2} \cdot \frac{10}{u_{app}} \\ &= \frac{5}{u_{app}} \ \textit{(s)} \end{aligned} \tag{4.17}$$

The inter-arrival time precision is changed to a 10^{th} of a second to reduce the statespace to a reasonable size when the `lps2lts` tool is used to perform a simulation. The result action that is printed by this tool once again contains: u_{meas}, lat_{meas}, $idle1_{meas}$, $idle2_{meas}$ and cnt_{meas}. These values respectively denote: the utilization, the last action time, the total idle time (per platform), and the total number of totes processed. For the utilization (u_{meas}), the average cycle-time (φ_{meas}) and the throughput (ψ_{meas}) of our simulations the following respectively holds:

$$u_{meas} = 1 - \frac{\frac{1}{2}(idle1_{meas} + idle2_{meas})}{lat_{meas}} \tag{4.18}$$

$$\varphi_{meas} = \frac{accu_{meas}}{10 cnt_{meas}} \ \textit{(s)} \tag{4.19}$$

$$\psi_{meas} = \frac{10 cnt_{meas}}{lat_{meas}} \ \textit{(totes/s)} \tag{4.20}$$

Work in progress is once again calculated using Little's Law and is not measured within our simulations.

A total of 14 simulations have been performed with a input distribution sequence of 80.000 totes per simulation. The results depicted in Figures 4.8, 4.9 and 4.10 show that the simulation matches the analytical version very well. This model, especially its approach to control and to keep track of the time aspect, can now be used as a part of the HDS controller model with confidence. Note that the throughput is twice as high (as expected) with 12 totes per minute

(when $u = 1$): twice as much work can be handled in the same time span as two servers are working in parallel. Note that when the utilization approaches zero, we still have an average cycle-time of five seconds as this is the minimum time required to handle a tote. However the average inter-arrival times are halved in comparison to the single-platform system (cf. 4.10 and 4.17) for the same utilization (which complies with the doubled throughput). In other words: the rate at which totes are serviced is doubled.

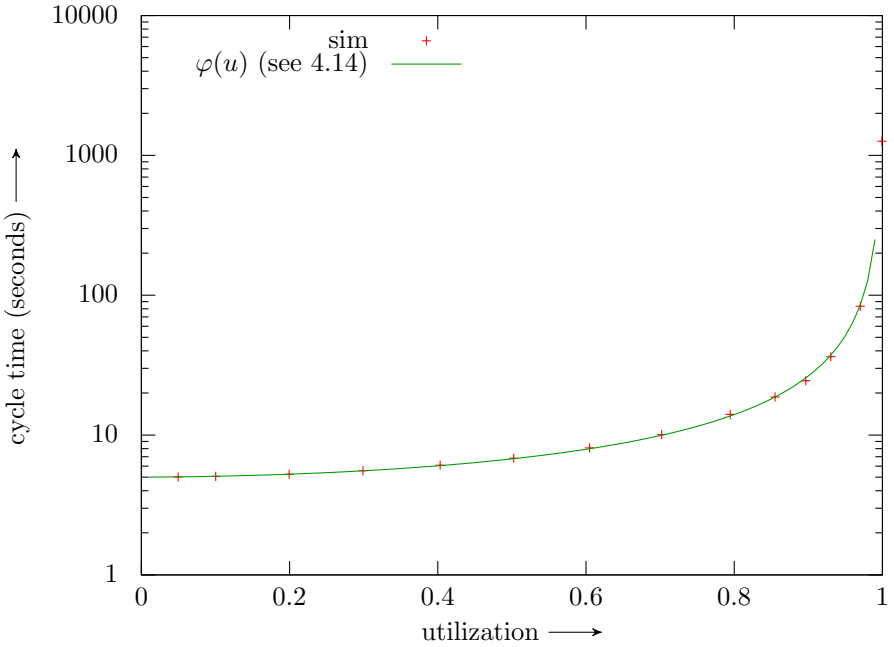


Figure 4.8: Parallel-platform elevator cycle time

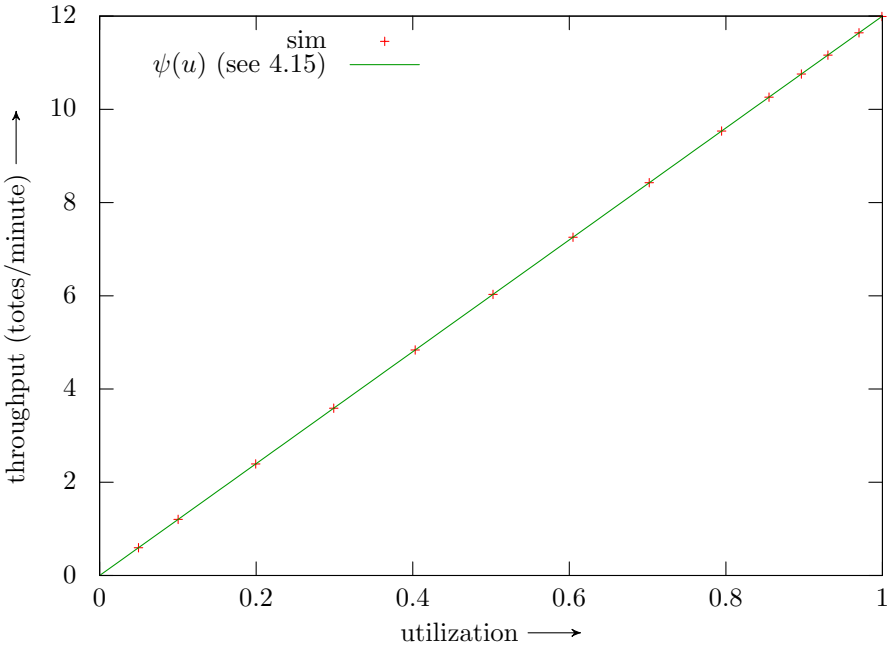


Figure 4.9: Parallel-platform elevator throughput

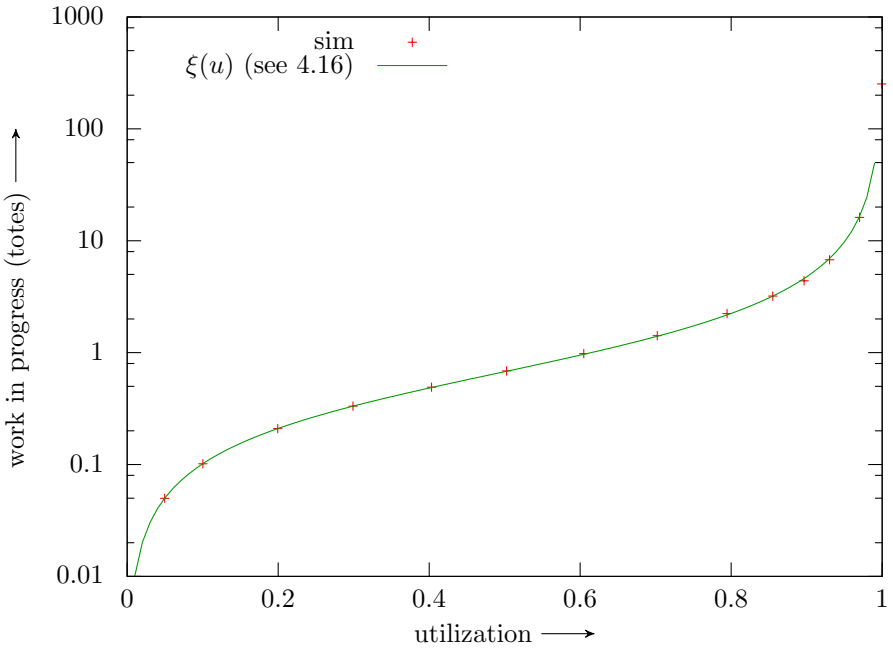


Figure 4.10: Parallel-platform elevator work in progress

Chapter 5

Modeling the HDS system

In this chapter the mCRL2 language, used to model the HDS system, is discussed. Subsequently the HDS system is described. The system is divided in layers where each has it own purpose and responsibilities. These layers are first discussed from an abstract point of view, followed by a detailed explanation of the model. The performance of the HDS system is discussed in the last section of this chapter.

5.1 mCRL2 language

The model is described in *mCRL2* which is based on the theory of *ACP*, as described in [1], [2] and [3]. Another thorough description is presented in [5]. In this chapter we look at this model but we first introduce the parts of the language that are used for this model.

5.1.1 Actions

Actions are the basic ingredients that make up a process. More precisely, an action can be seen as the most basic form of a process. Actions denote just what they suggest: the action that is performed by a system that is modeled. Actions are declared with the keyword **act** as follows:

act s;
r;

Here the actions s and r are declared that may denote the sending and receiving of messages. Actions carrying parameters must have them specified during the declaration of the action. We redeclare the actions s and r to send and receive a natural number:

act *s:* $\mathbb{N}$;
r: $\mathbb{N}$;

The natural number ($\mathbb{N}$) type is introduced, which is a predefined *standard data type* of mCRL2. The other predefined standard data types are: booleans ($\mathbb{B}$), positive numbers ($\mathbb{N}^+$), real numbers ($\mathbb{R}$), structured types, functions, lists and bags. In the HDS system model we often create a type alias for a standard data type to clarify the purpose of a parameter. The syntax to declare an alias is as follows:

sort *alias name* = *data type*

Structured types are used to introduce enumerated types and tuple types. An *enumerated type* is characterized as follows:

sort *enumeration name* = **struct** $value_1?isValue_1|value_2?isValue_2|\ldots|value_n?isValue_n$

The questionmark structure $x?y$ specifies one element of the enumeration where x denotes the enumeration value and optionally y specifies the name for a function that will be generated by mCRL2. This function is of the type *enumeration name* $\rightarrow \mathbb{B}$ and determines if a given

parameter of this type is of the value x. Without the optional function declaration the syntax is simply x, which is valid although no functions are generated.

A *tuple type* can be declared as follows:

sort *tuple name* = **struct** *pair(fst: data type, snd: data type)*

The function *fst* is used to access the first element of the tuple whereas *snd* can access the second element of the tuple. The value of the element is returned if these two functions are applied to a parameter of this tuple type. Any name is allowed for the names of the functions.

Finally we will have a look at *lists* and *sets*, which are a part of the predefined standard data types of mCRL2. It is quite intuitive to declare a list or set of data types by enclosing the data type with the keyword **List** and **Set**.

List(data type)
Set(data type)

For example a list of natural numbers 1, 5 and 3 is written as $[1, 5, 3]$ whereas a set is written as $\{1, 5, 3\}$. As expected the sequence of the elements in the list is maintained but not in the set. The empty list and set are denoted as [] and {}.

5.1.2 Action composition

Actions and processes can be combined with two main operators to get the desired behavior of your system: the sequential and alternative composition operators. Let a and b be actions then we write $a \cdot b$ to denote sequential execution of these actions: a is executed then b must follow, the b action can not be performed beforehand. Alternative composition allows us a choice between multiple actions and is written as $a + b$ which states that we can select to either do a a action or a b action. These two operators can be nested within each other. Let c be an action then we can write $a + b \cdot c$ to denote that we can either do an a action or we can do an b action. The b must be followed by a c action before any other action can occur again. Note from the previous example that the alternative composition operator binds stronger than the sequential composition operator.

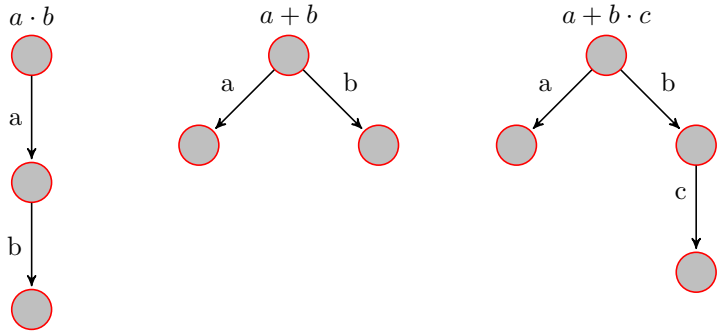


Figure 5.1: Composition operators

5.1.3 Processes

The HDS system model is divided in several processes that each have their own responsibilities. A *process* has a name and typically carries a number of parameters that denote the state of the process which control the flow of actions that describe it. A process is declared with the keyword **proc**, for example:

proc *P(sendMsg:* $\mathbb{B}$*, n:* $\mathbb{N}$*)* =
$sendMsg \rightarrow s(n) \cdot P(false, n)$
$\diamond \sum_{m:\mathbb{N}} \cdot r(m) \cdot P(true, m + 1)$;

First we take a little sidetrack from processes and see that this process contains a *conditional statement* and Σ operator. mCRL2 can execute an action under any given condition and optionally execute alternative actions if the condition does not hold. Of course the condition must have

a boolean result. The syntax for the conditional statement with, p and q processes or (sequences) of actions, is as follows:

$predicate \rightarrow p \diamond q$

The sum operator Σ is a generalization of the alternative composition operator. For finite data types we can even write out the complete specification that is generated of a sum operator. Let $d{:}\mathbb{B}$ be an action then:

$$\sum_{b:\mathbb{B}} d(b) \quad = d(false) + d(true)$$

In this case, if the process P has executed a number of actions it will become itself with updated parameters. This usually changes the selection path for the next action. In this example *sendMsg* determines if we perform an s or r action. We set *sendMsg* to false after a message is sent which means that we force the process to a *receive* action next. After a receive action it is set back to true which allows another *send* action. Thus we could also have written process P a lot shorter as follows:

proc $P(n : \mathbb{N}) = s(n) \cdot \sum_{m:\mathbb{N}} \cdot r(m) \cdot P(m + 1)$

5.1.4 Communication

The *parallel composition* operator || allows us to initialize as many instances of one or different processes as required, each one with their own private parameters. Often these processes want to exchange some data with each other. The process example from the previous section is an excellent example in which communication is useful. Let's say that we instantiate *P(false,0)* and *P(true,0)*, which we denote as $P(false, 0)||P(true, 0)$. This represent the process in which *P(false,0)* is waiting for a message and *P(true,0)* is ready to send a message.

Communication between two actions is possible if they share the same parameter data types and values. The two actions communicate to a third action in the model which denotes that the communication between the two actions took place. We introduce the action *srcomm* that acts as the communication action for our example. In the initialization block (see the next section) we need to indicate which actions are communication actions. In this case the following statement dictates the communication from the action s to r:

$s|r \rightarrow srcomm$

In our example this means that one process starts by performing the action *send(0)* which means that *receive(m)* occurs at the other process where $m = 0$ (the data is copied to the receiving process). This will show up as *srcomm* in mCRL2 tools where the *send* and *receive* actions are hidden. Now the two processes have actually switched their tasks: the send and receive actions are switched due to the process state change. The next message that is sent will be of value 1 as the state is updated to the incremented received value.

In the HDS system model a convention is used for the names of communication actions. A receiving action's name conforms to *r_actname* and a sending action's name conforms to *s_actname*. The communication action that communicates these two actions is named *actname*.

5.1.5 Initial state

The initial state is used to instantiate the processes and their parameters, comparable to the moment at which the a system is switched on. In this block we must indicate which actions are allowed, which actions communicate to each other. The example of the previous section is initialized as follows:

allow($\{srcomm\}$,
comm($\{s|r \rightarrow srcomm\}$,
$P(false, 0)||P(true, 0)$
))

The keyword **allow** is followed by a set of actions that may occur on their own. The communication actions are specified after the keyword **comm**. Note from the previous example that

the r and s are not specified in the *allow* section as it is implicitly specified that these actions must occur simultaneously in the **comm** section. Finally we initialize the processes and set their parameters as they should be initialized.

5.1.6 Functions

Functions can be declared for use in the mCRL2 specification of a system. First the function prototype must be declared along with the variables that are used in it. The keywords **map** and **var** are used to do so. The function can then be defined with the keyword *eqn*. Let's look at an example which accumulates a number of natural number from a list:

map $sum : List(\mathbb{N}) \to \mathbb{N};$
var $l : List(\mathbb{N}); e, acc : \mathbb{N};$
eqn $sum(e \triangleright l, acc) = sum(l, acc + e);$
$sum([], acc) = acc;$

The function's prototype is declared as a function of the type $List(\mathbb{N}) \times \mathbb{N} \to \mathbb{N}$. Multiple parameters of a function are separated by the $\times$ sign. Three variables are introduced: l, e and acc. Function definitions support pattern matching which is easier then writing conditional statements within the definition itself. So l contains the tail of the list and e contains the head. This means that at least one element must be in the list for this pattern to match, otherwise the list is empty. If the list is not empty then we add the value to the variable acc and repeat this for the tail of the list. Once the list is empty we return the result from acc. Note that pattern matching is only one choice of defining a function. Conditional statements in the definition of the functions, in this case to determine the emptiness of the list, are also a valid method of defining function although these tend to become messy and large in size.

5.2 Layers

In this section we take a look at the HDS system modular design. Before discussing each layer in detail, we first give an abstract overview of the model per layer. A schematic of the mCRL2 model is included as Appendix A.5.

The *monitor layer* represents a global timer from which the other layers can retrieve the current time. Other layers can also report their local time-stamp after they performed an action that consumes time like moving a lift platform. The monitor layer is not a part of the actual HDS system but is introduced in the model to keep track of time for simulation purposes.

The *delivery layer* simulates the conveyor that delivers totes for storage to HDS system. This layer will signal the arrival of a new tote to the system and will then wait until the lift platform layer send a signal that the tote is transferred onto a lift platform.

The *shuttle layer* is responsible for the retrieval of totes from the transfer-in points and for the storage of totes on the transfer-out points. This layer does not include the storage and retrieval of totes in the racks as we focus the model on the elevator system (see Section 2.2.2).

The retrieval and storage administration of totes at the transfer points is done by the *transfer agent layer*. This layer also determines if new totes are available for storage or retrieval and signals this to one of the lift agent layers.

The *lift agent layers* will determine if the retrieval-action, storage-action, simultaneous-action or sequential-action will be performed, as illustrated in Section 2.2.1, for both platforms have a lift agent. The lift agent signals the mast layer of the selected action (including storage and retrieve locations) and then waits until the mast layer returns a signal telling it that the selected action has been executed.

The *mast layer* creates a list of atomic action commands that reflect the selected actions from the lift agent layers. Next it sends the first atomic action command to the motion manager for either one of the lift platforms. In the case that there is an atomic action command to be executed for both lift platforms, and both lift platforms are idle, the mast will send both actions at once. It will wait for a signal from the motion manager telling it that the atomic action command is

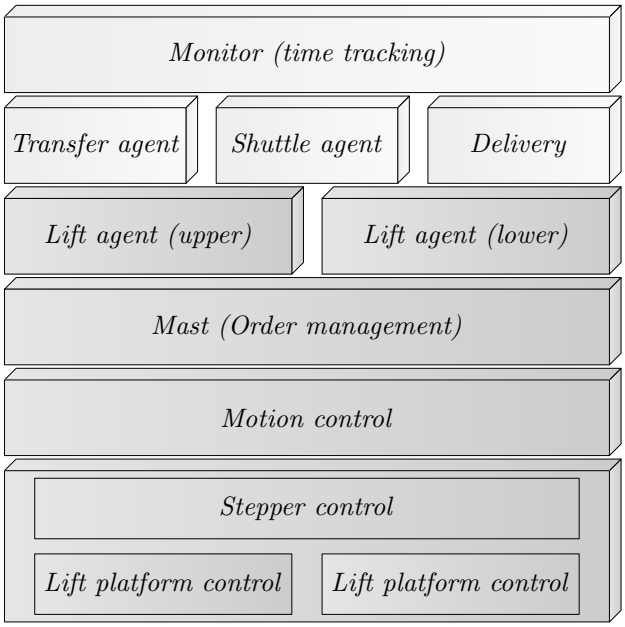


Figure 5.2: HDS layered view

executed after which it will send the next atomic action command in the list. In the case that the list is empty then the mast will signal the lift agent that its selected action has been executed.

The *motion manager layer* works at the knowledge level of atomic action commands. It makes sure that the two lift platforms do not collide against each other in case their atomic actions would cause such a situation. As it has no control over what these atomic action commands are, it uses the priority game (see Section 2.4) to temporarily move one lift platform out of the way, possibly overriding the current atomic action command for the corresponding lift platform. If the atomic action command can be executed safely, then it will signal the lift platform layer to perform the action (including the move command). It will then wait until the lift platform layer signals the completion of the command. If the atomic action command was not a move command (determined by the motion manager layer itself) then it will signal the mast layer that the atomic action command is executed.

The *lift platform layer* performs the atomic actions commands from the motion manager layer in cooperation with the stepper layer that controls the movement of the platforms (see the next paragraph). Each layer represent one lift platform and works independently. The stepper layer gets a signal to move a lift platform to a given location. The lift platform layer will then wait for the stepper layer to signal the arrival of the lift platform at the given location. Next it will transfer totes between lift platforms and conveyors or lift platforms and transfer points, except when the atomic action command was a move command. It will signal the delivery when a tote is retrieved from the input conveyor. Furthermore it will signal the transfer agent layer with information about the retrieved and stored totes on the transfer points. Finally it will signal the motion manager layer that the atomic action command is executed.

The *stepper layer* represents the stepper motors hardware that drive the lift platform. This layer starts the motor once the lift platform layer signals the next location to go to. The layer will then perform steps to move towards the locations. If both motors are enabled then the steps will occur sequentially: a single step for motor 1 and then a single step for motor 2 such that they remain synchronized. The lift platform layer is signaled once the location is reached to which the motor was commanded to drive the lift platform to.

Next we look at these layers in depth with all of the mCRL2 model specifics. We show the relationships between the layers and the communication that is involved to sustain these relationships. The communication actions in this model use the convention as explained in Section 5.1.4. Receiving actions are exactly the same as sending actions except that one sends the message and one receives the message. In this section we will explain communication actions in one single description, mentioning that it is a communication action.

Figure 5.2 shows all layers of the HDS system model. The dark colored blocks represent layers which have a strict relationship with respect to how commands are passed along. They are namely pushed from the top to the bottom starting at either one of the lift agents. A chain of signals is pushed back to the higher layers if the given order is executed. The light colored blocks are used by multiple other layers as we will see in the following paragraphs. As layers are explained we will encounter the actions and sorts that they use. These actions and sorts will be introduced and explained with their corresponding processes. Note that each process in the mCRL2 model represents one layer. Large processes are divided in smaller parts to keep a clear overview of the tasks a process performs.

Most actions have a time parameter, $t : \mathbb{N}$, that is used to determine which actions have priority over others. The value of this parameter dictates the time at which the action occurs, where lower values have priority over higher values. This nicely corresponds with how time works in the real world, performing actions at time t_1 before t_2 for all values satisfying $t_1 < t_2$.

5.2.1 Monitor

This layer is not a part of the HDS system itself, but manages time progression of the complete model. It acts as a global timer which allows each process to request the current time and update the time if it has performed an time consuming action.

Sorts

Sort	*Type*	*Description*
time	$\mathbb{N}$	Represents time unit-less.

The sort *time* is used in the model to store absolute time in the system, measured from the moment that it is switched on. In this model each time unit represents a tenth of a second which provides a good balance between simulation accuracy and model feasibility. Choosing smaller time units will cause the mCRL2 model to grow too large in terms of memory usage while running the simulation. Time units representing a second of time use less memory but make the simulation results inaccurate as discussed in Section 4.3.

The process of this layer is defined as follows with data parameter t which contains the global absolute time of the system:

$$monitor(t : time)$$

Actions

Action	*Data*	*Description*
getlat	*time*	This communication action contains a message with the current time value.
setlat	*time*	This communication action contains a message with the new time value.

The layers in the system can request the current time from the monitor with the *getlat* communication action. This doesn't change the state of the monitor process. Each layer can update the global time with the *setlat* action which accepts a value to update the monitor's current time. Note that the monitor does not allow any time values that occur in the past by the conditional restriction $l \geq t$.

$$\sum_{l:time}(l \geq t) \rightarrow r\_setlat(l) \cdot monitor(l)$$
$$+$$
$$s\_getlat(t) \cdot monitor(t)$$

Action

Action	*Data*	*Description*
tick	$time \times time$	This communication action has the lowest possible priority of all timed actions. It contains a message containing a new time.

Once all totes are processed such that none remain waiting to be handled, then the system will no longer perform actions that consume time. This results in a system in which time does not progress anymore. In other words the system becomes deadlocked because we update the time value based on progress. The communication action *tick* is introduced to overcome this modeling issue. As can be seen in the model fragment below, a constant `LOWPRIOR` is assigned to the first parameter of this action. This constant is a large value of the sort time of which we assume that for all model times t, the condition `LOWPRIOR` $> t$ holds. Using the priority properties from mCRL2 (see Section 4.3.2) we know that actions with the highest value are selected as last thus this action will only be performed if no other timed actions are possible. The monitor expects the earliest absolute time when work is available again, namely the arrival time of the next tote at the input conveyor.

$\sum_{d:time} r\_tick(\texttt{LOWPRIOR}, d).monitor(d)$

The first parameter of almost every action is of the sort *time* which specifies at what time the action occurs. The priority property is applied such that actions execute in order of time as actions with the lowest time value are executed first. From now on we assume that all actions with the sort *time* as their first parameter indicates the time at which they occur.

5.2.2 Lift platform

This layer represents a single lift platform of the elevator system which keeps track of the current state of that lift platform.

Sorts

Sort	*Type*	*Description*
lift_id	$\mathbb{N}^+$	A unique value to identify a lift platform in the mast.
location	$\mathbb{N}^+$	The height at which a lift platform is in the mast.
lift_action	Enumeration	The transfer action that the lift platform performs after it arrives at its destination.
lift_cmd	(*location*,*lift_action*)	Command tuple containing all the data for an atomic action.

The sort *lift_id* is an unique identifier assigned to a lift platform. In this model the lower lift platform is identified as 1 and the upper lift platform as 2. From now on we assume that all actions that contain this sort indicate that the action is related to the lift platform with the given identifier unless mentioned otherwise. Conveyor belts and transfer points are accessible at a static location where they are physically located. The two elements in the tuple *lift_cmd* can be accessed respectively with two functions: *dest* and *action*.

The enumeration *lift_action* consists of the following values:

Value	*Description*
loadtransfer	Transfer a tote on the lift platform (*TI*) to the transfer-in point where the lift platform is located.
unloadtransfer	Transfer a tote on the lift platform (*TO*) to the transfer-out point where the lift platform is located.
loadunloadtransfer	Perform *loadtransfer* and *unloadtransfer* simultaneously.
loadinput	Transfer the tote on the input conveyor to the lift platform (*TI*).
unloadoutput	Transfer the tote on the lift platform (*TO*) to the output conveyor.
move	No transfer actions are performed.

The process that represents this layer is defined as:

$$liftplatform(id : lift\_id, c : location, d : location, busy : Bool, a : lift\_action, e : time, i : time)$$

The parameter *id* contains an unique identifier of this lift platform (either 1 or 2) and is considered immutable. The parameter c contains the current height of the lift platform in the mast and parameter d is the destination location to which the lift platform must move to. The parameter *busy* indicates if the lift platform is in the process of moving to the destination as indicated by

d and the parameter a tells the lift platform what action it must perform once it has reached this destination. The parameter e and i respectively denote the time at which the platform last became idle and the total time it has been idle. Note that the calculations of these values are the same as from the control process of the parallel lift system (see Section 4.3), though now contained in this process. The arrival times, cumulative process time and the total totes count are now included in the mast layer which is discussed in the upcoming Section 5.2.5.

Based on the parameter *busy* we divide this process in two parts: one where the lift platform is idle and receives the next order and one where we the lift is busy handling a command. Commands that are received by this process are atomic actions as described in Section 2.2.1.

Actions

Action	*Data*	*Description*
sendLift	*time* × *lift_id* × *lift_cmd*	This communication action contains a message with an atomic action command.
startmotor	*time* × *lift_id* × *location*	This communication action contains a message which indicates to start the motor and move the lift platform to the given location.
print_idle	*lift_id* × *time*	This communication action contains a message which reports the total idle time of the platform to the mast layer (at the end of a simulation).

A lift platform waits for a command from the motion manager layer which it receives with the *sendLift* communication action. The command is then executed with the *startmotor* communication action which sends a message to the stepper layer, commanding the motor to move lift platform *id* to the destination location as given in *dest(cmd)*. The process parameters are updated to reflect the command by storing its destination and transfer action (d and a) and the *busy* parameter is set to true, indicating that the platform is performing a command.

$\neg busy \rightarrow$
$\sum_{cmd:lift\_cmd,t:time} r\_sendLift(t, id, cmd) \cdot s\_startmotor(t, id, dest(cmd)) \cdot$
$\sum_{lat:time} r\_getlat(lat) \cdot liftplatform(id, c, dest(cmd), true, action(cmd), e, i + (lat - e))$

Actions

Action	*Data*	*Description*
step	*time* × *lift_id* × *location*	This communication action contains a message with the location to where the motor moved the lift platform.
liftArrived	*time* × *lift_id*	This communication action contains a message indicating that the lift platform completed its current atomic action command.
stopmotor	*time* × *lift_id*	This communication action contains a message that the motor stopped.
loadedInput	*time*	This communication action contains a message indicating that the tote is transferred from the input conveyor to the lift platform.
tsoutput	*time* × *location*	This communication action contains a message indicating that the tote from the transfer-out point (at location) is transferred to the lift platform.
tsinput	*time* × *location*	This communication action contains a message indicating that the tote from the lift platform is transferred to the transfer-in point (at location).
liftLoad	*time* × *lift_id*	This action indicates that a tote is to be transferred from a transfer-out point to the lift platform.
liftUnload	*time* × *lift_id*	This action indicates that a tote is to be transferred from the lift platform to a transfer-in point or the output conveyor.
liftLoadInput	*time* × *lift_id*	This action indicates that a tote has to be transferred from the input conveyor to the lift platform.
liftUnloadOutput	*time* × *lift_id*	This action indicates that a tote is to be transferred from the lift platform to the output conveyor.

Once the command is received then the process will receive information from the stepper layer that the lift platform has moved one location with the *step* communication action. There are two cases that need to be distinguished: the current location of the lift platform is below the destination location or above it. This respectively means that the lift platform is moving up or down. The parameter c it updated accordingly and steps keep occurring until the destination location is reached.

$(busy \wedge c < d) \rightarrow \sum_{t:time} s\_step(t, id, c+1) \cdot liftplatform(id, c+1, d, busy, a, e, i)$
$+$
$(busy \wedge c > d) \rightarrow \sum_{t:time} s\_step(t, id, c-1) \cdot liftplatform(id, c-1, d, busy, a, e, i)$

Once we received the message from the stepper layer that the motor has stopped with the *stopmotor* communication action then we will handle the transfer action a. Unless a is *move*, the process will first perform the action that the tote must be loaded to a lift platform with the action *liftLoad* or unload a tote from a lift platform with the action *liftUnload*. After a transfer action the global time is increased by 5 time units: the time required to physically perform the transfer action. Note that five sequential *setlat* actions are shorthanded as $setlat(t_1 \ldots t_5)$ where we update t to $t+5$ in five steps. In case a tote is loaded from the input conveyor the *loadedInput* communication action is issued to inform the delivery layer that the first tote in the queue is retrieved. The retrieval of a tote from a transfer-out point and the storing of a tote on a transfer-in point must be signaled to the transfer agent layer that monitors the buffers. In order to do this the communication actions *tsinput* and *tsoutput* are performed to inform the transfer agent layer of the changes. Finally the process informs the the motion manager layer that the command is executed and that the lift platform is ready for the next one. Note that the parameter *busy* is set to false to indicate that the lift platform is waiting for the next command.

$(busy \wedge c = d) \rightarrow \sum_{t:Nat} r\_stopmotor(t, id).($
  $is\_loadinput(a) \rightarrow$
    $liftLoadInput(t, id) \cdot s\_setlat(t_1 \ldots t_5) \cdot$
    $s\_loadedInput(t+1) \cdot s\_liftArrived(t+1, id) \cdot liftplatform(id, c, d, false, a, t+5, e)$
  $+$
  $is\_move(a) \rightarrow$
    $s\_liftArrived(t, id) \cdot liftplatform(id, c, d, false, a, t, e)$
  $+$
  $is\_loadtransfer(a) \rightarrow$
    $liftLoad(t, id) \cdot s\_setlat(t_1 \ldots t_5) \cdot$
    $s\_tsoutput(t+1, d) \cdot s\_liftArrived(t+1, id) \cdot liftplatform(id, c, d, false, a, t+5, e)$
  $+$
  $is\_unloadtransfer(a) \rightarrow$
    $liftUnload(t, id) \cdot s\_setlat(t_1 \ldots t_5) \cdot$
    $s\_tsinput(t+1, d) \cdot s\_liftArrived(t+1, id) \cdot liftplatform(id, c, d, false, a, t+5, e)$
  $+$
  $is\_unloadoutput(a) \rightarrow$
    $liftUnloadOutput(t, id) \cdot s\_setlat(t_1 \ldots t_5) \cdot$
    $s\_liftArrived(t+1, id) \cdot liftplatform(id, c, d, false, a, t+5, e)$
  $+$
  $is\_loadunloadtransfer(a) \rightarrow$
    $liftLoad(t) \cdot liftUnload(t) \cdot s\_setlat(t_1 \ldots t_5) \cdot$
    $s\_tsinput(t+1, d) \cdot s\_tsoutput(t+1, d) \cdot$
    $s\_liftArrived(t+1, id) \cdot liftplatform(id, c, d, false, a, t+5, e)$
$)$

5.2.3 Stepper

This layer represent the two motors that move the lift platforms up and down along the mast. It also insures that the two motors take the same amount of steps if they are both moving their lift platform.

Sorts

Sort	*Type*	*Description*
direction	Enumeration	The direction in which the lift is traveling.
step_action	Enumeration	The current internal state of a motor (which move the lift platforms).

The enumeration *direction* consists of the following values:

Value	*Description*
up	The lift platform is moving upwards.
down	The lift platform is moving downwards.
neutral	The lift platform is not moving.

The enumeration *step_action* consists of the following values:

Value	*Description*
idle	The motor is idle.
stepping	The motor moving a lift platform.
arrived	The motor transported the lift platform the new location.

The process that represents this layer is defined as:

$$stepper(c1 : location, c2 : location, m1 : direction, m2 : direction, lat : time, \\ p1 : location, p2 : location, s1 : step\_action, s2 : step\_action)$$

The parameters in this process come in pairs of two, one reflecting the status related to (lower) lift platform 1 and one the reflecting the status related to (upper) lift platform 2. We will explain these kind of parameters as $parameter_x$ instead of $parameter_1$ and $parameter_2$, because they both hold the same meaning. The parameter c_x holds the current position of the lift platform and p_x holds the destination location. The parameter m_x holds the current direction in which the lift platform is traveling or if its not moving. The parameter *lat* holds the last action time for the lift platforms. Finally the parameter s_x hold the internal state of the motor.

In this section we leave out parts of the mCRL2 specification which are identical, except that they are related to the other lift platform. These parts do not add any information but do take up a lot of space. Refer to Appendix A.3 for the complete mCRL2 model specification.

The stepper receives a *startmotor* communication action from the lift platform layer and updates the process parameters accordingly. First we use the function *setDir* to determine the direction in which the lift platform will travel. We also store the time at which the motor was started in *lat*. Finally we set the destination location in parameter p and set the internal status of the motor to *stepping* in parameter s, to indicate that the motor will now perform the steps required to reach the destination. Note that the current time is updated after the motor is started. This is required as the times in the actions that eventually lead to the *startmotor* action might be in the past once they can be handled later if a platform becomes available.

$$\sum dest : location, id : lift\_id, t : timer\_startmotor(t, id, dest) \cdot \sum_{pt:time} r\_getlat(pt) \cdot \\ stepper(c1, c2, \\ if(id = 1, setDir(c1, dest, m1), m1), \\ if(id = 2, setDir(c2, dest, m2), m2), pt, \\ if(id = 1, dest, p1), if(id = 2, dest, p2), \\ if(id = 1, stepping, s1), if(id = 2, stepping, s2))$$

As long as neither motor has arrived at its destination then it will continue to perform the *step* communication action. With each *step* action the current location c_x is updated according to the

direction that the lift platform is going. The parameter s is set to arrived once the destination is reached. After each step the global time is increased by 1 unit with *setlat* (the time required to move to the next location) and also updates *lat* such that the next step will occur 1 time unit later. If both lift platforms are in motion then two steps are performed sequentially before the stepper updates the *lat* parameter.

$$
\begin{array}{l}
(\neg is\_arrived(s1) \wedge \neg is\_arrived(s2)) \rightarrow (\\
\quad (is\_neutral(m2) \wedge is\_up(m1)) \rightarrow \\
\qquad \sum_{d:location} r\_step(lat+1, 1, d) \cdot s\_setlat(lat+1) \cdot \\
\qquad stepper(c1+1, c2, m1, m2, lat+1, p1, p2, \\
\qquad\qquad if(c1+1 = p1 \wedge is\_stepping(s1), arrived, s1), \\
\qquad\qquad if(c2 = p2 \wedge is\_stepping(s2), arrived, s2)) \\
)
\end{array}
$$

The table below lists all the possibilities for the step actions, including the sequence in which they are performed:

m_1	m_2	*Performed steps*
neutral	*up*	Lift platform 1 moves up one location.
neutral	*down*	Lift platform 1 moves down one location.
up	*neutral*	Lift platform 2 moves up one location.
down	*neutral*	Lift platform 2 moves down one location.
down	*up*	Lift platform 1 moves down one location, lift platform 2 moves up one location.
up	*down*	Lift platform 1 moves up one location, lift platform 2 moves down one location.
down	*down*	Lift platform 1 moves down one location, lift platform 2 moves down one location.
up	*up*	Lift platform 2 moves up one location, lift platform 1 moves up one location.

Once the motor has arrived at the destination it will signal the lift platform layer that the motor stopped with the *stopmotor* communication action. The internal status of the motor is updated to idle in the s_x parameter to indicate that it completed its work.

$$
\begin{array}{l}
\sum_{id:lift\_id} s\_stopmotor(lat, id) \cdot \\
\quad stepper(c1, c2, \\
\qquad\qquad if(id = 1, neutral, m1), \\
\qquad\qquad if(id = 2, neutral, m2), \\
\qquad\qquad lat, p1, p2, \\
\qquad\qquad if(id = 1, idle, s1), if(id = 2, idle, s2))
\end{array}
$$

5.2.4 Motion manager

The motion manager layer makes sure that atomic action commands are only executed if it does not result in the lift platforms colliding. It also handles conflicts if the two lift platforms would block each others work.

Sorts

Sort	*Type*	*Description*
lift_status	Enumeration	The current state of the motion of the lift platforms.

The enumeration *lift_status* consists of the following values:

Value	*Description*
lidle	The lift platform is idle and waiting for a new command.
busy	The lift platform is in the process of handling a command.
standby	The lift platform has an command which can be dispatched to the next layer.

The process of this layer is defined as follows:

$$
\begin{array}{l}
motion\_mng(l1\_cmd : lift\_cmd, l2\_cmd : lift\_cmd, l1 : lift\_status, \\
\qquad\qquad l2 : lift\_status, priority : lift\_id, detour : lift\_cmd, \\
\qquad\qquad dt : Nat, s1 : time, s2 : time)
\end{array}
$$

The parameter l_x_cmd contains the atomic action command that the lift platform has to perform. The parameter l_x contains the internal state of the motion manager for the lift platform. The *priority* parameter indicates which platform will get precedence if one has to move out of the way. The *detour* parameter is a temporary variable which contains the atomic action command of the platform that had to move away but also wanted to execute its own command. The parameter *dt* indicates which lift platform is busy with such a detour. The parameter s_x is the start time of the atomic action.

Actions

Action	*Data*	*Description*
sndCmds	*time* × *lift_cmd* × *lift_cmd*	This communication action contains a message with two atomic action commands (one for each lift platform) to execute.
sendCmd	*time* × *lift_id* × *lift_cmd*	This communication action contains a message with a new atomic action command for the lift platform.
cmdDone	*time* × *lift_id*	This communication action contains a message indicating that the atomic action is executed.

The motion manager process receives an atomic action command from the mast layer with the *sendCmd* and *sendCmds* communication actions. It stores the command in the corresponding l_x_cmd parameter and sets the internal state of the motion manager for the corresponding lift platform to *standby*. If the state for both lift platforms is *lidle* then it is possible to receive both commands at once (see *sendCmds*), otherwise it receives only one atomic action command with the *sendCmd* communication action. If both commands arrive together then we know that both lift platforms are idle and we just store the commands for execution. In the single command case we will first check if the motion manager is currently not moving the lift platform (for which a command is received) to another location such that the other lift platform can perform its work. Note that this action command is determined by the motion manager process of which the mast layer has no knowledge.

If the motion manager ordered a lift platform to move out of the way then it will store the current command of that lift platform (which is not executed yet) in the *detour* parameter, because l_x_cmd is in use by the 'move' atomic action command. The parameter *dt* is set to the identity of the lift platform for which the command from the mast is intended such that the *detour* command can be restored to the correct lift platform. This is done when the lift platform layer notifies the completion of an atomic action command with the *liftArrived* communication action. The motion manager process will signal the mast layer that a command (and only commands send by the mast layer) that the command is executed with the *cmdDone* communication action. Note that this action is not performed if the atomic action command was a move command as this is ordered by the motion manager. Furthermore the process will check if any detour commands were stored if it has completed a move action and restores it if so. The internal state of the motion manager for the lift platform is also set: *lidle* if no detour command exists and *standby* if it does.

$$
\begin{array}{l}
(is\_lidle(l1) \wedge is\_lidle(l2)) \rightarrow \\
\quad \sum_{cmd1,cmd2:lift\_cmd,pt:time} \\
\quad r\_sendCmds(pt, cmd1, cmd2) \cdot \\
\quad motion\_mng(cmd1, cmd2, standby, standby, priority, detour, dt, pt, pt) \\
+ \\
\sum_{cmd:lift\_cmd,id:lift\_id,pt:time} \\
\quad r\_sendCmd(pt, id, cmd) \cdot (\\
\qquad (id = 1 \wedge is\_busy(l1) \wedge is\_move(action(l1\_cmd))) \rightarrow \\
\qquad\quad motion\_mng(l1\_cmd, l2\_cmd, l1, l2, priority, cmd, 1, pt, s2) \\
\qquad + \\
\qquad (id = 1 \wedge is\_lidle(l1)) \rightarrow \\
\qquad\quad motion\_mng(cmd, l2\_cmd, standby, l2, priority, detour, dt, pt, s2) \\
\quad) \\
+ \\
\sum_{pt:time} r\_liftArrived(pt, 1) \cdot (\\
\quad is\_move(action(l1\_cmd)) \rightarrow (\\
\quad motion\_mng(if(dt = 1, detour, l1\_cmd), l2\_cmd, \\
\qquad if(dt = 1, standby, lidle), l2, \\
\qquad priority, detour, 0, s1, s2) \\
\quad) \\
\qquad + \\
\qquad \neg is\_move(action(l1\_cmd)) \rightarrow (\\
\qquad\quad s\_cmdDone(pt, 1) \cdot \\
\qquad\quad motion\_mng(l1\_cmd, l2\_cmd, lidle, l2, priority, detour, dt, s1, s2) \\
\qquad) \\
)
\end{array}
$$

After validating that the execution of an atomic action does not cause a collision between the two lift platforms, then the motion manager process signals the lift platform processes to execute the atomic action command with the *sendLift* communication action. When both lift platforms are idle and the destination of both commands are such that the lift platform will not collide, then both are signaled to perform their commands. In case that the destinations do cause a collision then the lift with priority will perform its command first and the other one has to move out of the way. The lift platform with priority is the one of which its identifier is equal to the value of *priority*. The command that was supposed to be executed is temporary stored in *detour* and *dt* is set to the identity of the lift platform without priority. Finally we set the priority to the other lift platform according to the priority game (see Section 2.4).

$$
\begin{array}{l}
(is\_standby(l1) \wedge is\_standby(l2)) \rightarrow (\\
\quad (dest(l1\_cmd) < dest(l2\_cmd)) \rightarrow \\
\qquad s\_sendLift(s2, 2, l2\_cmd).s\_sendLift(s1, 1, l1\_cmd). \\
\qquad motion\_mng(l1\_cmd, l2\_cmd, busy, busy, priority, detour, dt, s1, s2) \\
\quad + \\
\quad (dest(l1\_cmd) \geq dest(l2\_cmd)) \rightarrow \\
\qquad s\_sendLift(if(priority = 1, s2, s1), if(priority = 1, 2, 1), \\
\qquad\quad pair(if(priority = 1, dest(l1\_cmd) + 1, dest(l2\_cmd) - 1), move)). \\
\qquad s\_sendLift(if(priority = 1, s1, s2), if(priority = 1, 1, 2), \\
\qquad\quad if(priority = 1, l1\_cmd, l2\_cmd)). \\
\qquad motion\_mng(if(priority = 1, l1\_cmd, pair(dest(l2\_cmd) - 1, move)), \\
\qquad\quad if(priority = 2, l2\_cmd, pair(dest(l1\_cmd) + 1, move)), \\
\qquad\quad busy, busy, if(priority = 1, 2, 1), \\
\qquad\quad if(priority = 1, l2\_cmd, l1\_cmd), \\
\qquad\quad if(priority = 1, 2, 1), s1, s2) \\
)
\end{array}
$$

When one lift platform has a command to perform but the other does not then the motion manager will check if the current location of the idle lift platform is not blocking the others

destination. If this is not the case the lift platform will be signaled, otherwise it will also execute a move action on the other lift platform such that it moves out of the way. Note that we use the command destination value as the current location of a idle lift platform. This is always correct since the last command is always stored in this parameter.

$$
\begin{array}{l}
(is\_standby(l2) \wedge is\_lidle(l1)) \rightarrow (\\
\quad (dest(l2\_cmd) > dest(l1\_cmd)) \rightarrow \\
\quad\quad s\_sendLift(s2, 2, l2\_cmd). \\
\quad\quad motion\_mng(l1\_cmd, l2\_cmd, l1, busy, priority, detour, dt, s1, s2) \\
\quad + \\
\quad (dest(l2\_cmd) \leq dest(l1\_cmd)) \rightarrow \\
\quad\quad s\_sendLift(s1, 1, pair(dest(l2\_cmd) - 1, move)). \\
\quad\quad s\_sendLift(s2, 2, l2\_cmd). \\
\quad\quad motion\_mng(pair(dest(l2\_cmd) - 1, move), l2\_cmd, busy, busy, 1, detour, dt, s1, s2) \\
)
\end{array}
$$

The final case that must be considered is when one lift platform is currently executing a command (already in motion) and the other lift platform has a command to perform. The command is only executed if no blocking issues exist, otherwise the motion manager does nothing until the other lift platform is finished with its current command. The motion manager will then determine the which command is executed next according to the above explained strategies.

$$
\begin{array}{l}
(is\_standby(l1) \wedge is\_busy(l2) \wedge dest(l1\_cmd) < dest(l2\_cmd)) \rightarrow (\\
\quad s\_sendLift(s1, 1, l1\_cmd). \\
\quad motion\_mng(l1\_cmd, l2\_cmd, busy, l2, priority, detour, dt, s1, s2) \\
)
\end{array}
$$

5.2.5 Mast

The mast layer is responsible for creating the list of atomic actions for the given command from the lift agent layer, which relays the list of atomic actions commands to the motion manager layer one by one. The motion manager layer is responsible for executing these commands safely.

The process that represents this layer is defined as:

$$
\begin{array}{l}
mast(l1 : lift\_status, l2 : lift\_status, l1\_work : List(lift\_cmd), l2\_work : List(lift\_cmd), \\
\qquad l1\_loc : location, l2\_loc : location, \\
\qquad sin1 : time, sin2 : time, sout1 : tim, sout2 : time, ct : time, cnt : Nat, e1 : time, e2 : time)
\end{array}
$$

The internal status of the mast (per lift platform) is stored in the l_x parameter. The list of atomic actions commands is stored in the parameter l_x_work. Parameter l_x_loc contains the location of the lift platform where a lift platform ends up after a atomic action command. The parameters sin_x and $sout_x$ contain the arrival times of the totes at the input conveyor and the totes that arrived at a transfer-out point. Next we have the cumulative time of the totes process time and the total totes that are handled denoted as ct and cnt. The last two parameters e_x denote the times at which new commands where received from a lift agent.

Actions

Action	*Data*	*Description*
result	$time \times time \times \mathbb{N}$	This communication action contains a message with the total totes processed and the consumed time to do so.
print_lat	*time*	This actions contains a message with the last action time.
doublecycle	$time \times time \times time \times lift\_id \times location$	This communication action contains a message indicating that a a transfer-in and transfer-out transfer occur simultaneously (both at location). The message also includes the time at which the totes arrived at the input conveyor and transfer-out point.
seqcycle	$time \times time \times time \times lift\_id \times location \times location$	This communication action contains a message indicating that a transfer-in and transfer-out transfer occur sequentially (at different locations. The message also includes the time at which the tote at the input arrived in the queue.
outcycle	$time \times time\ lift\_id \times location$	This communication action contains a message to process the tote from the transfer-out (at location), including the delivery to the output conveyor.
incycle	$time \times time \times lift\_id \times location$	This communication action contains a message to process the tote from the input conveyor and store it at the transfer-in point (at location). The message also includes the time at which the tote at the input arrived in the queue.
liftReady	$time \times lift\_id$	This communication action contains a message indicating that the lift platform is ready for the next command.

When all atomic actions regarding storage totes (totes arriving at the input conveyor) are executed, determined with the function *isHandlingInput*, and the delivery layer has no more totes in the queue then the *result*, *print_idle* and *print_lat* action are executed which contain the simulation results. Note that the *result* action always gets precedence over the other actions due to the 0 value for the time attribute.

$$
\begin{aligned}
&(\neg isHandlingInput(l1\_work) \wedge \neg isHandlingInput(l2\_work)) \rightarrow \\
&\quad s\_result(0, ct, cnt)\cdot \\
&\quad \textstyle\sum i1 : timer\_print\_idle(1, i1)\cdot \\
&\quad \textstyle\sum i2 : timer\_print\_idle(2, i2)\cdot \\
&\quad \textstyle\sum_{lat:time} r\_getlat(lat) \cdot print\_lat(lat)\cdot \\
&\quad mast(l1, l2, l1\_work, l2\_work, l1\_loc, l2\_loc, sin1, sin2, sout1, sout2, ct, cnt, e1, e2)
\end{aligned}
$$

The lift agent layer reports a command with one of the four *cycle* actions that informs the mast which totes must be retrieved and stored. The mast creates a list of atomic action commands that will perform the requested cycle. Here we will only discuss the *doublecycle* action but the other three actions are similar (see Appendix A.3 for the complete model). The list of atomic actions commands is stored in l_x_work and the internal status l_x of the mast is set to *standby*. The constant `CONV_POS` is explained later on this section.

$$
\begin{aligned}
&\textstyle\sum_{dest:location, id:lift\_id, pt:time, tin:time, tout:time} \\
&\quad r\_doublecycle(pt, tin, tout, id, dest)\cdot \\
&\quad mast(if(id = 1, standby, l1), if(id = 2, standby, l2), \\
&\qquad if(id = 1, [pair(\texttt{CONV\_POS}, loadinput), pair(dest, loadunloadtransfer), \\
&\qquad\quad pair(\texttt{CONV\_POS}, unloadoutput)], l1\_work), \\
&\qquad if(id = 2, [pair(\texttt{CONV\_POS}, loadinput), pair(dest, loadunloadtransfer), \\
&\qquad\quad pair(\texttt{CONV\_POS}, unloadoutput)], l2\_work), \\
&\qquad l1\_loc, l2\_loc, if(id = 1, tin, s1), if(id = 2, tin, s2), if(id = 1, tout, sout1), if(id = 2, tout, sout2), \\
&\qquad ct, cnt, if(id = 1, pt, e1), if(id = 2, pt, e2))
\end{aligned}
$$

If the internal status of the mast is *standby* (for one or both lift platforms) then the mast process executes the first atomic action from the list of commands for the corresponding lift platform(s). The *sendCmd* and *sendCmds* communication actions inform the motion manager of

the command. The internal status is changed to *busy* to indicate that the mast is now waiting for a signal from the lift platform layer that indicates that the atomic action command is executed.

$$
\begin{aligned}
&(is\_standby(l1) \land l1\_work \neq [] \land is\_standby(l2) \land l2\_work \neq []) \rightarrow \\
&\quad s\_sendCmds(min(e1, e2), head(l1\_work), head(l2\_work)) \cdot \\
&\quad mast(busy, busy, l1\_work, l2\_work, l1\_loc, l2\_loc, sin1, sin2, sout1, sout2, ct, cnt, e1, e2) \\
&+ \\
&(is\_standby(l1) \land l1\_work \neq [] \land \neg is\_standby(l2)) \rightarrow \\
&\quad s\_sendCmd(e1, 1, head(l1\_work)) \cdot \\
&\quad mast(busy, l2, l1\_work, l2\_work, l1\_loc, l2\_loc, sin1, sin2, sout1, sout2, ct, cnt, e1, e2)
\end{aligned}
$$

The motion manager layer signals the mast layer of the completion of the command with the *cmdDone* communication action. The mast layer then resumes with the next atomic action command in the list and repeats this until the list is empty, which is done by setting the internal status to *standby*. The *liftReady* communication action will inform the lift agent of the completion of the requested *cycle* once the atomic action command list is empty. Furthermore we set the internal status of the corresponding lift platform to *lidle*.

$$
\begin{aligned}
&\sum_{id:lift\_id} \\
&\quad \sum_{pt:time} r\_cmdDone(pt, id) \cdot (\\
&\qquad (id = 1 \land |l1\_work| = 1) \rightarrow (\\
&\qquad s\_liftReady(pt, 1) \cdot \sum_{e:lift\_action}(e = action(head(l1\_work))) \rightarrow \\
&\qquad mast(lidle, l2, [], l2\_work, dest(head(l1\_work)), l2\_loc, \\
&\qquad\quad sin1, sin2, sout1, sout2, \\
&\qquad\quad ct + if(is\_unloadtransfer(e) \lor is\_loadunloadtransfer(e), pt - sin1, if(is\_unloadoutput(e), pt - sout1, 0)), \\
&\qquad\quad cnt + if(is\_unloadtransfer(e) \lor is\_loadunloadtransfer(e) \lor is\_unloadoutput(e), 1, 0), pt, e2) \\
&\qquad) \\
&\qquad + \\
&\qquad (id = 1 \land |l1\_work| > 1) \rightarrow (\\
&\qquad\quad mast(standby, l2, tail(l1\_work), l2\_work, dest(head(l1\_work)), l2\_loc, \\
&\qquad\quad s1, s2, \\
&\qquad\quad ct + if(is\_unloadtransfer(e) \lor is\_loadunloadtransfer(e), pt - sin1, if(is\_unloadoutput(e), pt - sout1, 0)), \\
&\qquad\quad cnt + if(is\_unloadtransfer(e) \lor is\_loadunloadtransfer(e) \lor is\_unloadoutput(e), 1, 0), pt, e2) \\
&\qquad) \\
&\quad)
\end{aligned}
$$

5.2.6 Lift agent

This layer determines the next action for an idle lift platform following the algorithm as specified by VI.

Sorts

Sort	*Type*	*Description*
lift_state	(*Bool*,*location*)	This sort contains a tuple with the status of the lift platform.

The sort *lift_state* is a tuple that contains the necessary information for a lift agent to determine the next action for the lift platform. These elements can be accessed with *inMotion* and *loc* which denotes if the lift platform is idle (so not in motion) and at which location it is stationed.

The process of this layer is defined as follows:

$$lift\_agent(state : lift\_state, id : lift\_id)$$

The parameter *state* contains the location at which the lift platform *id* is located after a command from this layer is executed. A command from this process is made up from several atomic action commands that are executed in a given order.

Actions

Action	*Data*	*Description*
outReady	*time* × *lift_id* × *location* × *time*	This communication action contains a message indicating that a transfer-out (at location) must be retrieved next including the arrival time the tote at the transfer-out point.
inReady	*time* × *time* × *lift_id*	This communication action contains a message indicating that a tote is ready for retrieval at the input conveyor, including the arrival time of tote in the input queue.
transFree	*time* × *location* × *lift_id* × *location*	This communication action contains a message with the transfer-in point location to store the tote from the input conveyor.

If the lift platform is located at the conveyor location (denoted by the constant `CONV_POS`) then it will follow the strategy as described in Section 2.2.1. The *outReady* communication action indicates that a retrieval-action is possible whereas the *inReady* communication action indicates that a storage-action is possible. The retrieval-action will get priority as its first argument is 0. The four cycle communication actions (*double,seq,out* and *in*) send a message to the mast layer containing the locations from where to retrieve the totes. These cycles respectively denote the simultaneous-action, the sequential-action, the retrieval-action and the storage-action (see Section 2.2.1).

$$
\begin{array}{l}
(\neg inMotion(state) \wedge loc(state) = \texttt{CONV\_POS}) \rightarrow (\\
\quad \sum_{nextout:location, tout:time} \\
\quad r\_outReady(0, id, nextout, tout) \cdot (\\
\qquad \sum_{tin:time, pt:time} r\_inReady(pt, tin, id) \cdot \\
\qquad \sum_{nextin:location} s\_transFree(pt, NO\_LOC, id, nextin) \cdot \\
\qquad (nextin = nextout) \rightarrow \\
\qquad\quad s\_doublecycle(pt, tin, tout, id, nextout) \cdot lift\_agent(pair(true, nextout), id) \\
\qquad \diamond \\
\qquad\quad s\_seqcycle(pt, tin, tout, id, nextout, nextin) \cdot lift\_agent(pair(true, nextin), id) \\
\qquad + \\
\qquad \sum_{pt:time} r\_getlat(pt) \cdot \\
\qquad s\_outcycle(pt, tout, id, nextout) \cdot lift\_agent(pair(true, nextout), id) \\
\quad) \\
\quad + \\
\quad \sum_{tin:time, pt:time} r\_inReady(pt, tin, id) \cdot \\
\quad \sum_{nextin:location} s\_transFree(pt, \texttt{NO\_LOC}, id, nextin) \cdot \\
\quad s\_incycle(pt, tin, id, nextin) \cdot lift\_agent(pair(true, nextin), id) \\
)
\end{array}
$$

However if the current location lift platform differs from the conveyor location then the lift agent process will prefer to get a tote from a transfer-out point. This tote will then be delivered to the output conveyor. Thus afterwards the lift platform will be located at the conveyor location such that it can try to perform the above strategy. When totes are not available at any transfer-out point, the lift agent will try to process a tote from the input conveyor if available. This last choice is necessary as otherwise the system could be waiting indefinitely for a retrieval-action.

$$
\begin{array}{l}
(\neg inMotion(state) \wedge loc(state) \neq \texttt{CONV\_POS}) \rightarrow (\\
\quad \sum_{nextout:location, tout:time} \\
\quad r\_outReady(0, id, nextout, tout) \cdot \\
\quad \sum_{pt:time} r\_getlat(pt) \cdot \\
\quad s\_outcycle(pt, tout, id, nextout) \cdot lift\_agent(pair(true, nextout), id) \\
\quad + \\
\quad \sum \_tin : time, pt : time \\
\quad r\_inReady(pt, tin, id) \cdot \\
\quad \sum_{nextin:location} s\_transFree(pt, \texttt{NO\_LOC}, id, nextin) \cdot \\
\quad s\_incycle(pt, tin, id, nextin) \cdot lift\_agent(pair(true, nextin), id) \\
)
\end{array}
$$

Once the command is completed then the mast layer will signal the lift agent process with the *liftReady* communication action that the next command must be determined. The process parameters are updated to reflect that the lift platform is idle and it stores the current location at which it is.

$$inMotion(state) \rightarrow \\ \sum_{pt:time} r\_liftReady(pt, id) \cdot lift\_agent(pair(false, loc(state)), id)$$

5.2.7 Shuttle agent

This layer represents the shuttles which are represented as simple as possible in this model, because of the reasons outlined in Section 2.2.2.

The process that represent this layer is defined as:

$$shuttle\_agent$$

Actions

Action	*Data*	*Description*
shuttleUnload	*location*	This communication action contains a message indicating that a tote is transferred from the shuttle to the transfer-out point (at location).
shuttleLoad	*location*	This communication action contains a message indicating that a tote is transferred from the transfer-in point (at location) to the shuttle.

This process has two communication actions: *shuttleUnload* signals the transfer agent layer that a tote is transferred from a shuttle to a transfer-out point at the given location whereas *shuttleLoad* signals the transfer agent layer that a tote is retrieved from a transfer-in point at the given location by a shuttle. The function *inRng* limits the domain of the location summation and only allows locations at which transfer points are physically present. The constants `LOCS` and `BOTTOM` denote the lowest and highest possible locations in the HDS system.

$$\sum_{l:location}(inRng(l, \texttt{LOCS}, \texttt{BOTTOM})) \rightarrow (\\ s\_shuttleUnload(l) \\ + \\ s\_shuttleLoad(l) \\ ) \cdot shuttle\_agent$$

5.2.8 Delivery

This layer represents the input conveyor which contains a pre-generated finite list of absolute times at which totes arrive in the input queue. It is responsible for the totes that arrive at the input conveyor.

The process that represents this layer is defined as:

$$delivery(at : List(Nat), na : Nat)$$

The parameter *at* is a list of inter-arrival times of totes and *na* holds the next arrival time of the first tote in the queue.

Action

Action	*Data*	*Description*
hasInput	*time* × *time*	This communication action contains a message with the time that a tote has arrived in the queue of the input conveyor.

This process signals the arrival of a new tote at the input conveyor to the transfer agent layer with the *hasInput* communication action. After this it will wait until the communication action *loadedInput* is is signaled by the lift platform layer. It will continue to do so until the list *at* is empty.

$$
\begin{aligned}
&(at \neq []) \rightarrow (\\
&\quad \textstyle\sum_{t\_in:Nat}(t\_in \geq na + at.0) \rightarrow \\
&\qquad r\_getlat(t\_in) \cdot s\_hasInput(t\_in, na + at.0) \cdot \\
&\qquad \textstyle\sum_{t:time} r\_loadedInput(t) \cdot \\
&\qquad delivery(tail(at), na + at.0) \\
&\quad + \\
&\quad s\_tick(\texttt{LOWPRIOR}, na + at.0) \cdot delivery(at, na) \\
)
\end{aligned}
$$

Once the last arrival time has been signaled to the lift agent layer then it will wait for the mast layer to finish processing the tote. The mast layer will then provide the results of the simulation with the result communication action.

$$
\begin{aligned}
&(at = []) \rightarrow \\
&\quad \textstyle\sum_{ct,cnt} \cdot r\_result(0, ct, cnt) \cdot delivery(at, na)
\end{aligned}
$$

5.2.9 Transfer agent

This layer is responsible for the administration of the buffers from the transfer point and which totes are ready for retrieval and storage.

Sorts

Sort	*Type*	*Description*
loctime	(location,time)	Tuple containing the arrival time of a tote on transfer-out at the given location.

This tuple type is used to store the arrival time of totes on transfer-out points. The time is set to the moment at which a *shuttleUnload* action occurs, which can occur at any time in our model.

The process that represents this layer is defined as:

$$
\begin{aligned}
&transfer\_agent(il : Set(location), ol : Set(location), \\
&\qquad\qquad ic : Set(location), rl : List(retrieval\_state), \\
&\qquad\qquad t\_in : time, t\_na : time, rl\_t : List(loctime))
\end{aligned}
$$

The parameter *il* represents the collection of transfer-in point locations which are occupied by a tote. The parameter *ic* represents the collection of transfer-in points which are already claimed by a lift platform. However the lift platform still needs to store the tote on this transfer-in point. The parameter *ol* represents the collection of transfer-out point locations that are occupied by a tote. The parameter *rl* holds a list of transfer-out locations with totes, ordered as they arrived per shuttle. The parameters *t_in* and *t_na* represent: the time that the delivery layer signaled the tote as ready (*hasInput*)) and the time at which the tote arrived in the queue. The last parameter *rl_t* contains the arrival times of totes at transfer-out points.

Shuttles retrieve and store totes on the transfer points as shown in the shuttle layer. The transfer agent process will update the corresponding parameter which the communication actions *shuttleLoad* and *shuttleUnload* are performed. If a tote is retrieved from a transfer-in point then

the process updates the *il* parameter by removing the corresponding location from the collection. If a tote is stored on a transfer-out point then the process updates the *ol* parameter by adding the corresponding location to the collection. Also the new tote is added to the end of the *rl* list and marked unclaimed, as a lift platform is yet to be assigned to it. Furthermore we store the current time at which a tote has arrived at the transfer-out point, which is stored in the *rl_t* list. Note that we do not dictate when totes arrive at the transfer-out points but allow it be delivered at any time.

$$
\begin{array}{l}
\sum_{l:location} \\
((l \notin ol) \wedge inRng(l, \texttt{LOCS}, \texttt{BOTTOM})) \rightarrow \\
\quad r\_shuttleUnload(l) \cdot \sum_{lat:time} r\_getlat(lat) \cdot \\
\quad transfer\_agent(il, ol + l, ic, rl \triangleleft pair(l, false), t\_in, t\_na, rl\_t \triangleleft pair(l, lat)) \\
+ \\
\sum_{l:location} \\
(l \notin il) \rightarrow \\
\quad r\_shuttleLoad(l) \cdot transfer\_agent(il - l, ol, ic, rl, t\_in, t\_na, rl\_t)
\end{array}
$$

The transfer agent process handles the *transFree* communication action which requests if a certain transfer-in point is free. The function *ibCheck* determines if the given location is free and returns an alternative if not. If no suitable locations remain then the function returns the constant `NO_LOC`. The chosen transfer-in point is declared as claimed by adding its location to the *ic* parameter.

$$
\begin{array}{l}
\sum_{reqloc:location, lid:lift\_id, pt:time, inloc:location} \\
(inloc = ibCheck(lid, reqloc, il, ic)) \rightarrow \\
\quad r\_transFree(pt, reqloc, lid, inloc) \cdot \\
\quad transfer\_agent(il, ol, ic + inloc, rl, t\_in, t\_na, rl\_t)
\end{array}
$$

The transfer agent process will send a message to the lift agent with the *outReady* communication action once a tote is available. The function *getNextout* determines which tote is next in line to be retrieved by the lift platform. The chosen tote will be marked as claimed in the *rl* tuple list by the function *setClaimed*.

$$
\begin{array}{l}
\sum_{out:location}(out = getNextout(1, rl) \wedge out \neq \texttt{NO\_LOC}) \rightarrow \\
\quad s\_outReady(0, 1, out, getOuttime(rl\_t, out)) \cdot transfer\_agent(il, ol, ic, setClaimed(rl, out), t\_in, t\_na, rl\_t)
\end{array}
$$

The transfer agent process handles the *tsinput* and *tsoutput* communication actions from the lift platform layer that stored or retrieved a tote from a transfer point. The *ol* collection and *rl* list are updated by the removing the location of the transfer-out point from where the tote is retrieved. The function *rmRelease* removes the tuple element from the list. The stored time for the arrival of the tote at the corresponding transfer-out point is removed with the *rmOuttime* function.

$$
\begin{array}{l}
\sum_{loc:location, pt:time} r\_tsoutput(pt, loc) \cdot \\
transfer\_agent(il, ol - loc, ic, rmRelease(rl, loc), t\_in, t\_na, rmOuttime(rl\_t, loc)) \\
+ \\
\sum_{loc:location, pt:time} r\_tsinput(pt, loc) \cdot \\
transfer\_agent(il + loc, ol, ic, rl, t\_in, t\_na, rl\_t)
\end{array}
$$

The transfer agent process handles the *hasInput* communication action from the delivery layer and stores the queuing time of the tote in the parameter *t_na*. It stores the moment at when the *hasInput* communication action is performed (when the tote was signaled as available) in the parameter *t_in*.

$\sum_{pt:time,na:time} r\_hasInput(pt, na) \cdot transfer\_agent(il, ol, ic, rl, pt, na, rl\_t)$

The transfer agent process signals the lift agent layer if a new tote is ready at the input conveyor with the *inReady* communication action but only if a transfer-in point is free and reachable for the lift platform that will pick it up. This is determined with the function *canStore*. The constant `LOWPRIOR` indicates that no totes are ready at the input conveyor so we set the *t_in* variable to this value after signaling the lift platform layer such that it will not be signaled again until retrieval is completed.

$((t\_in \neq \texttt{LOWPRIOR}) \wedge canStore(1, il, ic, BOTTOM)) \rightarrow$
$s\_inReady(t\_in, t\_na, 1) \cdot$
$transfer\_agent(il, ol, ic, rl, \texttt{LOWPRIOR}, t\_na, rl\_t)$

5.2.10 Initial state

The state of the HDS system at the moment that is switched on is known as the initial state. The initialization of each process is discussed in this section. All these processes run in parallel with each other. The complete initialization block of the mCRL2 model can be found in Appendix A.3.

The monitor process is straightforward, starting at time unit 0.

$monitor(0)$

Two *liftplatform* processes are instantiated, one for each lift platform in the HDS system. The identifiers for the lift platforms are set to 1 and 2 during the instantiation. The lower lift platform starts at the bottom of the system which is denoted by the `BOTTOM` constant. The upper lift platform starts at the top of the system which his denoted by the `LOCS` constant. The destination parameter is also set to these values and the busy parameter is set to false as the lift platform is idle. The action parameter is set to *move* but the actual value does not matter at this point as there is no active order at this point.

$liftplatform(1, \texttt{BOTTOM}, \texttt{BOTTOM}, false, move, 0, 0)$
$liftplatform(2, \texttt{LOCS}, \texttt{LOCS}, false, move, 0, 0)$

The stepper process must match the initial setup of the lift platform processes. The first two parameters denote the locations of the two platforms: `BOTTOM` and `LOCS`. The directions of the lift platforms is set to *neutral*, the last action times to 0, the current destination to `NO_LOC` and the internal state of the motors to *idle* as both lift platforms are initially idle.

$stepper(\texttt{BOTTOM}, \texttt{LOCS}, neutral, neutral, 0, 0, \texttt{NO\_LOC}, \texttt{NO\_LOC}, idle, idle)$

The motion manager process must also match the setup of the lift platforms. The first two parameters denote the current locations of the lift platforms and the action that they are performing. The internal state for the motion manager process are set to *lidle* for both lift platforms. The priority token is assigned to the lower lift platform, namely 1. The temporary detour lift command parameter is can be set to a random value as we start without any platforms performing a detour. The parameter *dt* is set to 0 to reflect that no detour is being performed. The parameters that indicate the start time of each motor are set to 0.

$motion\_mng(pair(\texttt{BOTTOM}, move), pair(\texttt{LOCS}, move), lidle, lidle, 1, pair(\texttt{NO\_LOC}, move), 0, 0, 0)$

The mast process is also instantiated to match the rest of the processes. The internal status of the mast for both lift platform is set to *lidle* as the lift platforms are idle at start-up. The list of actions commands are set to empty as no work is determined yet at start-up. The next two parameters indicate the location of the two lift platforms and the final eight parameters are set to 0 which are all used for the simulation of the HDS system.

$mast(lidle, lidle, [], [], \texttt{BOTTOM}, \texttt{LOCS}, 0, 0, 0, 0, 0, 0, 0, 0)$

We also instantiate two lift agent processes: one that determines the next course of action for the lower lift platform and one that determines the next course of action for the upper lift platform (see the last parameter). The first parameter indicates that the lift platforms are currently not moving and their current location is set to match the location of the lift platform processes.

$lift\_agent(pair(false, \texttt{BOTTOM}), 1)$
$lift\_agent(pair(false, \texttt{LOCS}), 2)$

The shuttle agent does not contain any data parameters and is simply instantiated.

$shuttle\_agent$

The transfer agent process is instantiated with all transfer points empty. The last two parameters are set to LOWPRIOR and 0. The first one indicates that no totes are ready for retrieval at the input conveyor. The second one indicates the arrival time of the tote in the queue, now set to 0 but any value would do.

$transfer\_agent(\{\}, \{\}, \{\}, [], \texttt{LOWPRIOR}, 0, [])$

The final process to instantiate is the delivery process. This process contains the list of simulated inter-arrival times of totes that arrive in the queue at the input conveyor. The second parameter contains the next arrival time of a tote: the accumulated value of the first inter-arrival time in the list and the ones that are already processed. We start at time 0.

$delivery([\ldots], 0)$

5.3 Time-priority conversion

The HDS system model, as discussed in section 5.2, uses absolute times to denote the arrival moments of totes and to model the behavior of the system. By instantiating the delivery process with a random list of inter-arrival times we can perform simulations of the system on the model, which raises two issues.

Instantiating the model with a random list of inter-arrival times produces a model of which the behavior is according to a single possible input sequence of totes, but we want to verify the behavior for all possible input sequences instead. Furthermore using absolute times in the model leaves us with a statespace of the system that is not viable in terms of memory requirements to validate the requirements with the mCRL2 toolset, as specified in Chapter 3: Table 3.2.

In this section we explain how we transformed the timed model to an untimed model to overcome these problems, although these modification change the system behavior to a certain extent. We discuss these modifications and argue the impact of the changes.

Handling a tote starts at a certain absolute moment in time and requires an amount of time to complete, depending on the distance the platforms need to travel to the required destinations. A following tote arrives after this one and repeats this pattern, but the time values are larger than before. For the requirement verification these absolute time values are not interesting though they do control the sequence in which actions are performed. We introduce a number of priorities that are used instead of the time attribute which preserve the pattern but remove the need for the time values. Priorities are denoted by P_x, for all values satisfying $x \geq 0$. An action with priority P_x has precedence over an action with P_y, for all values satisfying $x < y$. The choice of the priorities is such that the correct order is kept, while also the statespace is limited to a size at which property validation is possible.

The actions used in the model can be divided in actions that take time in the physical system, such as movement of the lift platforms, and actions that represent internal HDS controller commands

(e.g. the *sendLift* action). We assume that the internal controller commands take zero time; furthermore the shuttle system can perform actions without any time restriction, allowing all possibilities of the shuttle system throughout the system. These actions have to be assigned a priority in such a way that the behavior of the untimed HDS system's model remains as close as possible to the timed model. The prioritized 'untimed' model is included as Appendix A.4.

In the first model transformation step all the monitor process related actions and processes are removed. Next we remove the time attribute from the *hasInput* which allows new totes to arrive at the input conveyor at all possible moments, thus allowing all possible input sequences. Next we make sure that totes that are ready to be processed are handled before any other timed action by assigning the *outReady* and *inReady* the highest priority P_2 in our model. Priority P_1 does not exist in our model but was assigned to *hasInput* during initial statespace testing. The priority P_2 is also assigned to the action *transFree* which is closely related to handling the storage of a tote (see the lift agent process).

As *outReady* and *inReady* actions have equal precedence (unlike the timed model), the statespace includes choices of the lift agent that are not in the timed model. This means that we validate more than strictly necessary, including the part that we are interested in.

All other internal controller actions are assigned priority P_3, which controls the distribution and execution of commands that are required to handle a tote. The real motion occurs when the lift is moving to a location or when a tote is transported from or onto a lift platform. In the real HDS system it is possible that one platform is moving while the other is transporting a tote from or onto the lift platform or both platforms are moving at the same time. The stepper process makes sure that both platforms will move together if they are both in motion, so we assign lowest priority P_5 to the *step* action. The transportation of totes from and to the lift platforms in the timed model will be executed at the same time as the *step* action due to the time attribute but here this will not be the case. For this reason we assign priority P_4 to the actions *liftLoad*, *liftLoadInput*, *liftUnload* and *liftUnloadOutput* actions, such that they are executed once it is possible instead of allowing possibly many step actions. We also set the the number of transfer buffers to three, the lowest possible where the HDS system still functions as intended. We also place the buffers right above each other so one step from the stepper results in arriving at the next transfer buffer; placing the buffers further apart introduces extra step actions and results in a statespace that is too large to validate our functional requirements.

Finally we assign priority P_5 to the actions *shuttleLoad* and *shuttleUnload*, since allowing these actions to be executed at any possible moment (as in the timed model) generates a statespace of such large proportions that it is not viable for the validation of the requirements. By allowing the actions to only occur during the motion of the lift platforms (performing step actions), we still stay close to the actual behavior of the system. In the real system totes from the racks also arrive when the platforms are in motion, however we do not allow it during the transportation of totes from and to the lift platforms which is not the true behavior. Of course the shuttle actions can also occur if nothing else is going on in the system, which is also possible in the physical HDS system.

5.4 HDS simulation

In this section we present the results of the HDS systems simulation and discuss them. Two different simulations have been performed: in the first version we only allow storage totes (input) to arrive at the system and in the second version we allow both storage and retrieval totes (input and output) to arrive at the system. The simulation results are calculated according to the approach of the parallel lift platform system from Chapter 4, see Formulas 4.18, 4.19 and 4.20.

Figures 5.3, 5.4 and 5.5 display the results for the simulation in which only storage totes are processed, meaning that no retrieval totes arrive at the system. We compare the results of this simulation against the *M/D/2* system to see the difference in performance for these two systems.

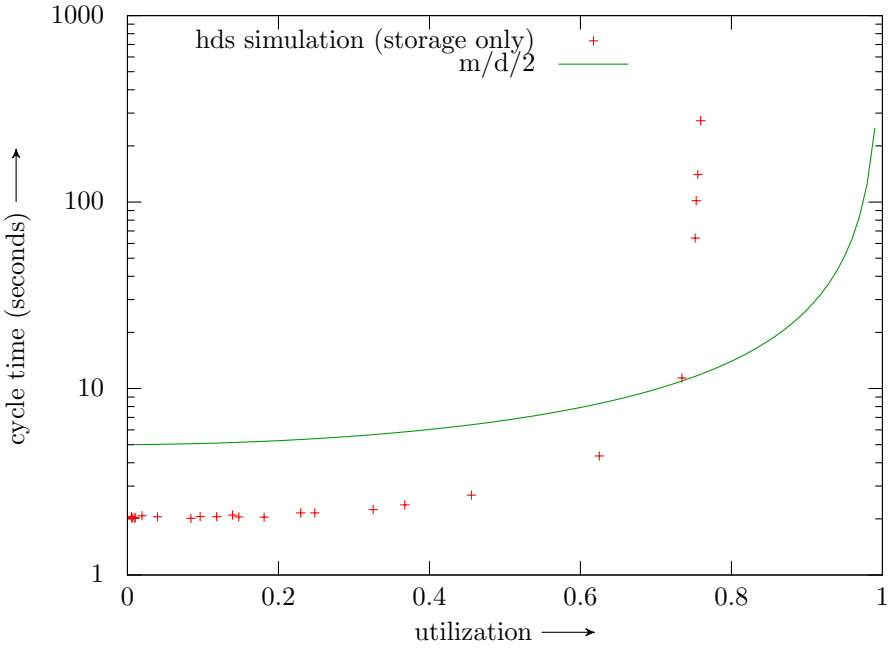


Figure 5.3: HDS system cycle time

Figure 5.3 shows that the average time required to handle storage totes is less than in the *M/D/2* case up to approximately 77 percent. However we should see this 77 percent mark as the maximum utilization of the HDS system, thus we should conclude that the cycletime is always lower than with the *M/D/2* case. The fact that the HDS system has an utilization maximum of about 80 percent, at which the cycle time increases steeply, can be explained by the fact that the two lift platforms sometimes have to wait for each other. This creates a certain amount of unavoidable idle time for the lift platforms, thus decreasing the 'maximum' utilization. Note that we have to take care that shuttles must not always immediately retrieve a tote from a transfer-in point. Recall from the model description in Chapter 5 that shuttles remove totes from transfer-in points at any given time, assuming that one is available on it. Combining this fact with the random prioritized walk option that we used to perform the simulations, we know that totes can remain on a transfer-in point for a certain period of time before a shuttle picks it up because other actions are available for selection as well. This behavior is the same as with the real HDS system thus providing us a realistic simulation where totes cannot always be placed on the nearest transfer-in point because it has not yet been emptied by a shuttle. Note that maximum utilization value of 77 percent is approximately in line with real-life simulation test of the HDS system at VI.

The throughput of the HDS system, the number of totes that go through the system per minute, is greatly improved with a factor of about four in comparison with the *M/D/2* system as depicted in Figure 5.4. We also see that the utilization limit is about 80 percent for the same reasons as mentioned by the cycle time analysis. However we do not believe that this system is capable of processing up to 52 totes per minute as this is too fast, somewhat more than 1 second per tote. This leaves us to believe that we still need to calibrate the model of the HDS system someday.

Finally we look at the work in progress, the average length of the totes queue, which is displayed in Figure 5.5 that shows that the average queue length (WIP) is slightly larger than it is the case in the *M/D/2* system. However around a utilization of about 77 percent we see that the queue length is only about 10 totes, which is still acceptable as the throughput and the cycle time are both greatly improved in comparison. The calibration issues also reflect on these results thus we must conclude that these results are not perfect and are probably too optimistic.

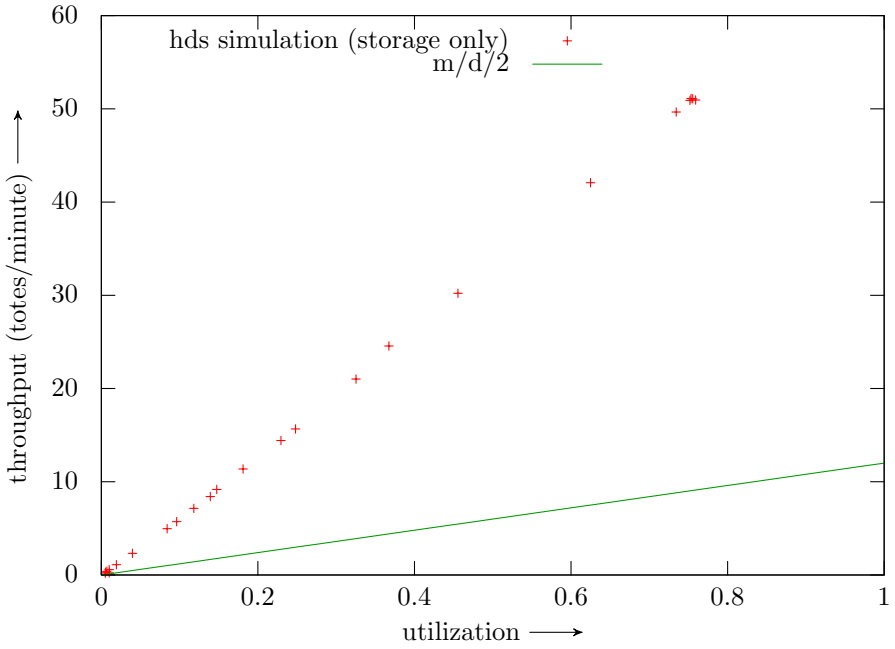


Figure 5.4: HDS system throughput

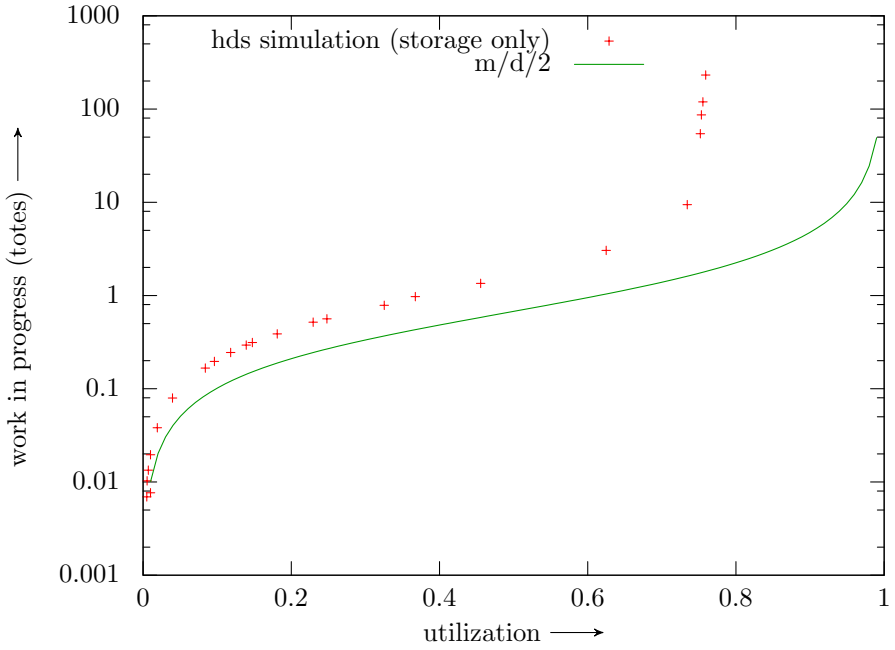


Figure 5.5: HDS system work in progress

The simulation results of the system in which retrieval totes (output) also arrive at the system are depicted in Figures 5.6 and 5.7. In both figures we see that all the data points reside in the upper limits of the utilization as if the system always has work to perform. Although this seems strange at first sight it is actually what is expected from this model. We allow retrieval totes to arrive at the system at any time (see the action *shuttleUnload* in Chapter 5) which results in lots of retrieval totes arriving at the system when storage totes are not available because retrieval totes can be added instead of jumping to the time at which a new storage tote is available (see the *tick* action). The probability that the *tick* action is selected is $\frac{1}{6}$ (five *shuttleUnload* actions and one *tick* action) so most of the time it probably adds new retrieval totes. The shuttle issues

still need to be solved by implementing a similar delivery process at the shuttle process. There might also be other calibration problems with the model that need to be sorted out someday.

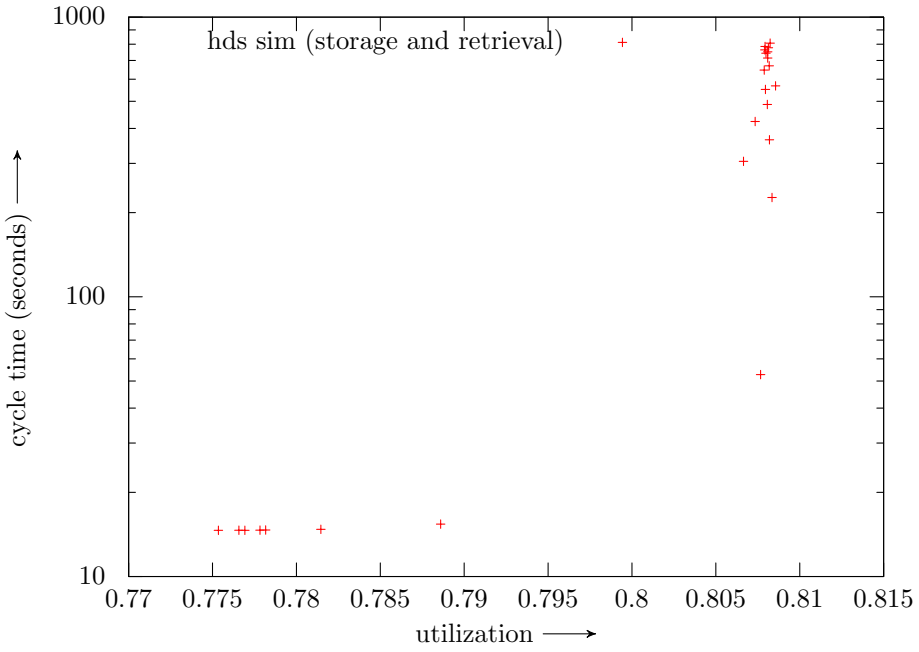


Figure 5.6: HDS system cycle time

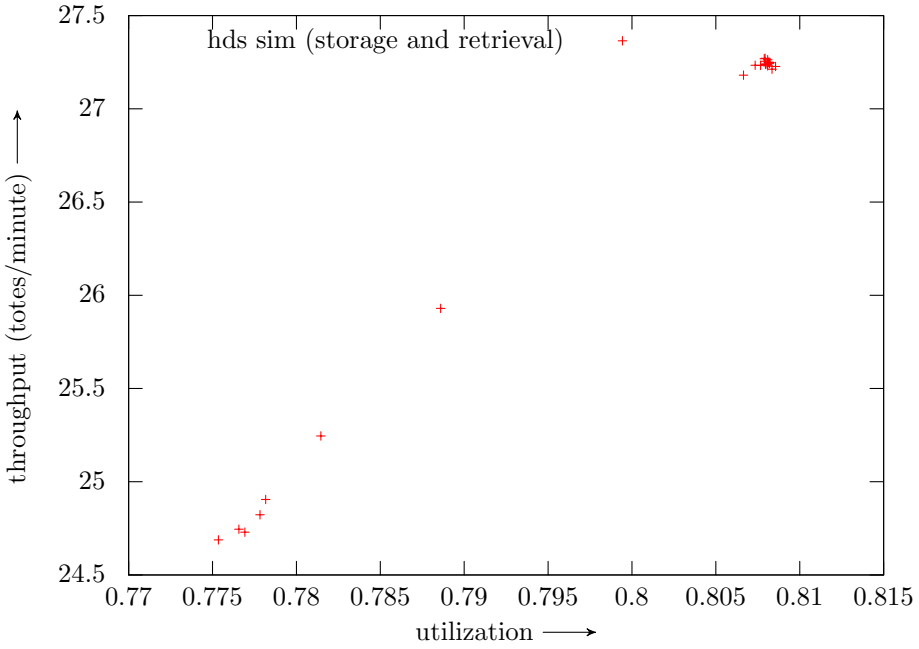


Figure 5.7: HDS system throughput

We can compare these results with the previous simulation, based on the inter-arrival time instead of the utilization, and see how the system behaves when both storage and retrieval totes are processed by the system, where we assume that retrieval totes are always available if no storage totes are available (which is likely mostly the case). Figures 5.8, 5.9 and 5.10 show the comparison between both simulation versions. We selected the data points that are of the most interest to us and display only these, thus allowing more detail in the graph. The remaining data points (those left out) have almost identical results as the rightmost data point (drawn at the mean inter-arrival time of 50 seconds).

In Figure 5.8 we see that the cycle time of a tote remains relatively large when both storage and retrieval totes are processed together in comparison to only handling storage totes. The cycle time becomes acceptable with mean inter-arrival time in the area of about 20 seconds for storage totes, whereas a mean value of 10 seconds or less increases the cycle time quickly to generally unacceptable values. When only processing storage totes we find that the cycle time decreases much faster thus allowing totes to be *processed* (not arrive) at a faster rate. When processing both types of totes (storage and retrieval) we see that we always need a minimal processing time of about 15 seconds per tote. This once again confirms our beliefs that the calibration of the system is not yet correct because we would aspect that retrieval and storage of totes should also be possible in less time, when both totes are stored close to the conveyor belts.

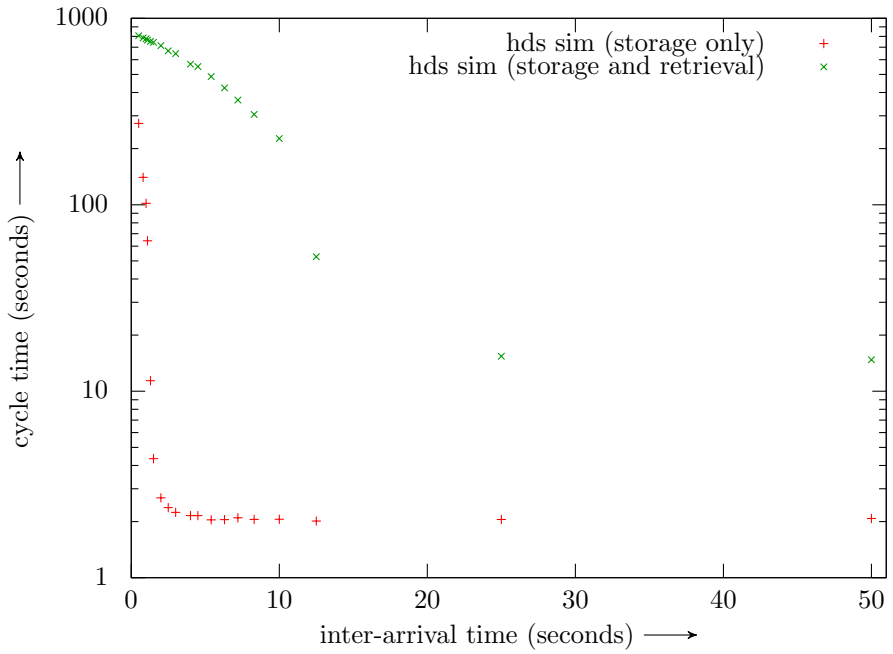


Figure 5.8: HDS system cycle time

The throughput, as depicted in Figure 5.9, shows that the maximal throughput decreased but the overall throughput is now about constant at a value of 27 totes per minute (see Figure 5.7). Note that the throughput at high mean inter-arrival times is relatively large in comparison to the other simulation because retrieval totes keep arriving if no storage totes are available (side-effect of the model). Now assume that the systems model side-effect does not exists. Then the throughput would decrease as the system handles less totes per minute, hence we can conclude that the best possible throughput is about 27 totes per minute. We also see that times below a mean inter-arrival time of 4 seconds gives the storage totes simulation version the upperhand in terms of throughput.

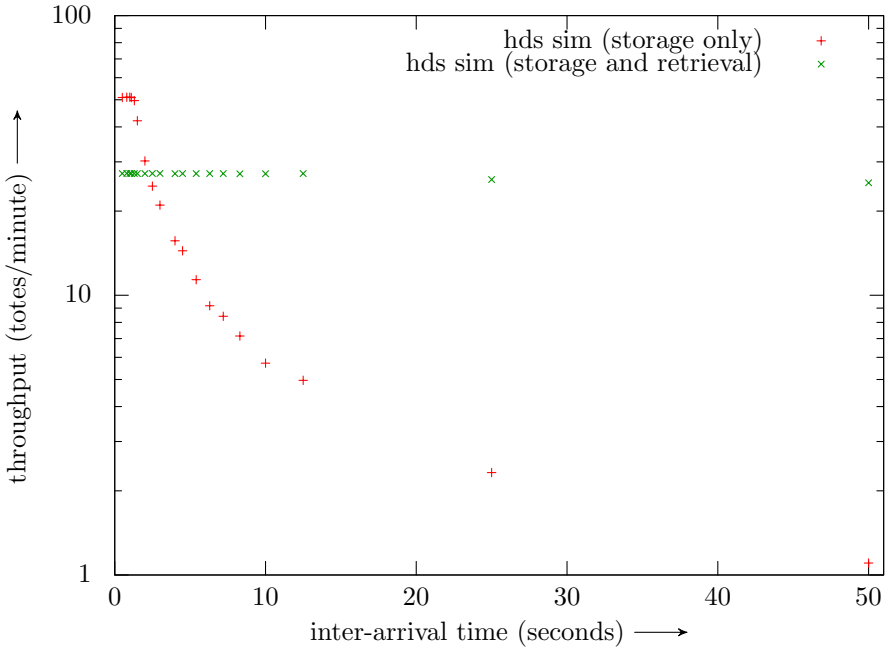


Figure 5.9: HDS system throughput

The work in progress, depicted in Figure 5.10, displays that the storage and retrieval tote simulation version, has a large average queue length for a relatively large inter-arrival time in comparison with the storage simulation version. Also here the work in progress becomes generally acceptable around the mean inter-arrival area of 20 seconds where the average queue length is between 10 and 20 totes long. Note that the results at the higher inter-arrival times are heavily influenced by the models side-effect and could in reality be lower.

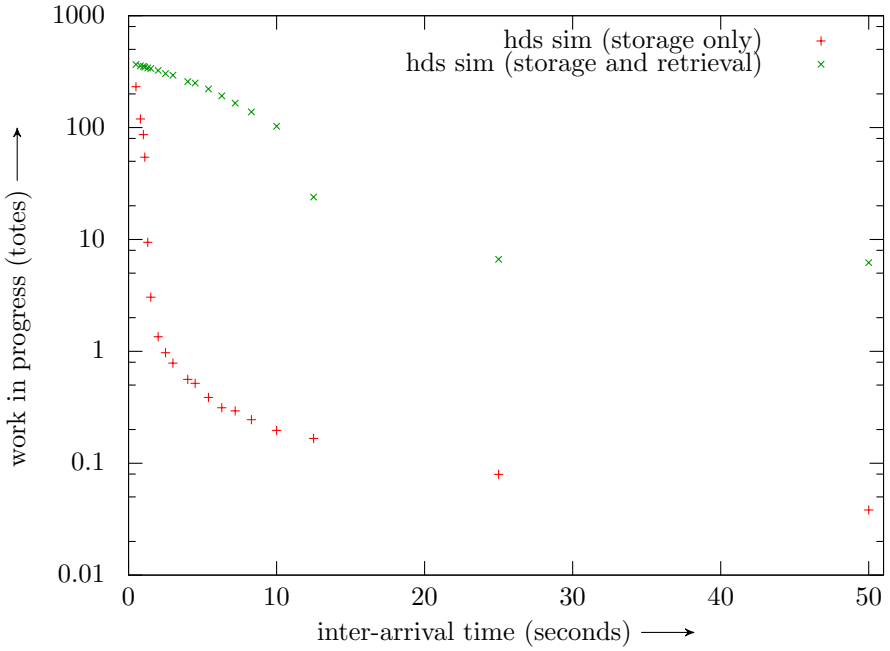


Figure 5.10: HDS system work in progress

Considering all these comparisons we can conclude that if the inter-arrival times of storage totes lies somewhere in the range from 10 to 20 seconds that the HDS system exhibits a good

performance. We have a realistic average queue length and a cycle time that can be considered good. What is actually considered good and realistic will eventually depend on the hardware and system demands; for example: an average queue length of 150 is considered acceptable. Furthermore we have seen that the storage only simulation version performance is better than the *M/D/2* system as was expected since the buffer system is available in the HDS system. However we strongly believe that the model of the HDS requires more calibration, because the results of the simulations are too optimistic and are not always plausible. Especially the simulation in which retrieval and storage of totes is applicable leaves us with results that are unlikely to be correct.

Chapter 6

Modal μ-calculus

In this chapter we will introduce modal μ-formulas with which we can describe properties of reactive systems like the HDS system's controller. In Section 5.1.1 we introduced the notion of an action that can be performed by a reactive system. With modal μ-formulas we are able to express properties in terms of sequences of actions (see Section 5.1.2) which are validated against the system's model. When a validation holds then we can conclude that the property is valid in all possible states of the system, thus always. Although modal μ-formulas allow a great deal of expressiveness, they are quite difficult to formulate and understand. In this chapter we will first introduce the the modal μ-calculus language, starting with the basic elements of the language and work towards the most advanced elements that are at our disposal.

The requirements from Chapter 3 are translated to modal formulas. Finally we explain how the 'prioritized walk' option from the `lps2lts` tool is incorporated in a modal formula and apply this technique to the modal formulas of the requirements.

6.1 The modal μ-calculus language

Hennessy-Milner logic [4] is the underlying model logic for this language which syntax is given by the following BNF grammar:

$\phi ::= \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a]\phi$

The modal formula *true* is valid in all possible states as in contradiction to the modal formula *false*, which is not valid in any state. The logic operators $\wedge$ (and), $\vee$ (or) and $\neg$ (not) have their usual meaning. So the formula $\phi_1 \wedge \phi_2$ is valid wherever both ϕ_1 and ϕ_2 holds, $\phi_1 \vee \phi_2$ is valid wherever either ϕ_1 or ϕ_2 holds and $\neg\phi$ is valid if the negation of ϕ holds. The two modality operators, the diamond modality and the box modality, are somewhat more difficult.

The diamond modality $\langle a \rangle \phi$ is valid whenever an a action can be done such that ϕ is valid after the a action is performed. The box modality $[a]\phi$ is valid when for every a action that can be done, ϕ is valid. So for example the formula $\langle a \rangle \langle b \rangle \mathit{true}$ expresses that it is possible to do an a action followed by a b action. The formula $[a]\langle b \rangle \mathit{true}$ expresses that after all a actions it is possible to do a b action.

Let's look at the labeled transition system in Figure 6.1 to get a better understanding of the modality operators. In the left transition system $\langle a \rangle \phi$ is valid since there is a transition path that leads to ϕ after doing an a action. However this is not the case for all a actions thus $[a]\phi$ is not valid. In the second transition system the validity is reversed. There is no a action possible at all, so certainly no state can be reached in which ϕ holds. However for all possible a action, which are none, it is possible to reach a state in which ϕ holds, so the box modality formula does hold since there are no paths that violate this property. Both formulas are valid in the third transition system. For all a actions (which is exactly one) we end up in a state in which ϕ is valid. In the final and fourth transition system neither formula holds as all a actions (which is exactly one) lead to a state in which ϕ is invalid.

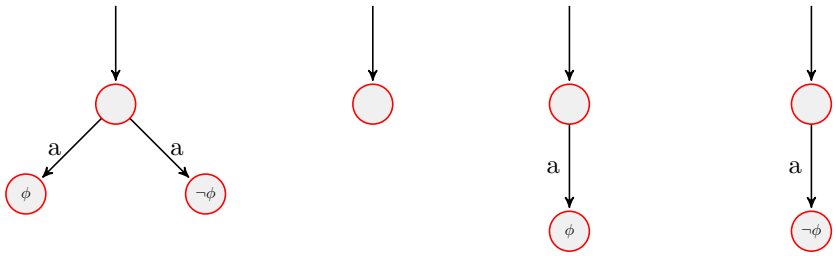


Figure 6.1: Example transition systems for diamond and box modality

To allow more than just a single action in a modality formula the Hennessy-Milner logic is extended with regular formulas within modalities. Regular formulas are based on action formulas which are defined as:

$\alpha ::= a \mid \mathit{true} \mid \mathit{false} \mid \overline{\alpha} \mid \alpha \cap \alpha \mid \alpha \cup \alpha$

Action formulas define a set of actions. So the formula a defines the set with only one action namely a. The formula *true* defines the set of all actions whereas the formula *false* defines the empty set. The operators $\cap$ and $\cup$ denote the intersection and union of sets. The notation $\overline{\alpha}$ denotes the complement of the set α with respect to the set of all actions. Let α be the set of actions, then the definition of modalities with action formulas in them is:

$$\langle\alpha\rangle = \bigvee_{a\in\alpha} \langle a\rangle\phi \qquad\qquad [\alpha] = \bigwedge_{a\in\alpha} [a]\phi$$

Action formulas are extended to regular formulas to allow the use of sequences of actions. Let α be an action formula, then the definition of a regular formula is as follows:

$R ::= \epsilon \mid \alpha \mid R \cdot R \mid R + R \mid R^* \mid R^+$

The formula ϵ represent the empty sequence of actions. The $\cdot$ operator concatenates two sequences of actions whereas the $+$ operator allows the choice between two sequences of actions. The R^* and R^+ modal formulas allow iterative behavior. The first denotes a repetition of a sequence of actions. The second one denotes the same, but states that the sequence of actions is performed at least once in comparison to the first one where it is also allowed to perform it zero times. E.g. the formula $(a \cdot b + c \cdot d)^+$ states that either the sequence a followed by b is performed or the sequence c followed by d is performed. These two sequences may occur multiple times in any order but one sequence must be performed at least once.

Regular formulas are very expressive though but they do not serve every purpose. We introduce the minimal and maximal fixed point operators to the Hennessy-Milner logic to allow an even more expressive language. The modal μ-calculus is given by the following syntax:

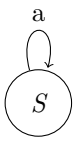
$\phi ::= \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \langle a\rangle\phi \mid [a]\phi \mid \mu X.\phi \mid \nu X.\phi \mid X$

The formula $\mu X.\phi$ denotes the minimal fixpoint and $\nu X.\phi$ denotes the maximal fixed point and X is a variable. Note that X is commonly used as a variable name but any other would do, like Y or Z. Now consider the variable X as a set of states, then the formula $\mu X.\phi$ is valid for all those states in the smallest set X that satisfies the equation $X = \phi$ where X generally occurs in ϕ. The equation should be read as X being the set of states in which ϕ is valid. The same holds for $\nu X.\phi$ except that it is valid for all those states in the largest set X that satisfies the equation $X = \phi$.

We will illustrate this by looking at two examples, namely $\mu X.X$ and $\nu X.X$. So respectively we are looking for the smallest and largest set of states for which $X = X$ holds. As both hands of the equation are the same we know that any set of states is valid. Now the smallest set out of the superset of all possible sets of states is of course the empty set whereas the largest set is the set with all states. This is respectively equivalent to stating that $\mu X.X = \mathit{false}$ and $\nu X.X = \mathit{true}$.

Let's have a look at a second example where we consider the formulas $\mu X.\langle a\rangle X$ and $\nu X.\langle a\rangle X$. Assume that we have a transition system with one state S and one transition a.

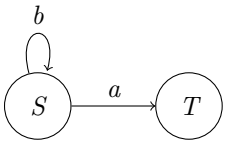
In total we have to consider two sets of states, namely the empty set $X = \emptyset$ and the set of all states $X = S$. These are also respectively the smallest and largest set of states for this transition



system. The minimal fixpoint formula can be solved by substituting the variable X with the smallest set of states, thus the formula becomes $false = \langle a \rangle false$. This states that only in the set of empty states it is impossible to do an a action. For the maximum fixpoint formula we substitute X with the largest set which is S in this case, so the formula becomes $\{S\} = \langle a \rangle \{S\}$. This states that from every possible state we can always do an a action. So the minimal fixpoint does not holds in state S whereas the maximal fixpoint does hold.

An intuitive method to understand if a fixpoint formula is valid is by thinking of it as a states-pace graph, where the fixpoint variables can be considered as states and the box and diamond modalities can be seen as transitions between these states. A minimal fixpoint formula is true if it can be made true by passing a finite amount of times through the fixpoint variables. For a maximal fixpoint formula it must be possible to pass an infinite number of times through the fixpoint variables for it to be true.

We look at another example to get a better feeling for these formulas where we consider the following transition system:



The formula $\mu X.[\overline{a}]X$ is not valid for the above transition system. It states that is possible to perform an finite number of b actions (actually all actions except an a action) which is clearly not the case. The formula $\mu X.([\overline{a}]X \vee \langle a \rangle true)$ does hold as it possible to eventually perform an a action. We end this example by taking a look at the formula $\nu X.[\overline{a}]X$ which does hold for this transition system, because an infinite number of b actions can be performed.

We conclude this section be mentioning that safety properties usually are expressed with a maximum fixpoint formula, because we generally want such properties to always hold for a system. Liveliness properties are generally expressed with minimum fixpoint formulas, as these state that something good will eventually happen and is not postponed indefinitely.

6.2 HDS requirements

In this section we translate the functional requirements for the HDS system to modal formulas, as described in chapter 3. We start with a generic, generally wanted property for a system that is not included in the functional requirements: the system is deadlock free.

Deadlock free:
$[true^*]\langle true \rangle true$

This property states that after any number of actions is performed, it remains possible to do another action. In other words the system never ends up in a situation where the system is unable to perform anything anymore.

FR.1:
$\forall_{n,m:location}[true^* \cdot step(P_5, 1, m) \cdot \forall_{l:location, id:lift\_id}(\overline{step(P_5, id, l)})^* \cdot step(P_5, 2, n)](n > m)$
$\forall_{n,m:location}[true^* \cdot step(P_5, 2, m) \cdot \forall_{l:location, id:lift\_id}(\overline{step(P_5, id, l)})^* \cdot step(P_5, 1, n)](n < m)$

These properties state that two consecutive steps, of the two motors that drive the lift platforms, can never end up that same location. If they would then this implies that the lift platforms have

collided, so we verify that the location of one platform is below (or above) the other one. This property is inspected in two parts: one formula checks a step from motor 1 followed by a step by motor 2 and the second formula does the exact opposite. The formula had to be split in two parts to keep memory requirements small enough to calculate the results. This is also the case for functional requirement FR.6.

FR.2:

$\forall_{id:lift\_id}[true^* \cdot liftLoadInput(P_4, id) \cdot (\overline{liftUnload(P_4, id)})^* \cdot liftLoadInput(P_4, id)]false$
$\forall_{id:lift\_id}[true^* \cdot liftLoad(P_4, id) \cdot (\overline{liftUnloadOutput(P_4, id)})^* \cdot liftLoad(P_4, id)]false$

The first formula states that a lift platform does not transfer a tote from the input conveyor to the lift platform if a previous transferred tote is not yet delivered to a transfer-in buffer location. In other words this property states that it is impossible to place two totes on the temporary input location of the lift platform at the same time. The second formula states that a lift platform does not transfer a tote from a transfer-out buffer to the lift platform if a previously retrieved tote is not yet delivered to the output conveyor. In other words this property states that it is impossible to place two totes on the lift platform temporary output location at the same time. The formulas express this by stating the loading actions can not occur twice without an unload action in between.

FR.3:

$\forall_{l:location}[true^* \cdot tsinput(P_3, l) \cdot (\overline{shuttleLoad(P_5, l)})^* \cdot tsinput(P_3, l)]false$
$\forall_{l:location}[true^* \cdot shuttleUnload(P_5, l) \cdot (\overline{tsoutput(P_3, l)})^* \cdot shuttleUnload(P_5, l)]false$

The first formula states that once a tote is placed on a transfer-in buffer (*tsinput*), then a shuttle must first retrieve this tote from the buffer before another tote may be placed on the same location. The second formula states that once a tote is placed on a transfer-out buffer by a shuttle, then a lift platform must first retrieve this tote from the buffer (*tsoutput*) before another tote may be placed on the same location.

FR.4:

$[true^* \cdot hasInput]\mu X.[\overline{loadedInput(P_3)}]X$

This property states that totes that arrive at the input conveyor are eventually retrieved by a lift platform. This is expressed by saying that after a *hasInput* action, eventually a *loadedInput* action is performed. Note that we assume the system to be deadlock free, as otherwise the minimal fixpoint (μ) also becomes valid if the system deadlocks since then a shortest path also exists. However we are only interested in a shortest path in which an *loadedInput* action is performed.

FR.5:

$\forall_{id:lift\_id}[true^* \cdot (\forall_{l:location} startmotor(P_3, id, l))]\mu X.[\overline{stopmotor(P_3, id)}]X$

This property states that after a motor receives a command to go to a new location, that it will eventually signal the completion of that command. In other words this property states that a command from the lift platform layer is eventually executed. Note that deadlock freedom is required here as well. The next three requirements, FR.6, FR.7 and FR.8, assume deadlock freedom as well.

FR.6:

$\forall_{id:lift\_id}[true^* \cdot (\forall_{l:location} doublecycle(P_3, id, l))]\mu X.[\overline{liftReady(P_3, id)}]X$
$\forall_{id:lift\_id}[true^* \cdot (\forall_{l,l':location} seqcycle(P_3, id, l, l'))]\mu X.[\overline{liftReady(P_3, id)}]X$
$\forall_{id:lift\_id}[true^* \cdot (\forall_{l:location} incycle(P_3, id, l))]\mu X.[\overline{liftReady(P_3, id)}]X$
$\forall_{id:lift\_id}[true^* \cdot (\forall_{l:location} outcycle(P_3, id, l))]\mu X.[\overline{liftReady(P_3, id)}]X$

This property is distributed over four formulas. Each strategy (cycle), as discussed in Section 2.2, is eventually followed by the *liftReady* action that indicates that the chosen strategy has been performed. In other words this property states that the lift agent layer's commands are eventually executed.

FR.7:
$\forall_{id:lift\_id}[true^* \cdot (\forall_{cmd:lift\_cmd} sendCmd(P_3, id, cmd))]\mu X.[\overline{cmdDone(P_3, id)}]X$

This property states that commands from the mast layer are eventually executed.

FR.8:
$\forall_{id:lift\_id}[true^* \cdot (\forall_{cmd:lift\_cmd} sendLift(P_3, id, cmd))]\mu X.[\overline{liftArrived(P_3, id)}]X$

The final property states that commands from the motion manager layer are eventually executed.

The model of the HDS system can be verified with these formulas but this does not give us the correct results. These formulas check all possible transition paths through the system but also the ones that should not be checked as other action have precedence over others. So these formulas are insufficient for the mCRL2 model the HDS system as they do not respect the priorities that are bound to the actions. In the following two section we will see how this issue is resolved.

6.3 Priorities in modal formulas

In Section 4.3 we mentioned that we use the 'prioritized walk' option for the simulation of the model. The modal formulas that we use to check properties of the mCRL2 modal of the HDS system are used to generate a PBES with the `lps2pbes` tool. The PBES is then verified by the `pbes2bool` tool that determines if the modal formula is valid for the model. However the `lps2pbes` tool does not have a 'prioritized walk' option so we need to find a way to incorporate the priority property within the modal formula. In this section we will show how to translate the 'prioritized walk' option from the `lps2lts` tool into a modal formula. A good way to understand this translation is by looking at an example specification which behaves differently under the 'prioritized walk' option than without it. Assume the following specification P_{spec}:

$act\ a : \mathbb{N}; b;$

$proc\ P(predicate : \mathbb{B}) =$
$\quad predicate \rightarrow a(1) \cdot P(false)$
$\quad + \quad a(2) \cdot P(true)$
$\quad + \quad b \cdot P(predicate);$

$init\ P(false);$

This specification contains two actions: an a action with a priority parameter and a b action without a priority parameter. From now on we use the notation a_n to denote the action $a(n)$ (where $n : \mathbb{N}$). Keep in mind that a b action can be always be performed and that a_n has a higher priority than a_m, for all values that satisfy $n < m$.

Assume that we want to verify the following property P_{req}: it is not possible to perform two consecutive a_2 actions without an a_1 action in between. This property holds in the case that priorities are respected, but does not hold when priorities are ignored because a_2 can always be performed. In other words the selection of an a_1 action is not enforced. The following formula expresses this property; P_{req}: $[true \cdot a_2 \cdot \overline{a_1}^* \cdot a_2]false$, which is invalid in specification P_{spec}. Note that P_{req} does hold if we modify specification P_{spec} to only allow a_2 under the condition $\neg predicate$. However the specification should also hold without this modification if the priorities are respected so this formula is not strong enough.

Let's first look at the formula $[true^*]\phi$, and see how we can incorporate the priority property for a sequence of actions. First we must state that this will hold for the entire state space. For this we use a maximal fixpoint as this states that the formula must hold for the largest possible set of states. First we must take caution that is always possible to do a b action regardless of the other actions. In combination with the maximal fixpoint this translates to the modal formula $\nu X.([b]X)$. Note that if no outgoing transitions exists by doing a b action then $[b]$ also results

in true. Now we would actually like something such as an if-else-statement but this is not supported by the μ-calculus language. Instead we will follow the following pattern: we state that we can always do an action of the lowest priority and then validate the rest of the state space after this transition. However if it was possible to do an action with a higher priority then we could actually forget about the lower priority action and follow one of the higher priority actions (though we don't know which one at this point). This leads us to the following modal formula:

$\nu X.([b]X \wedge ([a_2]X \vee \langle a_1 \rangle true) \wedge \phi)$

Solving this structure is handled well by the `pbes2bool` tool which does not evaluate the complete recursive path of the a_2 actions. Instead it also evaluates the diamond box modality which means that if a higher priority action exists, then the diamond modality becomes *true*. Because the box and diamond modality formula parts are disjunctive, this allows the tool to immediately conclude that the formula part $[a_2]X \vee \langle a_1 \rangle true$ is valid.

We generalize the above formula and give a definition for priority based formulas. Let Φ_n denote the box modality of the formula and Ψ_m denote the diamond modality part of the formula, where n and m denote the priority of the actions in the modalities. Furthermore let ζ be the disjunction of all non-prioritized actions, then the definition of a prioritized formula is as follows:

Definition 6.1 *(prioritized modal formulas)*
$\nu X.([\zeta]X \bigwedge_{n,m:\mathbb{N},n<m} (\Phi_n \vee \Psi_m))$

Property P_{req} also starts with a sequence of actions followed by an a_2 action. An conjunction is added to the formula, according to the priority definition: either an a_2 action is performed after which the rest of the formula must hold (θ), or an higher priority action (a_1) is possible.

$\nu X.([b]X \wedge ([a_2]X \vee \langle a_1 \rangle true) \wedge [a_1]X \wedge ([a_2]\theta \vee \langle a_1 \rangle true))$

Here θ denotes the remainder of P_{req} (denoted in teletext below) which we still need to translate:
$[true \cdot a_2 \cdot \overline{\mathtt{a_1}}^* \cdot \mathtt{a_2}]\mathtt{false}$

Thus a sequence of actions excluding an a_1 action, after which an a_2 action is not allowed. The formula θ is according to the definition 6.1 and is as follows:

$\theta = \nu Y.([b]Y \wedge ([a_2]Y \vee \langle a_1 \rangle true) \wedge ([a_2]false \vee \langle a_1 \rangle true))$

Note that we do not inspect the a_1 transitions ($[a_1]X$) in θ. However we do need to check if an a_1 action was possible that acts as a higher priority action, eliminating the need to verify the a_2 transitions paths from the current state as these actions do not need to be inspected because of their lower priority.

6.4 HDS prioritized requirements

In this section we adapt the formulas from Section 6.2, to formulas that respect the priority aspect. As these formulas tend to become large, we add them as appendix B. However we first look at the deadlock free property and show how we can reduce the formula which is used for all the formulas. The deadlock free property is expressed as: $[true^*]\langle true \rangle true$

In our system we have a total of four priorities denoted as $P_{2...5}$ and a larger set of actions than in our example from the previous example. By applying definition 6.1 from the previous section we get the following formula:

$$
\begin{array}{l}
\nu X \cdot ([hasInput]X \wedge \\
\quad (\forall_{id:lift\_id,l:location}[step(P_5,id,l) \vee shuttleLoad(P_5,l) \vee shuttleUnload(P_5,l)]X \vee \\
\qquad \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\qquad \langle transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id) \vee startmotor(P_3,id,l) \vee \\
\qquad stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee loadedInput(P_3) \vee liftReady(P_3,id) \vee \\
\qquad tsoutput(P_3,l) \vee tsinput(P_3,l) \vee incycle(P_3,id,l) \vee outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee \\
\qquad doublecycle(P_3,id,l) \vee sendCmd(P_3,id,cmd) \vee sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id) \vee \\
\qquad liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id) \rangle true) \\
\quad \wedge \\
\quad (\forall_{id:lift\_id}[liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id)]X \vee \\
\qquad \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\qquad \langle transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id) \vee startmotor(P_3,id,l) \vee \\
\qquad stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee loadedInput(P_3) \vee liftReady(P_3,id) \vee \\
\qquad tsoutput(P_3,l) \vee tsinput(P_3,l) \vee incycle(P_3,id,l) \vee outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee \\
\qquad doublecycle(P_3,id,l) \vee sendCmd(P_3,id,cmd) \vee sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id) \rangle true) \\
\quad \wedge \\
\quad (\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\qquad [startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\qquad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee incycle(P_3,id,l) \vee \\
\qquad outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee sendCmd(P_3,id,cmd) \vee \\
\qquad sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id)]X \vee \\
\qquad \exists_{id:lift\_id,l:location,l':location} \langle transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id) \rangle true) \\
\quad \wedge \\
\quad \forall_{id;lift\_id,l:location,l':location}[transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id)]X \\
) \\
\wedge \langle true \rangle true
\end{array}
$$

This can be shortened by rewriting the formula, denoted according to the syntax of definition 6.1, which reduces the required amount of memory that is needed to validate the formula:

$$
\begin{array}{l}
\quad (\Phi_5 \vee \Psi_4 \vee \Psi_3 \vee \Psi_2) \wedge (\Phi_4 \vee \Psi_3 \vee \Psi_2) \wedge (\Phi_3 \vee \Psi_2) \wedge \Phi_2 \\
\equiv \\
\quad (((\Phi_5 \vee \Psi_4 \vee \Psi_3) \wedge (\Phi_4 \vee \Psi_3) \wedge \Phi_3) \vee \Psi_2) \wedge \Phi_2 \\
\equiv \\
\quad (((((\Phi_5 \vee \Psi_4) \wedge \Phi_4) \vee \Psi_3) \wedge \Phi_3) \vee \Psi_2) \wedge \Phi_2
\end{array}
$$

The complete formula is then as follows:

$$
\begin{array}{l}
\nu X \cdot ([hasInput]X \wedge \\
(((((\\
\forall_{id:lift\_id,l:location}[step(P_5,id,l) \vee shuttleLoad(P_5,l) \vee shuttleUnload(P_5,l)]X \vee \\
\exists_{id:lift\_id} \langle liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id) \rangle true) \wedge \\
\quad \exists_{id:lift\_id}[liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id)]X) \vee \\
\qquad \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\qquad \langle startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\qquad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee \\
\qquad incycle(P_3,id,l) \vee outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee \\
\qquad sendCmd(P_3,id,cmd) \vee sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id) \rangle true) \wedge \\
\qquad \forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\qquad [startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\qquad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee incycle(P_3,id,l) \vee \\
\qquad outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee sendCmd(P_3,id,cmd) \vee \\
\qquad sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id)]X) \vee \\
\qquad\quad \exists_{id:lift\_id,l:location,l':location} \langle transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id) \rangle true \\
) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id)]X \\
) \\
\wedge \langle true \rangle true
\end{array}
$$

The modal formulas of the functional requirements FR.1 until FR.8 are included as Appendix B,

which have all been verified with the `pbes2bool` tool of the mCRL2 toolset. So all the functional requirements of Appendix B, which are the prioritized translated formulas of Section 6.2 are valid, for the prioritized HDS model.

Chapter 7

Conclusion

Throughout this thesis, we have described a specification for the HDS system according to the vision of Vanderlande Industries. We presented two models for this system: one model is used to determine the performance of the system whereas the other model is used to validate its behavioral properties and safety requirements.

We took great care to design a modeling technique that allows us to run simulations of the model for the HDS system. For this we introduced and designed two lift systems, that act as a *M/D/1* and *M/D/2* server system. First we designed a small and simple *M/D/1* lift system to determine the viability of the measuring technique. This technique is verified by comparing the simulation results to analytical approximations. When we wanted to run the simulation with a large number of totes we found that the mCRL2 toolset had limitations on the length of a list, which we used to store the Poisson distributed arrival times of incoming totes. We solved this issue by splitting the list in multiple parts and concatenate them on initialization. We found that the simulation of the *M/D/1* system matched the analytical approximation very well and continued with the design of the *M/D/2* lift system.

An *M/D/2* system has been designed in such a way that it can be used as the core part of the model of the HDS system. In this model we have introduced a monitor process that allows multiple processes of an mCRL2 model to globally keep track of time. This monitor allows us to synchronize all processes of the model in which timed actions, actions that occur at the same time in the real system, are also executed in such a manner in the model. We then applied the 'prioritized walk' option of `lps2lts` to perform a simulation run of the model. We compared the analytical approximation to the simulation results of our *M/D/2* system and found the results matched with high accuracy as shown in Chapter 4.

The timing construction of the *M/D/2* model has been incorporated into the model of the HDS system and simulations have been performed on it. We have determined that the performance is better than in the *M/D/2* case if we mimic it by only allowing totes to be stored in the system. We have also performed simulations in which the system both stores and retrieves totes from the system as it is specified to do. Because we have opted that the arrival of retrieval totes can occur at any time we found that the results all end up around the maximum utilization of the system. However we believe that the simulation results of the HDS system are not yet correct. The model still needs to be calibrated someday, such that more accurate results can be determined from the simulations.

The simulated version of the model of the HDS system does not allow for all possible situations to be verified. We have translated this model to a timeless variant in which priorities determine the order in which actions are executed instead of the time attributes. We validated all of the wanted properties of the system on this model, which was only possible if we kept the distance between two transfer point places to a minimum with a total of three transfer points: the minimum number required for the HDS system to function properly. Allowing more space between the transfer points or allowing more transfer points was found to be unfeasible in terms of computer memory requirements. We have verified all possibilities of the system instead of just a single simulation instance, as we were able to allow new totes to arrive at any given time instead of some

sequence of absolute times. This holds for both storage as well as retrieval totes in the prioritized model, thus we inspect all possible combinations that the HDS system might encounter.

The modal formulas that were used to determine the properties have been adapted such that they respect the priorities that are bound to the actions. The expressions for such formulas tend to become very large. However we found that the `pbes2bool` program handled these kind of structures efficiently, thus limiting the disadvantage of these formulas to their expression size only.

In conclusion we can say that we have determined that it is possible to perform simulations of the HDS system with the mCRL2 toolset of which we also provided verification on a timeless model variant, which relates close to the timed simulation variant.

Bibliography

[1] Michael Alexander and William Gardner. *Process Algebra for Parallel and Distributed Processing*. Chapman & Hall/CRC, 2008. The mCRL2 homepage: http://www.mcrl2.org.

[2] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. <http://drops.dagstuhl.de/opus/volltexte/2007/862> [date of citation: 2007-01-01].

[3] J. F. Groote and M. A. Reniers. *Designing and understanding the behaviour of systems: Lecture notes for RADV 2IW25 (2007/2008)*. Department of Computer Science, Eindhoven University of Technology, Eindhoven, 2007. (unpublished).

[4] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, London, UK, 1980. Springer-Verlag.

[5] R. P. W. Springer. *Design and verificiation of the SmartPixel II Protocol*. Department of Computer Science, Eindhoven University of Technology, Eindhoven, 2008.

[6] H.C. Tijms. *Stochastic modelling and analysis : a computational approach*. Chichester: Wiley, 1990.

Appendix A

mCRL2 models

A.1 Single-platform elevator system

```
act
  s_input, r_input, input : Nat;
  s_terminate, r_terminate, terminate;
  result: Nat#Nat#Nat#Nat;

map C: Nat;
eqn C = 500; % denotes 5,00 seconds

proc delivery(t:List(Nat), at:Nat) =
  (t!=[]) -> s_input(at+t.0).delivery(tail(t),at+t.0)
  +
  (t==[]) -> s_terminate.delivery(t,at)
;

proc lift(lat: Nat, ct: Nat, nrt: Nat, idle:Nat) = % last action time = lat
  sum t:Nat.r_input(t).
    lift(max(t,lat+C)+C, ct+Int2Nat(max(t,lat+C)+C-t), nrt+1,
         idle+if(t>lat+C, Int2Nat(t-(lat+C)),0) )
  +
  r_terminate.result(ct,lat,nrt,idle).delta
;

init(
  allow({input,terminate,result},
    comm({s_input|r_input->input,
          s_terminate|r_terminate->terminate},
         lift(0,0,0,0)||
  delivery(f0,0)
  )))
;
map f0: List(Nat);
eqn f0 = [<exponential distributed numbers>];
```

A.2 Parallel-platform elevator system

```
sort
    lift_id = Pos;
    location = Pos;
    time = Nat;
    priority = Nat;
    direction = struct up?is_up |
                       down?is_down |
                       neutral?is_neutral;
    lift_action = struct loadtransfer?is_loadtransfer |
                         unloadtransfer?is_unloadtransfer |
                         loadunloadtransfer?is_loadunloadtransfer |
                         loadinput?is_loadinput |
                         unloadoutput?is_unloadoutput |
                         move?is_move;
    lift_cmd = struct pair(dest:location, action:lift_action);
    step_action = struct idle?is_idle |
                         stepping?is_stepping |
                         arrived?is_arrived;

map LOCS: location;
```

```
eqn LOCS = 60;

map    setDir: location#location#direction -> direction;
var    curloc,dest: location;
curdir: direction;
eqn    setDir(curloc,dest,curdir) = if(curloc==dest,neutral,
           if(dest>curloc,up,down));

map HUGE: time;
eqn HUGE = 99999999999999999999;

act
  s_startmotor,r_startmotor,startmotor: time#lift_id#location;
  s_stopmotor,r_stopmotor,stopmotor: time#lift_id;
  s_step,r_step,step: time#lift_id#location;

  s_sendLift,r_sendLift,sendLift: time#lift_id#lift_cmd;
  s_liftArrived,r_liftArrived,liftArrived: time#lift_id;

  s_setlat,r_setlat,setlat: time;
  s_getlat,r_getlat,getlat: time;

  r_loadedInput,s_loadedInput,loadedInput: time;
  s_liftReady,r_liftReady,liftReady: time#lift_id;
  s_inReady,r_inReady,inReady: time;

  r_result,s_result,result: time#Nat#time#time; %cm,cnt.idle1,idle2
  print_lat: time;
  r_tick,s_tick,tick: time#time;

proc

delivery(at: List(Nat), na:Nat) =
(at!=[]) -> (
  sum t_in:Nat. (t_in>=na+at.0) ->
    r_getlat(t_in).
    s_inReady(na+at.0).
    sum t_load:time.r_loadedInput(t_load).
    delivery(tail(at),na+at.0)
  +
  s_tick(HUGE,na+at.0).delivery(at,na)
)
+
(at==[]) ->
  sum ct,cnt,i1,i2:time.r_result(ct,cnt,i1,i2).delta
;

control(l1:List(lift_cmd),l2:List(lift_cmd),b1:Bool,b2:Bool,
    s1:time,s2:time,ct:time,cnt:Nat,
    e1:time,e2:time,i1:time,i2:time) =
(l1==[]) -> %assign to lift1 if both idle
  sum t:time.r_inReady(t).
  control([pair(10,loadinput),
         pair(LOCS,unloadtransfer),
         pair(10,move)],l2,
         b1,b2,t,s2,ct,cnt,e1,e2,i1,i2)
+
(l1!=[]&&l2==[]) -> % assign to lift2 if other busy
  sum t:Nat.r_inReady(t).
  control(l1,[pair(10,loadinput),
           pair(LOCS,unloadtransfer),
           pair(10,move)],
         b1,b2,s1,t,ct,cnt,e1,e2,i1,i2)
+
(l1==[]&&l2==[]) ->
  s_result(ct,cnt,i1,i2).
  sum lat:time.r_getlat(lat).print_lat(lat).
  control(l1,l2,b1,b2,s1,s2,ct,cnt,e1,e2,i1,i2)
+
(l1!=[]&&!b1) ->
  sum t:time.r_getlat(t).
  s_sendLift(t,1,head(l1)).
  control(l1,l2,true,b2,s1,s2,ct,cnt,
         e1,e2,
         if(is_loadinput(action(head(l1))),i1+Int2Nat(t-e1),i1),i2)
+
(l2!=[]&&!b2) ->
  sum t:time.r_getlat(t).
  s_sendLift(t,2,head(l2)).
  control(l1,l2,b1,true,s1,s2,ct,cnt,
         e1,e2,
         i1,if(is_loadinput(action(head(l2))),i2+Int2Nat(t-e2),i2))
+
sum id:lift_id.
  sum t:Nat.r_liftArrived(t,id).
  sum tr:time.r_liftReady(tr,id). (
    (id==1) ->
```

```
      control(tail(l1),l2,false,b2,
        s1,s2,
        ct+if(is_unloadtransfer(action(head(l1))),Int2Nat(t-s1),0),
        cnt+if(is_unloadtransfer(action(head(l1))),1,0),
        if(is_move(action(head(l1))),t,e1),e2,i1,i2)
    +
    (id==2) ->
      control(l1,tail(l2),b1,false,
        s1,s2,
        ct+if(is_unloadtransfer(action(head(l2))),Int2Nat(t-s2),0),
        if(is_unloadtransfer(action(head(l2))),cnt+1,cnt),
        e1,if(is_move(action(head(l2))),t,e2),i1,i2)
  )
;

liftplatform(id: lift_id, c:location, d:location, busy:Bool, a:lift_action) =
!busy ->
sum cmd:lift_cmd,t:time.
r_sendLift(t,id,cmd).
s_startmotor(t,id,dest(cmd)).
liftplatform(id,c,dest(cmd),true,action(cmd))
+
(busy && c<d) ->
sum t:time.s_step(t,id,c+1).liftplatform(id,c+1,d,busy,a)
+
(busy && c>d) ->
sum t:time.s_step(t,id,Int2Pos(c-1)).liftplatform(id,Int2Pos(c-1),d,busy,a)
+
(busy && c==d) ->
sum t:Nat. r_stopmotor(t,id).
  (
    is_loadinput(a) ->
      s_loadedInput(t). % report 'tote loaded' to delivery
      s_liftArrived(t,id).
      s_liftReady(t,id). % report ready to control
  liftplatform(id,c,d,false,a)
    +
    (is_unloadtransfer(a)||is_move(a)) ->
      s_liftArrived(t,id).
      s_liftReady(t,id).
  liftplatform(id,c,d,false,a)
  )
;

stepper(c1:location, c2:location, m1:direction, m2:direction,
    lat: Nat, p1:location, p2:location, s1:step_action, s2:step_action) =
(!is_arrived(s1) && !is_arrived(s2)) -> (
(is_neutral(m2) && is_up(m1)) ->
sum d:location.r_step(lat+1,1,d).s_setlat(lat+1).
  stepper(c1+1,c2,m1,m2,lat+1,p1,p2,
        if(c1+1==p1&&is_stepping(s1),arrived,s1),
        if(c2==p2&&is_stepping(s2),arrived,s2))
+
(is_neutral(m2) && is_down(m1)) ->
sum d:location.r_step(lat+1,1,d).s_setlat(lat+1).
  stepper(Int2Pos(c1-1),c2,m1,m2,lat+1,p1,p2,
        if(Int2Pos(c1-1)==p1&&is_stepping(s1),arrived,s1),
        if(c2==p2&&is_stepping(s2),arrived,s2))
+
(is_neutral(m1) && is_up(m2)) ->
sum d:location.r_step(lat+1,2,d).s_setlat(lat+1).
  stepper(c1,c2+1,m1,m2,lat+1,p1,p2,
        if(c1==p1&&is_stepping(s1),arrived,s1),
        if(c2+1==p2&&is_stepping(s2),arrived,s2))
+
(is_neutral(m1) && is_down(m2)) ->
sum d:location.r_step(lat+1,2,d).s_setlat(lat+1).
  stepper(c1,Int2Pos(c2-1),m1,m2,lat+1,p1,p2,
        if(c1==p1&&is_stepping(s1),arrived,s1),
        if(Int2Pos(c2-1)==p2&&is_stepping(s2),arrived,s2))
+
(is_up(m1) && is_up(m2)) ->
sum d:location.r_step(lat+1,2,d).
sum d:location.r_step(lat+1,1,d).
s_setlat(lat+1).
  stepper(c1+1,c2+1,m1,m2,lat+1,p1,p2,
        if(c1+1==p1&&is_stepping(s1),arrived,s1),
        if(c2+1==p2&&is_stepping(s2),arrived,s2))
+
(is_up(m1) && is_down(m2)) ->
sum d:location.r_step(lat+1,1,d).
sum d:location.r_step(lat+1,2,d).
s_setlat(lat+1).
  stepper(c1+1,Int2Pos(c2-1),m1,m2,lat+1,p1,p2,
        if(c1+1==p1&&is_stepping(s1),arrived,s1),
        if(Int2Pos(c2-1)==p2&&is_stepping(s2),arrived,s2))
+
```

```
(is_down(m1) && is_down(m2)) ->
sum d:location.r_step(lat+1,1,d).
sum d:location.r_step(lat+1,2,d).
s_setlat(lat+1).
  stepper(Int2Pos(c1-1),Int2Pos(c2-1),m1,m2,lat+1,p1,p2,
        if(Int2Pos(c1-1)==p1&&is_stepping(s1),arrived,s1),
        if(Int2Pos(c2-1)==p2&&is_stepping(s2),arrived,s2))
+
(is_down(m1) && is_up(m2)) ->
sum d:location.r_step(lat+1,1,d).
sum d:location.r_step(lat+1,2,d).
s_setlat(lat+1).
  stepper(Int2Pos(c1-1),c2+1,m1,m2,lat+1,p1,p2,
        if(Int2Pos(c1-1)==p1&&is_stepping(s1),arrived,s1),
        if(c2+1==p2&&is_stepping(s2),arrived,s2))
)
+
sum dest:location,id:lift_id,t:time.
  r_startmotor(t,id,dest).
stepper(c1,c2,
       if(id==1,setDir(c1,dest,m1),m1),
         if(id==2,setDir(c2,dest,m2),m2),
         t,
         if(id==1,dest,p1),if(id==2,dest,p2),
         if(id==1,stepping,s1),if(id==2,stepping,s2))
+
sum id:lift_id.
s_stopmotor(lat,id).
stepper(c1,c2,
         if(id==1,neutral,m1),
         if(id==2,neutral,m2),
         lat,p1,p2,
         if(id==1,idle,s1),if(id==2,idle,s2))
;

monitor(t: Nat) =
sum l:Nat. (l>=t) -> r_setlat(l).monitor(l)
+
s_getlat(t).monitor(t)
+
sum d:time.r_tick(HUGE,d).monitor(d)
;

init
allow({startmotor,stopmotor,
       liftArrived,
       sendLift,
       step,
       setlat,getlat,
       inReady,loadedInput,liftReady,
       result,tick,print_lat
       },
  comm({s_liftArrived|r_liftArrived->liftArrived,
       s_sendLift|r_sendLift->sendLift,
       s_startmotor|r_startmotor->startmotor,
       s_stopmotor|r_stopmotor->stopmotor,
       s_step|r_step->step,
       s_setlat|r_setlat->setlat,
       s_getlat|r_getlat->getlat,
       s_loadedInput|r_loadedInput->loadedInput,
       s_inReady|r_inReady->inReady,
       s_result|r_result->result,
       s_liftReady|r_liftReady->liftReady,
       s_tick|r_tick->tick
       },
monitor(0) ||
%liftplatform(id,pos,dest,busy,action)
liftplatform(1,10,10,false,move) ||
liftplatform(2,10,10,false,move) ||
%stepper(pos1,pos2,dir1,dir2,lat,p1,p2,s1,s2)
stepper(10,10,neutral,neutral,0,10,10,idle,idle)||
control([],[],false,false,0,0,0,0,0,0,0,0)||
delivery(f0,0)
));

map f0: List(Nat);
eqn f0 = [<exponential distributed list of natural numbers>];
```

A.3 Highly Dynamic Storage system timed version

```
sort
    lift_id = Pos;
    location = Pos;
    time = Nat;
    priority = Nat;
```

```
    direction = struct up?is_up |
                down?is_down |
                neutral?is_neutral;
      lift_action = struct loadtransfer?is_loadtransfer |
                           unloadtransfer?is_unloadtransfer |
                           loadunloadtransfer?is_loadunloadtransfer |
                           loadinput?is_loadinput |
                           unloadoutput?is_unloadoutput |
                           move?is_move;
      lift_cmd = struct pair(dest:location, action:lift_action);
    step_action = struct idle?is_idle |
                    stepping?is_stepping |
                    arrived?is_arrived;
    retrieval_state = struct pair(loc:location, claimed:Bool);
    loctime = struct pair(loc:location, t:time);
    lift_state = struct pair(inMotion:Bool,loc:location);
    lift_status = struct lidle?is_lidle | % no assignments left
                    busy?is_busy | % executing an assignment
                    standby?is_standby; % ready for next assignment


map CONV_POS: location;
eqn CONV_POS = 20;

map LOCS: location;
eqn LOCS = 50;

map BOTTOM: location;
eqn BOTTOM = 10;

map NO_LOC: location;
eqn NO_LOC = LOCS+1;

map LOCMOD: location;
eqn LOCMOD = 10;

map   setDir: location#location -> direction;
var   curloc,dest: location;
eqn   setDir(curloc,dest) = if(curloc==dest,neutral,
          if(dest>curloc,up,down));

map LOWPRIOR: time;
eqn LOWPRIOR = 99999999999999999999;

act
  s_startmotor,r_startmotor,startmotor: time#lift_id#location;
  s_stopmotor,r_stopmotor,stopmotor: time#lift_id;
  s_step,r_step,step: time#lift_id#location;

  s_sendLift,r_sendLift,sendLift: time#lift_id#lift_cmd;
  s_liftArrived,r_liftArrived,liftArrived: time#lift_id;

  s_setlat,r_setlat,setlat: time;
  s_getlat,r_getlat,getlat: time;

  r_loadedInput,s_loadedInput,loadedInput: time;
  s_liftReady,r_liftReady,liftReady: time#lift_id;
  s_inReady,r_inReady,inReady: time#time#lift_id; % current-time#queued-time

  r_result,s_result,result: time#time#Nat; %0,cm,cnt
  print_lat: time;
  r_print_idle,s_print_idle,print_idle: lift_id#time;
  r_tick,s_tick,tick: time#time;

  s_shuttleLoad,r_shuttleLoad,shuttleLoad: location;
  s_shuttleUnload,r_shuttleUnload,shuttleUnload: location;

  r_transFree,s_transFree,transFree: time#location#lift_id#location;
  r_outReady,s_outReady,outReady: time#lift_id#location#time;

  r_tsoutput,s_tsoutput,tsoutput: time#location;
  r_tsinput,s_tsinput,tsinput: time#location;

  r_incycle,s_incycle,incycle: time#time#lift_id#location;
  r_outcycle,s_outcycle,outcycle: time#time#lift_id#location;
  r_seqcycle,s_seqcycle,seqcycle: time#time#time#lift_id#location#location;
  r_doublecycle,s_doublecycle,doublecycle: time#time#time#lift_id#location;

  r_sendCmd,s_sendCmd,sendCmd: time#lift_id#lift_cmd;
  r_sendCmds,s_sendCmds,sendCmds: time#lift_cmd#lift_cmd;
  r_cmdDone,s_cmdDone,cmdDone: time#lift_id;

  r_hasInput,s_hasInput,hasInput: time#time; % current-time#queued-time
  liftLoad, liftUnload, liftLoadInput, liftUnloadOutput: time#lift_id;

proc
```

```
% at: inter-arrival times (exponential distribution)
% na: next arrival time in line
delivery(at: List(Nat), na:Nat) =
(at!=[]) -> (
  sum t_in:Nat. (t_in>=na+at.0) ->
    r_getlat(t_in).
    s_hasInput(t_in,na+at.0).
    sum t:time.r_loadedInput(t).
    delivery(tail(at),na+at.0)
  +
  s_tick(LOWPRIOR,na+at.0).delivery(at,na)
)
+
(at==[]) ->
% 0 to force this before the other actions such that it stops after input queue is emtpy
  sum ct,cnt:time.r_result(0,ct,cnt).delivery(at,na)
;

liftplatform(id: lift_id, c:location, d:location, busy:Bool, a:lift_action, e:time,i:time) =
s_print_idle(id,i).
liftplatform(id,c,d,busy,a,e,i)
+
!busy ->
sum cmd:lift_cmd,t:time.
r_sendLift(t,id,cmd).
s_startmotor(t,id,dest(cmd)).
sum lat:time.r_getlat(lat).
liftplatform(id,c,dest(cmd),true,action(cmd),e,i+Int2Nat(lat-e))
+
(busy && c<d) ->
sum t:time.s_step(t,id,c+1).liftplatform(id,c+1,d,busy,a,e,i)
+
(busy && c>d) ->
sum t:time.s_step(t,id,Int2Pos(c-1)).liftplatform(id,Int2Pos(c-1),d,busy,a,e,i)
+
(busy && c==d) ->
sum t:Nat. r_stopmotor(t,id).
(
  is_loadinput(a) ->
    liftLoadInput(t,id).
    s_setlat(t+1).
    s_setlat(t+2).
    s_setlat(t+3).
    s_setlat(t+4).
    s_setlat(t+5).
    s_loadedInput(t+5).
    s_liftArrived(t+5,id).
    liftplatform(id,c,d,false,a,t+5,i)
  +
  (is_move(a)) ->
    s_liftArrived(t,id).
    liftplatform(id,c,d,false,a,t,i)
  +
  (is_loadtransfer(a)) ->
    liftLoad(t,id).
    s_setlat(t+1).
    s_setlat(t+2).
    s_setlat(t+3).
    s_setlat(t+4).
    s_setlat(t+5).
    s_tsoutput(t+5,d).
    s_liftArrived(t+5,id).
    liftplatform(id,c,d,false,a,t+5,i)
  +
  (is_unloadtransfer(a)) ->
    liftUnload(t,id).
    s_setlat(t+1).
    s_setlat(t+2).
    s_setlat(t+3).
    s_setlat(t+4).
    s_setlat(t+5).
    s_tsinput(t+5,d).
    s_liftArrived(t+5,id).
    liftplatform(id,c,d,false,a,t+5,i)
  +
  (is_unloadoutput(a)) ->
    liftUnloadOutput(t,id).
    s_setlat(t+1).
    s_setlat(t+2).
    s_setlat(t+3).
    s_setlat(t+4).
    s_setlat(t+5).
    s_liftArrived(t+5,id).
    liftplatform(id,c,d,false,a,t+5,i)
  +
  (is_loadunloadtransfer(a)) ->
    liftLoad(t,id).liftUnload(t,id).
```

```
    s_setlat(t+1).
    s_setlat(t+2).
    s_setlat(t+3).
    s_setlat(t+4).
    s_setlat(t+5).
    s_tsinput(t+5,d).
    s_tsoutput(t+5,d).
    s_liftArrived(t+5,id).
    liftplatform(id,c,d,false,a,t+5,i)
)
;

stepper(c1:location, c2:location, m1:direction, m2:direction,
    lat:Nat, p1:location, p2:location, s1:step_action, s2:step_action) =
(!is_arrived(s1) && !is_arrived(s2)) -> (
(is_neutral(m2) && is_up(m1)) ->
sum d:location.r_step(lat+1,1,d).s_setlat(lat+1).
  stepper(c1+1,c2,m1,m2,lat+1,p1,p2,
        if(c1+1==p1&&is_stepping(s1),arrived,s1),
        if(c2==p2&&is_stepping(s2),arrived,s2))
+
(is_neutral(m2) && is_down(m1)) ->
sum d:location.r_step(lat+1,1,d).s_setlat(lat+1).
  stepper(Int2Pos(c1-1),c2,m1,m2,lat+1,p1,p2,
        if(Int2Pos(c1-1)==p1&&is_stepping(s1),arrived,s1),
        if(c2==p2&&is_stepping(s2),arrived,s2))
+
(is_neutral(m1) && is_up(m2)) ->
sum d:location.r_step(lat+1,2,d).s_setlat(lat+1).
  stepper(c1,c2+1,m1,m2,lat+1,p1,p2,
        if(c1==p1&&is_stepping(s1),arrived,s1),
        if(c2+1==p2&&is_stepping(s2),arrived,s2))
+
(is_neutral(m1) && is_down(m2)) ->
sum d:location.r_step(lat+1,2,d).s_setlat(lat+1).
  stepper(c1,Int2Pos(c2-1),m1,m2,lat+1,p1,p2,
        if(c1==p1&&is_stepping(s1),arrived,s1),
        if(Int2Pos(c2-1)==p2&&is_stepping(s2),arrived,s2))
+
(is_up(m1) && is_up(m2)) ->
sum d:location.r_step(lat+1,2,d).
sum d:location.r_step(lat+1,1,d).
s_setlat(lat+1).
  stepper(c1+1,c2+1,m1,m2,lat+1,p1,p2,
        if(c1+1==p1&&is_stepping(s1),arrived,s1),
        if(c2+1==p2&&is_stepping(s2),arrived,s2))
+
(is_up(m1) && is_down(m2)) ->
sum d:location.r_step(lat+1,1,d).
sum d:location.r_step(lat+1,2,d).
s_setlat(lat+1).
  stepper(c1+1,Int2Pos(c2-1),m1,m2,lat+1,p1,p2,
        if(c1+1==p1&&is_stepping(s1),arrived,s1),
        if(Int2Pos(c2-1)==p2&&is_stepping(s2),arrived,s2))
+
(is_down(m1) && is_down(m2)) ->
sum d:location.r_step(lat+1,1,d).
sum d:location.r_step(lat+1,2,d).
s_setlat(lat+1).
  stepper(Int2Pos(c1-1),Int2Pos(c2-1),m1,m2,lat+1,p1,p2,
        if(Int2Pos(c1-1)==p1&&is_stepping(s1),arrived,s1),
        if(Int2Pos(c2-1)==p2&&is_stepping(s2),arrived,s2))
+
(is_down(m1) && is_up(m2)) ->
sum d:location.r_step(lat+1,1,d).
sum d:location.r_step(lat+1,2,d).
s_setlat(lat+1).
  stepper(Int2Pos(c1-1),c2+1,m1,m2,lat+1,p1,p2,
        if(Int2Pos(c1-1)==p1&&is_stepping(s1),arrived,s1),
        if(c2+1==p2&&is_stepping(s2),arrived,s2))

)
+
sum dest:location,id:lift_id,t:time.
  r_startmotor(t,id,dest).
  sum pt:time.r_getlat(pt).
stepper(c1,c2,
      if(id==1,setDir(c1,dest),m1),
        if(id==2,setDir(c2,dest),m2),pt,
        if(id==1,dest,p1),if(id==2,dest,p2),
        if(id==1,stepping,s1),if(id==2,stepping,s2))
+
sum id:lift_id.
s_stopmotor(lat,id).
stepper(c1,c2,
        if(id==1,neutral,m1),
        if(id==2,neutral,m2),
```

```
        lat,p1,p2,
        if(id==1,idle,s1),if(id==2,idle,s2))
;

monitor(t: Nat) =
sum l:Nat. (l>=t) -> r_setlat(l).monitor(l)
+
s_getlat(t).monitor(t)
+
sum d:time.r_tick(LOWPRIOR,d).monitor(d)
;

shuttle_agent =
sum l:location. (inRng(l,LOCS,BOTTOM)) -> (
  s_shuttleUnload(l)
+
  s_shuttleLoad(l)
).shuttle_agent
;

% il: input buffers
% ol: output buffers
% ic: claimed input buffers (not full but reserved)
% rl: retrieval status of output buffers (FIFO order)
transfer_agent(il:Set(location), ol:Set(location),
           ic:Set(location), rl:List(retrieval_state),
           t_in:time, t_na:time, rl_t:List(loctime)) =
% Shuttle delivers tote to a (free) output buffer
sum l:location.
(!(l in ol)&&inRng(l,LOCS,BOTTOM)) ->
  r_shuttleUnload(l).
  sum lat:time.r_getlat(lat).
  transfer_agent(il,ol+{l},ic,rl<|pair(l,false),t_in,t_na,rl_t<|pair(l,lat))
+
% Shuttle retrieves tote from (full) input buffer
sum l:location.
(l in il) ->
  r_shuttleLoad(l).transfer_agent(il-{l},ol,ic-{l},rl,t_in,t_na,rl_t)
+
% Free input buffer request
% case 1: reqloc is free and honoured
% case 2: reqloc is full, an alternative is presented
% case 3: all input (reachable) input buffers are full, NO_LOC is returned
sum reqloc:location,lid:lift_id,pt:time.
sum inloc:location.(inloc==ibCheck(lid,reqloc,il,ic)) ->
r_transFree(pt,reqloc,lid,inloc).
transfer_agent(il,ol,ic+{inloc},rl,t_in,t_na,rl_t)
+
% Request which output buffer to retrieve next
sum out:location.(out==getNextout(1,rl)&&out!=NO_LOC) ->
s_outReady(0,1,out,getOuttime(rl_t,out)).
transfer_agent(il,ol,ic,setClaimed(rl,out),t_in,t_na,rl_t)
+
% Request which output buffer to retrieve next
sum out:location.(out==getNextout(2,rl)&&out!=NO_LOC) ->
s_outReady(0,2,out,getOuttime(rl_t,out)).
transfer_agent(il,ol,ic,setClaimed(rl,out),t_in,t_na,rl_t)
+
% An output buffer is emptied (tote loaded from buffer)
sum loc:location.
sum pt:time.r_tsoutput(pt,loc).
transfer_agent(il,ol-{loc},ic,rmRelease(rl,loc),t_in,t_na,rmOuttime(rl_t,loc))
+
% An input buffer is now full (tote unloaded onto buffer)
sum loc:location.
sum pt:time.r_tsinput(pt,loc).
transfer_agent(il+{loc},ol,ic,rl,t_in,t_na,rl_t)
+
% new input available
sum pt:time,na:time.r_hasInput(pt,na).
transfer_agent(il,ol,ic,rl,pt,na,rl_t)
+
% Check if a new input can be stored by lift 1
((t_in!=LOWPRIOR)&&canStore(1,il,ic,BOTTOM)) ->
s_inReady(t_in,t_na,1).
transfer_agent(il,ol,ic,rl,LOWPRIOR,t_na,rl_t)
+
% Check if a new input can be stored by lift 2
((t_in!=LOWPRIOR)&&canStore(2,il,ic,BOTTOM)) ->
s_inReady(t_in,t_na,2).
transfer_agent(il,ol,ic,rl,LOWPRIOR,t_na,rl_t)
;

lift_agent(state:lift_state, id: lift_id) =
  % alleen runnen als de opdracht verwerkt is dus als !inMotion(state)
  inMotion(state) ->
    sum pt:time.
```

```
    r_liftReady(pt,id).lift_agent(pair(false,loc(state)),id)
  +
  !inMotion(state) ->
  % @conv_pos and ready to do some work
  (!inMotion(state)&&loc(state)==CONV_POS) -> (
    sum nextout:location,tout:time.
    r_outReady(0,id,nextout,tout). (
      sum tin:time,pt:time.r_inReady(pt,tin,id).
      sum nextin:location.s_transFree(pt,NO_LOC,id,nextin).
      (nextin==nextout) ->
        s_doublecycle(pt,tin,tout,id,nextout).lift_agent(pair(true,nextout),id)
      <>
        s_seqcycle(pt,tin,tout,id,nextout,nextin).lift_agent(pair(true,nextin),id)
      +
      sum pt:time.r_getlat(pt).
      s_outcycle(pt,tout,id,nextout).lift_agent(pair(true,nextout),id)
    )
    +
    sum tin:time,pt:time.r_inReady(pt,tin,id).
    sum nextin:location.s_transFree(pt,NO_LOC,id,nextin).
    s_incycle(pt,tin,id,nextin).lift_agent(pair(true,nextin),id)
  )
  +
  % not @conv_pos and ready to do some work
  (!inMotion(state)&&loc(state)!=CONV_POS) -> (
    sum nextout:location,tout:time.
    r_outReady(0,id,nextout,tout).
    sum pt:time.r_getlat(pt).
    s_outcycle(pt,tout,id,nextout).lift_agent(pair(true,nextout),id)
    +
    sum tin:time,pt:time.
    r_inReady(pt,tin,id).
    sum nextin:location.s_transFree(pt,NO_LOC,id,nextin).
    s_incycle(pt,tin,id,nextin).lift_agent(pair(true,nextin),id)
  )
;

mast(l1:lift_status, l2:lift_status, l1_work:List(lift_cmd), l2_work:List(lift_cmd),
     l1_loc: location, l2_loc: location,
     sin1:time,sin2:time,sout1:time,sout2:time,
     ct:time,cnt:Nat,e1:time,e2:time) =
  % delivery reports an empty input, terminate simulation and 'print' result action
  (!isHandlingInput(l1_work)&&!isHandlingInput(l2_work)) ->
    s_result(0,ct,cnt).
    sum i1:time.r_print_idle(1,i1).
    sum i2:time.r_print_idle(2,i2).
    sum lat:time.r_getlat(lat).print_lat(lat).delta
    %terminate with delta for simulation, for a complete
    % statespace generation the following line below is needed instead
    %mast(l1,l2,l1_work,l2_work,l1_loc,l2_loc,sin1,sin2,sout1,sout2,ct,cnt,e1,e2)
  +
  % lift_agent signals a input-cycle
  sum dest:location, id:lift_id,pt:time,tin:time.
    r_incycle(pt,tin,id,dest).
    mast(if(id==1,standby,l1), if(id==2,standby,l2),
           if(id==1, [pair(CONV_POS,loadinput),pair(dest,unloadtransfer)], l1_work),
         if(id==2, [pair(CONV_POS,loadinput),pair(dest,unloadtransfer)], l2_work),
         l1_loc, l2_loc, if(id==1, tin, sin1), if(id==2, tin, sin2),sout1,sout2,
         ct,cnt, if(id==1,pt,e1), if(id==2,pt,e2))
  +
  % lift_agent signals a output-cycle
  sum dest:location, id:lift_id, pt:time,tout:time.
    r_outcycle(pt,tout,id,dest).
    mast(if(id==1,standby,l1), if(id==2,standby,l2),
         if(id==1, [pair(dest,loadtransfer),pair(CONV_POS,unloadoutput)], l1_work),
         if(id==2, [pair(dest,loadtransfer),pair(CONV_POS,unloadoutput)], l2_work),
         l1_loc,l2_loc, sin1,sin2, if(id==1, tout, sout1), if(id==2, tout, sout2),
         ct,cnt, if(id==1,pt,e1),if(id==2,pt,e2))
  +
  % lift_agent signals a sequential-cycle
  sum destin, destout:location, id:lift_id, pt:time,tin:time,tout:time.
    r_seqcycle(pt,tin,tout,id,destout,destin).
    mast(if(id==1,standby,l1), if(id==2,standby,l2),
         if(id==1, [pair(CONV_POS,loadinput),pair(destin,unloadtransfer),
                    pair(destout,loadtransfer),pair(CONV_POS,unloadoutput)],
                    l1_work),
         if(id==2, [pair(CONV_POS,loadinput),pair(destin,unloadtransfer),
                    pair(destout,loadtransfer),pair(CONV_POS,unloadoutput)],
                    l2_work),
         l1_loc,l2_loc,
         if(id==1,tin,sin1), if(id==2,tin,sin2),
         if(id==1,tout,sout1), if(id==2,tout,sout2),
         ct,cnt, if(id==1,pt,e1), if(id==2,pt,e2))
  +
  % lift_agent signals a double-cycle
  sum dest: location, id:lift_id, pt:time,tin:time,tout:time.
    r_doublecycle(pt,tin,tout,id,dest).
```

```
    mast(if(id==1,standby,l1), if(id==2,standby,l2),
         if(id==1, [pair(CONV_POS,loadinput),pair(dest,loadunloadtransfer),
                    pair(CONV_POS,unloadoutput)], l1_work),
         if(id==2, [pair(CONV_POS,loadinput),pair(dest,loadunloadtransfer),
                    pair(CONV_POS,unloadoutput)], l2_work),
         l1_loc, l2_loc,
         if(id==1,tin,sin1), if(id==2,tin,sin2),
         if(id==1,tout,sout1), if(id==2,tout,sout2),
         ct,cnt, if(id==1,pt,e1), if(id==2,pt,e2))
  +
  % request the next order for both l1,l2
  (is_standby(l1)&&l1_work!=[] && is_standby(l2)&&l2_work!=[]) ->
    s_sendCmds(min(e1,e2),head(l1_work),head(l2_work)).
    mast(busy,busy,l1_work,l2_work,l1_loc,l2_loc,
         sin1,sin2,sout1,sout2,ct,cnt,e1,e2)
  +
  % request the next order for l1 only, l2 busy or idle
  (is_standby(l1)&&l1_work!=[] && !is_standby(l2)) ->
    s_sendCmd(e1,1,head(l1_work)).
    mast(busy,l2,l1_work,l2_work,l1_loc,l2_loc,
         sin1,sin2,sout1,sout2,ct,cnt,e1,e2)
  +
  % request the next order for l2 only, l1 busy or idle
  (is_standby(l2)&&l2_work!=[] && !is_standby(l1)) ->
    s_sendCmd(e2,2,head(l2_work)).
    mast(l1,busy,l1_work,l2_work,l1_loc,l2_loc,
         sin1,sin2,sout1,sout2,ct,cnt,e1,e2)
  +
  % got signal that previous cmd is now executed
  sum id:lift_id.
    sum pt:time.r_cmdDone(pt,id). (
      (id==1&&#l1_work==1) -> ( % final cmd is now processed for l1
        % signal lift_agent (1) to determine next output
        s_liftReady(pt,1).
        sum e:lift_action.(e==action(head(l1_work))) ->
        mast(lidle,l2,[],l2_work,dest(head(l1_work)),l2_loc,
             sin1,sin2,sout1,sout2,
             ct+if(is_unloadtransfer(e)||is_loadunloadtransfer(e),Int2Nat(pt-sin1),
               if(is_unloadoutput(e),Int2Nat(pt-sout1),0)),
             cnt+if(is_unloadtransfer(e)||is_loadunloadtransfer(e)||
                  is_unloadoutput(e),1,0),
             pt,e2)
      )
      +
      (id==1&&#l1_work>1) -> ( % more cmd's to process for l1
        sum e:lift_action.(e==action(head(l1_work))) ->
        mast(standby,l2,tail(l1_work),l2_work,dest(head(l1_work)),l2_loc,
             sin1,sin2,sout1,sout2,
             ct+if(is_unloadtransfer(e)||is_loadunloadtransfer(e),Int2Nat(pt-sin1),
               if(is_unloadoutput(e),Int2Nat(pt-sout1),0)),
             cnt+if(is_unloadtransfer(e)||is_loadunloadtransfer(e)||
                  is_unloadoutput(e),1,0),
             pt,e2)
      )
      +
      (id==2&&#l2_work==1) -> ( % final cmd is now processed for l2
        % signal lift_agent (2) to determine next output
        s_liftReady(pt,2).
        sum e:lift_action.(e==action(head(l2_work))) ->
        mast(l1,lidle,l1_work,[],l1_loc,dest(head(l2_work)),
             sin1,sin2,sout1,sout2,
             ct+if(is_unloadtransfer(e)||is_loadunloadtransfer(e),Int2Nat(pt-sin2),
               if(is_unloadoutput(e),Int2Nat(pt-sout2),0)),
             cnt+if(is_unloadtransfer(e)||is_loadunloadtransfer(e)||
                  is_unloadoutput(e),1,0),
             e1,pt)
      )
      +
      (id==2&&#l2_work>1) -> ( % more cmd's to process for l2
        sum e:lift_action.(e==action(head(l2_work))) ->
        mast(l1,standby,l1_work,tail(l2_work),l1_loc,dest(head(l2_work)),
             sin1,sin2,sout1,sout2,
             ct+if(is_unloadtransfer(e)||is_loadunloadtransfer(e),Int2Nat(pt-sin2),
               if(is_unloadoutput(e),Int2Nat(pt-sout2),0)),
             cnt+if(is_unloadtransfer(e)||is_loadunloadtransfer(e)||
                  is_unloadoutput(e),1,0),
             e1,pt)
      )
    )
;

% accepts one order at a time from each lift,
% and makes sure that no collissions come into existence
% it will then forward the command to the hardware (liftplatform/stepper)
% also waiting for the arrive (and load handling) of a lift
% s_cmdDone(id) : signal lift_agent that the current command has been executed
% r_liftArrived(id) : receive signal that hardware has executed the command entirely
```

```
motion_mng(l1_cmd: lift_cmd , l2_cmd:lift_cmd , l1:lift_status ,
        l2:lift_status , priority:lift_id , detour:lift_cmd , dt:Nat,
        s1:time , s2:time) =
  % both lifts report a new order (from mast)
  (is_lidle(l1)&&is_lidle(l2)) ->
    sum cmd1, cmd2: lift_cmd , pt:time.r_sendCmds(pt,cmd1,cmd2).
    motion_mng(cmd1,cmd2,standby,standby,priority,detour,dt,pt,pt)
  +
  % lift (id) reports a new order (from mast)
  sum cmd: lift_cmd , id:lift_id .
    sum pt:time.r_sendCmd(pt,id,cmd). (
      (id==1 && is_busy(l1) && is_move(action(l1_cmd))) ->
        motion_mng(l1_cmd,l2_cmd,l1,l2,priority,cmd,1,pt,s2)
      +
      (id==1 && is_lidle(l1)) ->
        motion_mng(cmd,l2_cmd,standby,l2,priority,detour,dt,pt,s2)
      +
      (id==2 && is_busy(l2) && is_move(action(l2_cmd))) ->
        motion_mng(l1_cmd,l2_cmd,l1,l2,priority,cmd,2,s1,pt)
      +
      (id==2 && is_lidle(l2)) ->
        motion_mng(l1_cmd,cmd,l1,standby,priority,detour,dt,s1,pt)
    )
  +
  % lift 1 reports arriving at the scene
  sum pt:time.r_liftArrived(pt,1).  (
    is_move(action(l1_cmd)) -> (
      motion_mng(if(dt==1,detour,l1_cmd),l2_cmd,
               if(dt==1,standby,lidle),l2,
               priority,detour,0,s1,s2)
    )
    +
    !is_move(action(l1_cmd)) -> (
      s_cmdDone(pt,1).
      motion_mng(l1_cmd,l2_cmd,lidle,l2,priority,detour,dt,s1,s2)
    )
  )
  +
  % lift 2 reports arriving at the scene
  sum pt:time.r_liftArrived(pt,2).  (
    is_move(action(l2_cmd)) -> (
      motion_mng(l1_cmd,if(dt==2,detour,l2_cmd),
               l1,if(dt==2,standby,lidle),
               priority,detour,0,s1,s2)
    )
    +
    !is_move(action(l2_cmd)) -> (
      s_cmdDone(pt,2).
      motion_mng(l1_cmd,l2_cmd,l1,lidle,priority,detour,dt,s1,s2)
    )
  )
  % Send commands away, keeping priority in mind
  % This process makes sure that cmd's do not
  % result in physical deadlocks where both
  % lifts want to cross each other
  +
  (is_standby(l1) && is_standby(l2) ) ->  (
    (dest(l1_cmd) < dest(l2_cmd)) ->
      s_sendLift(s2,2,l2_cmd).s_sendLift(s1,1,l1_cmd).
      motion_mng(l1_cmd,l2_cmd,busy,busy,priority,detour,dt,s1,s2)
    +
    (dest(l1_cmd) >= dest(l2_cmd)) ->
      s_sendLift(if(priority==1,s2,s1),if(priority==1,2,1),
               pair(if(priority==1,dest(l1_cmd)+1,Int2Pos(dest(l2_cmd)-1)),move)).
      s_sendLift(if(priority==1,s1,s2),if(priority==1,1,2),
               if(priority==1,l1_cmd,l2_cmd)).
      motion_mng(if(priority==1,l1_cmd,pair(Int2Pos(dest(l2_cmd)-1),move)),
               if(priority==2,l2_cmd,pair(dest(l1_cmd)+1,move)),
               busy,busy,if(priority==1,2,1),
               if(priority==1,l2_cmd,l1_cmd),
               if(priority==1,2,1),s1,s2)
  )
  +
  (is_standby(l1) && is_lidle(l2)) -> (
    (dest(l1_cmd) < dest(l2_cmd)) ->
      s_sendLift(s1,1,l1_cmd).
      motion_mng(l1_cmd,l2_cmd,busy,l2,priority,detour,dt,s1,s2)
    +
    (dest(l1_cmd) >= dest(l2_cmd)) ->
      s_sendLift(s2,2,pair(dest(l1_cmd)+1,move)).s_sendLift(s1,1,l1_cmd).
      motion_mng(l1_cmd,pair(dest(l1_cmd)+1,move),busy,busy,2,detour,dt,s1,s2)
  )
  +
  (is_standby(l1) && is_busy(l2) && dest(l1_cmd) < dest(l2_cmd)) -> (
      s_sendLift(s1,1,l1_cmd).
      motion_mng(l1_cmd,l2_cmd,busy,l2,priority,detour,dt,s1,s2)
  )
```

```
  +
  (is_standby(l2) && is_lidle(l1)) -> (
    (dest(l2_cmd) > dest(l1_cmd)) ->
      s_sendLift(s2,2,l2_cmd).
      motion_mng(l1_cmd,l2_cmd,l1,busy,priority,detour,dt,s1,s2)
    +
    (dest(l2_cmd) <= dest(l1_cmd)) ->
      s_sendLift(s1,1,pair(Int2Pos(dest(l2_cmd)-1),move)).
      s_sendLift(s2,2,l2_cmd).
      motion_mng(pair(Int2Pos(dest(l2_cmd)-1),move),l2_cmd,
                 busy,busy,1,detour,dt,s1,s2)
  )
  +
  (is_standby(l2) && is_busy(l1) && dest(l2_cmd) > dest(l1_cmd)) -> (
      s_sendLift(s2,2,l2_cmd).
      motion_mng(l1_cmd,l2_cmd,l1,busy,priority,detour,dt,s1,s2)
  )
;

init
allow({startmotor,stopmotor,
       shuttleLoad,shuttleUnload,
       liftLoad,liftUnload,
       liftLoadInput,liftUnloadOutput,
       sendLift,liftArrived,
       outReady,hasInput,
       transFree,
       tsinput,tsoutput,
       sendCmds,sendCmd,cmdDone,
       incycle,outcycle,seqcycle,doublecycle,
       step,setlat,getlat,
       inReady,loadedInput,liftReady,
       result,tick,print_lat,print_idle
       },
  comm({s_liftArrived|r_liftArrived->liftArrived,
       s_sendLift|r_sendLift->sendLift,
       s_startmotor|r_startmotor->startmotor,
       s_stopmotor|r_stopmotor->stopmotor,
       s_step|r_step->step,
       s_setlat|r_setlat->setlat,
       s_getlat|r_getlat->getlat,
       s_loadedInput|r_loadedInput->loadedInput,
       s_inReady|r_inReady->inReady,
       s_result|r_result->result,
       s_print_idle|r_print_idle->print_idle,
       s_liftReady|r_liftReady->liftReady,
       s_tick|r_tick->tick,
       s_shuttleLoad|r_shuttleLoad->shuttleLoad,
       s_shuttleUnload|r_shuttleUnload->shuttleUnload,
       r_tsoutput|s_tsoutput->tsoutput,
       r_tsinput|s_tsinput->tsinput,
       r_incycle|s_incycle->incycle,
       r_outcycle|s_outcycle->outcycle,
       r_seqcycle|s_seqcycle->seqcycle,
       r_doublecycle|s_doublecycle->doublecycle,
       r_sendCmd|s_sendCmd->sendCmd,
       r_sendCmds|s_sendCmds->sendCmds,
       r_cmdDone|s_cmdDone->cmdDone,
       r_hasInput|s_hasInput->hasInput,
       r_transFree|s_transFree->transFree,
       r_outReady|s_outReady->outReady
       },
monitor(0) ||
%liftplatform(id,pos,dest,busy,action,endtime,idle)
liftplatform(1,BOTTOM,BOTTOM,false,move,0,0) ||
liftplatform(2,LOCS,LOCS,false,move,0,0) ||
%stepper(pos1,pos2,dir1,dir2,lat,p1,p2,s1,s2)
stepper(BOTTOM,LOCS,neutral,neutral,0,NO_LOC,NO_LOC,idle,idle)||
mast(lidle,lidle,[],[],BOTTOM,LOCS,0,0,0,0,0,0,0,0) ||
motion_mng(pair(BOTTOM,move),pair(LOCS,move),
         lidle,lidle,1,pair(NO_LOC,move),0,0,0) ||
lift_agent(pair(false,BOTTOM),1) ||
lift_agent(pair(false,LOCS),2) ||
shuttle_agent ||
transfer_agent({},{},{},[],LOWPRIOR,0,[]) ||
delivery([0,0,0,0,0],0)));

map inRng:  location#location#location->Bool;
var p,max,l:  location;
eqn (l<max)-> inRng(p,max,l) = if(p==l,true,inRng(p,max,l+LOCMOD));
    (l==max) -> inRng(p,max,l) = p==l;

map ibCheck:  lift_id#location#Set(location)#Set(location)->location;
var id: lift_id;
    l: location; % requested location
    ib, ic: Set(location);
eqn (l!=NO_LOC) -> ibCheck(id,l,ib,ic) =
```

```
      if(!(l in ib)&&!(l in ic)&&if(id==1,l!=LOCS,l!=BOTTOM),l,NO_LOC);
    (l==NO_LOC) -> ibCheck(id,l,ib,ic) = getFree(id,ib,ic,BOTTOM);
map getFree: lift_id#Set(location)#Set(location)#location -> location;
var id: lift_id;
    ib, ic: Set(location);
    l: location;
eqn (l<LOCS) -> getFree(id,ib,ic,l) =
      if(!(l in ib)&&!(l in ic)&&if(id==1,l!=LOCS,l!=BOTTOM),l,getFree(id,ib,ic,l+LOCMOD));
    (l==LOCS) -> getFree(id,ib,ic,l) =
      if(!(l in ib)&&!(l in ic)&&if(id==1,l!=LOCS,l!=BOTTOM),l,NO_LOC);

map canStore: lift_id#Set(location)#Set(location)#location -> Bool;
var ib, ic: Set(location);
    id: lift_id;
    l: location;
eqn (l<LOCS) -> canStore(id,ib,ic,l) =
      if(!(l in ib)&&!(l in ic)&&
        if(id==1,l!=LOCS,l!=BOTTOM),true,canStore(id,ib,ic,l+LOCMOD));
    (l==LOCS) -> canStore(id,ib,ic,l) =
      !(l in ib)&&!(l in ic)&&if(id==1,l!=LOCS,l!=BOTTOM);

map reachable: lift_id#location -> Bool;
var id: lift_id;
    destination: location;
eqn reachable(1,destination) = destination<LOCS;
    reachable(2,destination) = destination>BOTTOM;

map bothreachable: lift_id#location -> Bool;
var id: lift_id;
    destination: location;
eqn bothreachable(id,destination) =
      reachable(1,destination) &&
      reachable(2,destination);

map mereachable: lift_id#location -> Bool;
var id: lift_id;
    destination: location;
eqn mereachable(id,destination) =
      reachable(id,destination) &&
      !reachable(if(id==1,2,1),destination);

map otherreachable: lift_id#location -> Bool;
var id: lift_id;
    destination: location;
eqn otherreachable(id,destination) =
      !reachable(id,destination) &&
      reachable(if(id==1,2,1),destination);

% selection-tree algorithm of VI
map getNextout: lift_id#List(retrieval_state) -> location;
var rsl: List(retrieval_state);
    rs: retrieval_state;
    id: lift_id;
eqn getNextout(id,[]) = NO_LOC;
    (!claimed(rs)&&bothreachable(id,loc(rs))) ->
      getNextout(id,rs|>rsl) =
        getNextout2(id,rsl,loc(rs));
    (!claimed(rs)&&mereachable(id,loc(rs))) ->
      getNextout(id,rs|>rsl) =  loc(rs);
    (!claimed(rs)&&otherreachable(id,loc(rs))) ->
      getNextout(id,rs|>rsl) =
        firstOldestReachable(id,rsl);
    claimed(rs) ->
      getNextout(id,rs|>rsl) = getNextout(id,rsl);

map getNextout2: lift_id#List(retrieval_state)#location -> location;
var rsl: List(retrieval_state);
    rs: retrieval_state;
    fst_candidate: location;
    id: lift_id;
eqn getNextout2(id,[],fst_candidate) = NO_LOC; % nothing left unclaimed
    (!claimed(rs)&&bothreachable(id,loc(rs))) ->
      getNextout2(id,rs|>rsl,fst_candidate) =
        if(id==2, max(loc(rs),fst_candidate), min(loc(rs),fst_candidate));
    (!claimed(rs)&&mereachable(id,loc(rs))) ->
      getNextout2(id,rs|>rsl,fst_candidate) = loc(rs);
    (!claimed(rs)&&otherreachable(id,loc(rs))) ->
      getNextout2(id,rs|>rsl,fst_candidate) = fst_candidate;
    (claimed(rs)) ->
      getNextout2(id,rs|>rsl,fst_candidate) =
        getNextout2(id,rsl,fst_candidate);

map firstOldestReachable: lift_id#List(retrieval_state) -> location;
var rsl: List(retrieval_state);
    rs: retrieval_state;
    id: lift_id;
eqn firstOldestReachable(id,[]) = NO_LOC;
```

```
    firstOldestReachable(id,rs|>rsl) =
      if(!claimed(rs)&&mereachable(id,loc(rs)),
          loc(rs),firstOldestReachable(id,rsl));

map setClaimed: List(retrieval_state)#location -> List(retrieval_state);
var rsl: List(retrieval_state);
    rs: retrieval_state;
    l: location;
eqn setClaimed([],l) = [];
    (l==NO_LOC) -> setClaimed(rsl,l) = rsl;
    (l!=NO_LOC) -> setClaimed(rs|>rsl,l) =
      if(l==loc(rs),pair(loc(rs),true)|>rsl,rs|>setClaimed(rsl,l));

map rmRelease: List(retrieval_state)#location -> List(retrieval_state);
var rsl: List(retrieval_state);
    rs: retrieval_state;
    l: location;
eqn rmRelease([],l) = [];
    rmRelease(rs|>rsl,l) = if(loc(rs)==l,rsl,rs|>rmRelease(rsl,l));

% l is assumed to be in the list as the loc element of loctime struct pair
map getOuttime: List(loctime)#location -> time;
var rlt: List(loctime);
    lt: loctime;
    l: location;
eqn getOuttime(lt|>rlt,l) = if(l==loc(lt),t(lt),getOuttime(rlt,l));

% l is assumed to be in the list as the loc element of loctime struct pair
map rmOuttime: List(loctime)#location -> List(loctime);
var rlt: List(loctime);
    lt: loctime;
    l:location;
eqn rmOuttime(lt|>rlt,l) = if(l==loc(lt),rlt,lt|>rmOuttime(rlt,l));

% handling input in progress by mast layer?
map isHandlingInput: List(lift_cmd) -> Bool;
var llt: List(lift_cmd);
    lt: lift_cmd;
eqn isHandlingInput(lt|>llt) =
      if(action(lt)==unloadtransfer||
        action(lt)==loadunloadtransfer,true,isHandlingInput(llt));
    isHandlingInput([]) = false;
```

A.4 Highly Dynamic Storage system priority version

```
sort
    lift_id = Pos;
    location = Pos;
    time = Nat;
    priority = Nat;
    direction = struct up?is_up |
                    down?is_down |
                    neutral?is_neutral;
    lift_action = struct loadtransfer?is_loadtransfer |
                    unloadtransfer?is_unloadtransfer |
                    loadunloadtransfer?is_loadunloadtransfer |
                    loadinput?is_loadinput |
                    unloadoutput?is_unloadoutput |
                    move?is_move;
    lift_cmd = struct pair(dest:location, action:lift_action);
    step_action = struct idle?is_idle |
                      stepping?is_stepping |
                      arrived?is_arrived;
    retrieval_state = struct pair(loc:location, claimed:Bool);
    lift_state = struct pair(inMotion:Bool,loc:location);
    lift_status = struct lidle?is_lidle | % no assignments left
                      busy?is_busy | % executing an assignment
                      standby?is_standby; % ready for next assignment

map CONV_POS: location;
eqn CONV_POS = 2;

map LOCS: location;
eqn LOCS = 3;

map BOTTOM: location;
eqn BOTTOM = 1;

map NO_LOC: location;
eqn NO_LOC = LOCS+1;

map   setDir: location#location -> direction;
var   curloc,dest: location;
eqn   setDir(curloc,dest) = if(curloc==dest,neutral,
          if(dest>curloc,up,down));
```

```
map P2:time; eqn P2=2;
map P3:time; eqn P3=3;
map P4:time; eqn P4=4;
map P5:time; eqn P5=5;

act
  s_startmotor,r_startmotor,startmotor: time#lift_id#location;
  s_stopmotor,r_stopmotor,stopmotor: time#lift_id;
  s_step,r_step,step: time#lift_id#location;

  s_sendLift,r_sendLift,sendLift: time#lift_id#lift_cmd;
  s_liftArrived,r_liftArrived,liftArrived: time#lift_id;

  r_loadedInput,s_loadedInput,loadedInput: time;
  s_liftReady,r_liftReady,liftReady: time#lift_id;
  s_inReady,r_inReady,inReady: time#lift_id; % current-time

  s_shuttleLoad,r_shuttleLoad,shuttleLoad: time#location;
  s_shuttleUnload,r_shuttleUnload,shuttleUnload: time#location;

  r_transFree,s_transFree,transFree: time#location#lift_id#location;
  r_outReady,s_outReady,outReady: time#lift_id#location;

  r_tsoutput,s_tsoutput,tsoutput: time#location;
  r_tsinput,s_tsinput,tsinput: time#location;

  r_incycle,s_incycle,incycle: time#lift_id#location;
  r_outcycle,s_outcycle,outcycle: time#lift_id#location;
  r_seqcycle,s_seqcycle,seqcycle: time#lift_id#location#location;
  r_doublecycle,s_doublecycle,doublecycle: time#lift_id#location;

  r_sendCmd,s_sendCmd,sendCmd: time#lift_id#lift_cmd;
  r_sendCmds,s_sendCmds,sendCmds: time#lift_cmd#lift_cmd;
  r_cmdDone,s_cmdDone,cmdDone: time#lift_id;

  r_hasInput,s_hasInput,hasInput;
  liftLoad, liftUnload, liftLoadInput, liftUnloadOutput: time#lift_id;

proc

delivery =
    s_hasInput.
    r_loadedInput(P3).
    delivery
;

liftplatform(id: lift_id, c:location, d:location, busy:Bool, a:lift_action) =
!busy ->
sum cmd:lift_cmd.
r_sendLift(P3,id,cmd).
s_startmotor(P3,id,dest(cmd)).
liftplatform(id,c,dest(cmd),true,action(cmd))
+
(busy && c<d) ->
sum t:time.s_step(t,id,c+1).liftplatform(id,c+1,d,busy,a)
+
(busy && c>d) ->
sum t:time.s_step(t,id,Int2Pos(c-1)).liftplatform(id,Int2Pos(c-1),d,busy,a)
+
(busy && c==d) ->
sum t:Nat. r_stopmotor(P3,id).  (
  is_loadinput(a)->
    liftLoadInput(P4,id).
    s_loadedInput(P3).
    s_liftArrived(P3,id).
    liftplatform(id,c,d,false,a)
  +
  (is_move(a))->
    s_liftArrived(P3,id).
    liftplatform(id,c,d,false,a)
  +
  (is_loadtransfer(a))->
    liftLoad(P4,id).
    s_tsoutput(P3,d).
    s_liftArrived(P3,id).
    liftplatform(id,c,d,false,a)
  +
  (is_unloadtransfer(a))->
    liftUnload(P4,id).
    s_tsinput(P3,d).
    s_liftArrived(P3,id).
    liftplatform(id,c,d,false,a)
  +
  (is_unloadoutput(a))->
    liftUnloadOutput(P4,id).
    s_liftArrived(P3,id).
```

```
    liftplatform(id,c,d,false,a)
  +
  (is_loadunloadtransfer(a))->
    liftLoad(P4,id).liftUnload(P4,id).
    s_tsinput(P3,d).
    s_tsoutput(P3,d).
    s_liftArrived(P3,id).
    liftplatform(id,c,d,false,a)
)
;

stepper(c1:location, c2:location, m1:direction, m2:direction,
    p1:location, p2:location,
    s1:step_action, s2:step_action) =
(!is_arrived(s1) && !is_arrived(s2)) -> (
(is_neutral(m2)&&is_up(m1)) ->
  sum d:location.r_step(P5,1,d).
  stepper(c1+1,c2,m1,m2,p1,p2,
        if(c1+1==p1&&is_stepping(s1),arrived,s1),s2)
+
(is_neutral(m2)&&is_down(m1)) ->
  sum d:location.r_step(P5,1,d).
  stepper(Int2Pos(c1-1),c2,m1,m2,p1,p2,
        if(Int2Pos(c1-1)==p1&&is_stepping(s1),arrived,s1),s2)
+
(is_neutral(m1)&&is_up(m2)) ->
  sum d:location.r_step(P5,2,d).
  stepper(c1,c2+1,m1,m2,p1,p2,
        s1,if(c2+1==p2&&is_stepping(s2),arrived,s2))
+
(is_neutral(m1)&&is_down(m2)) ->
  sum d:location.r_step(P5,2,d).
  stepper(c1,Int2Pos(c2-1),m1,m2,p1,p2,
        s1,if(Int2Pos(c2-1)==p2&&is_stepping(s2),arrived,s2))
+
(is_up(m1)&&is_up(m2)) -> (
  sum d2:location.r_step(P5,2,d2).
  sum d1:location.r_step(P5,1,d1).
  stepper(c1+1,c2+1,m1,m2,p1,p2,
        if(c1+1==p1&&is_stepping(s1),arrived,s1),
        if(c2+1==p2&&is_stepping(s2),arrived,s2))
  )
+
(is_down(m1)&&is_down(m2)) -> (
  sum d1:location.r_step(P5,1,d1).
  sum d2:location.r_step(P5,2,d2).
  stepper(Int2Pos(c1-1),Int2Pos(c2-1),m1,m2,p1,p2,
        if(Int2Pos(c1-1)==p1&&is_stepping(s1),arrived,s1),
        if(Int2Pos(c2-1)==p2&&is_stepping(s2),arrived,s2))
  )
+
(is_down(m1)&&is_up(m2)) -> (
  sum d1:location.r_step(P5,1,d1).
  sum d2:location.r_step(P5,2,d2).
  stepper(Int2Pos(c1-1),c2+1,m1,m2,p1,p2,
        if(Int2Pos(c1-1)==p1&&is_stepping(s1),arrived,s1),
        if(c2+1==p2&&is_stepping(s2),arrived,s2))
  )
+
(is_up(m1)&&is_down(m2)) -> (
  sum d1:location.r_step(P5,1,d1).
  sum d2:location.r_step(P5,2,d2).
  stepper(c1+1,Int2Pos(c2-1),m1,m2,p1,p2,
        if(c1+1==p1&&is_stepping(s1),arrived,s1),
        if(Int2Pos(c2-1)==p2&&is_stepping(s2),arrived,s2))
  )
)
+
sum dest:location,id:lift_id.
  r_startmotor(P3,id,dest).
stepper(c1,c2,
      if(id==1,setDir(c1,dest),m1),
        if(id==2,setDir(c2,dest),m2),
        if(id==1,dest,p1),if(id==2,dest,p2),
        if(id==1,stepping,s1),if(id==2,stepping,s2))
+
sum id:lift_id.
s_stopmotor(P3,id).
stepper(c1,c2,
        if(id==1,neutral,m1),
        if(id==2,neutral,m2),
        p1,p2,
        if(id==1,idle,s1),if(id==2,idle,s2))
;

shuttle_agent =
sum l:location. (
```

```
  s_shuttleUnload(P5,l)
+
  s_shuttleLoad(P5,l)
).shuttle_agent
;

% il: input buffers
% ol: output buffers
% ic: claimed input buffers (not full but reserved)
% rl: retrieval status of output buffers (FIFO order)
transfer_agent(il:Set(location), ol:Set(location),
               ic:Set(location), rl:List(retrieval_state),
               t_in:Bool) =
% Shuttle delivers tote to a (free) output buffer
sum l:location.
(!(l in ol)&&inRng(l,LOCS,BOTTOM)) ->
  r_shuttleUnload(P5,l).transfer_agent(il,ol+{l},ic,rl<|pair(l,false),t_in)
+
% Shuttle retrieves tote from (full) input buffer
sum l:location.
(l in il) ->
  r_shuttleLoad(P5,l).transfer_agent(il-{l},ol,ic-{l},rl,t_in)
+
% Free input buffer request
% case 1: reqloc is free and honoured
% case 2: reqloc is full, an alternative is presented
% case 3: all input (reachable) input buffers are full, NO_LOC is returned
sum reqloc:location,lid:lift_id.
sum inloc:location.(inloc==ibCheck(lid,reqloc,il,ic)) ->
r_transFree(P2,reqloc,lid,inloc).
transfer_agent(il,ol,ic+{inloc},rl,t_in)
+
% Request which output buffer to retrieve next
sum out:location.(out==getNextout(1,rl)&&out!=NO_LOC) ->
s_outReady(P2,1,out).
transfer_agent(il,ol,ic,setClaimed(rl,out),t_in)
+
% Request which output buffer to retrieve next
sum out:location.(out==getNextout(2,rl)&&out!=NO_LOC) ->
s_outReady(P2,2,out).
transfer_agent(il,ol,ic,setClaimed(rl,out),t_in)
+
% An output buffer is emptied (tote loaded from buffer)
sum loc:location.
r_tsoutput(P3,loc).
transfer_agent(il,ol-{loc},ic,rmRelease(rl,loc),t_in)
+
% An input buffer is now full (tote unloaded onto buffer)
sum loc:location.
r_tsinput(P3,loc).
transfer_agent(il+{loc},ol,ic,rl,t_in)
+
% new input available
r_hasInput.%(P1).
transfer_agent(il,ol,ic,rl,true)
+
% Check if a new input can be stored by lift 1
(t_in&&canStore(1,il,ic,BOTTOM)) ->
s_inReady(P2,1).
transfer_agent(il,ol,ic,rl,false)
+
% Check if a new input can be stored by lift 2
(t_in&&canStore(2,il,ic,BOTTOM)) ->
s_inReady(P2,2).
transfer_agent(il,ol,ic,rl,false)
;

lift_agent(state:lift_state, id: lift_id) =
  % alleen runnen als de opdracht verwerkt is dus als !inMotion(state)
  inMotion(state) ->
    r_liftReady(P3,id).lift_agent(pair(false,loc(state)),id)
  +
  % @conv_pos and ready to do some work
  (!inMotion(state)&&loc(state)==CONV_POS) -> (
    sum nextout:location.
    r_outReady(P2,id,nextout). (
      (
      r_inReady(P2,id).
      sum nextin:location.s_transFree(P2,NO_LOC,id,nextin).
      (nextin==nextout) ->
        s_doublecycle(P3,id,nextout).lift_agent(pair(true,nextout),id)
      <>
        s_seqcycle(P3,id,nextout,nextin).lift_agent(pair(true,nextin),id)
      )
      +
      s_outcycle(P3,id,nextout).lift_agent(pair(true,nextout),id)
    )
```

```
    +
    r_inReady(P2,id).
    sum nextin:location.s_transFree(P2,NO_LOC,id,nextin).
    s_incycle(P3,id,nextin).lift_agent(pair(true,nextin),id)
  )
  +
  % not @conv_pos and ready to do some work
  (!inMotion(state)&&loc(state)!=CONV_POS) -> (
    sum nextout:location.
    r_outReady(P2,id,nextout).
    s_outcycle(P3,id,nextout).lift_agent(pair(true,nextout),id)
    +
    r_inReady(P2,id).
    sum nextin:location.s_transFree(P2,NO_LOC,id,nextin).
    s_incycle(P3,id,nextin).lift_agent(pair(true,nextin),id)
  )
;

mast(l1:lift_status, l2:lift_status, l1_work:List(lift_cmd), l2_work:List(lift_cmd),
    l1_loc: location, l2_loc: location) =
  % lift_agent signals a input-cycle
  sum dest:location, id:lift_id.
    r_incycle(P3,id,dest).
    mast(if(id==1,standby,l1), if(id==2,standby,l2),
          if(id==1, [pair(CONV_POS,loadinput),pair(dest,unloadtransfer)], l1_work),
        if(id==2, [pair(CONV_POS,loadinput),pair(dest,unloadtransfer)], l2_work),
        l1_loc, l2_loc)
  +
  % lift_agent signals a output-cycle
  sum dest:location, id:lift_id.
    r_outcycle(P3,id,dest).
    mast(if(id==1,standby,l1), if(id==2,standby,l2),
        if(id==1, [pair(dest,loadtransfer),pair(CONV_POS,unloadoutput)], l1_work),
        if(id==2, [pair(dest,loadtransfer),pair(CONV_POS,unloadoutput)], l2_work),
        l1_loc, l2_loc)
  +
  % lift_agent signals a sequential-cycle
  sum destin, destout:location, id:lift_id.
    r_seqcycle(P3,id,destout,destin).
    mast(if(id==1,standby,l1), if(id==2,standby,l2),
        if(id==1, [pair(CONV_POS,loadinput),pair(destin,unloadtransfer),
                pair(destout,loadtransfer),pair(CONV_POS,unloadoutput)],
                l1_work),
        if(id==2, [pair(CONV_POS,loadinput),pair(destin,unloadtransfer),
                pair(destout,loadtransfer),pair(CONV_POS,unloadoutput)],
                l2_work),
        l1_loc, l2_loc)
  +
  % lift_agent signals a double-cycle
  sum dest: location, id:lift_id.
    r_doublecycle(P3,id,dest).
    mast(if(id==1,standby,l1), if(id==2,standby,l2),
        if(id==1, [pair(CONV_POS,loadinput),pair(dest,loadunloadtransfer),
                pair(CONV_POS,unloadoutput)], l1_work),
        if(id==2, [pair(CONV_POS,loadinput),pair(dest,loadunloadtransfer),
                pair(CONV_POS,unloadoutput)], l2_work),
        l1_loc, l2_loc)
  +
  % request the next order for both l1,l2
  (is_standby(l1)&&l1_work!=[] && is_standby(l2)&&l2_work!=[]) ->
    s_sendCmds(P3,head(l1_work),head(l2_work)).
    mast(busy,busy,l1_work,l2_work,l1_loc,l2_loc)
  +
  % request the next order for l1 only, l2 busy or idle
  (is_standby(l1)&&l1_work!=[] && !is_standby(l2)) ->
    s_sendCmd(P3,1,head(l1_work)).
    mast(busy,l2,l1_work,l2_work,l1_loc,l2_loc)
  +
  % request the next order for l2 only, l1 busy or idle
  (is_standby(l2)&&l2_work!=[] && !is_standby(l1)) ->
    s_sendCmd(P3,2,head(l2_work)).
    mast(l1,busy,l1_work,l2_work,l1_loc,l2_loc)
  +
  % got signal that previous cmd is now executed
  sum id:lift_id.
    r_cmdDone(P3,id). (
      (id==1&&#l1_work==1) -> ( % final cmd is now processed for l1
        % signal lift_agent (1) to determine next output
        s_liftReady(P3,1).
        mast(lidle,l2,[],l2_work,dest(head(l1_work)),l2_loc)
      )
      +
      (id==1&&#l1_work>1) -> ( % more cmd's to process for l1
        mast(standby,l2,tail(l1_work),l2_work,dest(head(l1_work)),l2_loc)
      )
      +
      (id==2&&#l2_work==1) -> ( % final cmd is now processed for l4
```

```
        % signal lift_agent (2) to determine next output
        s_liftReady(P3,2).
        mast(l1,lidle,l1_work,[],l1_loc,dest(head(l2_work)))
      )
      +
      (id==2&&#l2_work>1) -> ( % more cmd's to process for l2
        mast(l1,standby,l1_work,tail(l2_work),l1_loc,dest(head(l2_work)))
      )
    )
;

% accepts one order at a time from each lift,
% and makes sure that no collisssions should occur
% it will then forward the command to the hardware (liftplatform/stepper)
% also waiting for the arrive (and load handling) of a lift
% s_cmdDone(id) : signal lift_agent that the current command has been executed
% r_liftArrived(id) : receive signal that hardware has executed the command entirely
motion_mng(l1_cmd: lift_cmd, l2_cmd:lift_cmd, l1:lift_status,
        l2:lift_status, priority:lift_id, detour:lift_cmd, dt:Nat) =
  % both lifts report a new order (from mast)
  (is_lidle(l1)&&is_lidle(l2)) ->
    sum cmd1, cmd2: lift_cmd.
    r_sendCmds(P3,cmd1,cmd2).
    motion_mng(cmd1,cmd2,standby,standby,priority,detour,dt)
  +
  % lift (id) reports a new order (from mast)
  sum cmd: lift_cmd, id:lift_id.
    r_sendCmd(P3,id,cmd). (
      (id==1 && is_busy(l1) && is_move(action(l1_cmd))) ->
        motion_mng(l1_cmd,l2_cmd,l1,l2,priority,cmd,1)
      +
      (id==1 && is_lidle(l1)) ->
        motion_mng(cmd,l2_cmd,standby,l2,priority,detour,dt)
      +
      (id==2 && is_busy(l2) && is_move(action(l2_cmd))) ->
        motion_mng(l1_cmd,l2_cmd,l1,l2,priority,cmd,2)
      +
      (id==2 && is_lidle(l2)) ->
        motion_mng(l1_cmd,cmd,l1,standby,priority,detour,dt)
    )
  +
  % lift 1 reports arriving at the scene
  r_liftArrived(P3,1). (
    is_move(action(l1_cmd)) -> (
      motion_mng(if(dt==1,detour,l1_cmd),l2_cmd,
              if(dt==1,standby,lidle),l2,
              priority,detour,0)
    )
    +
    !is_move(action(l1_cmd)) -> (
      s_cmdDone(P3,1).
      motion_mng(l1_cmd,l2_cmd,lidle,l2,priority,detour,dt)
    )
  )
  +
  % lift 2 reports arriving at the scene
  r_liftArrived(P3,2). (
    is_move(action(l2_cmd)) -> (
      motion_mng(l1_cmd,if(dt==2,detour,l2_cmd),
              l1,if(dt==2,standby,lidle),
              priority,detour,0)
    )
    +
    !is_move(action(l2_cmd)) -> (
      s_cmdDone(P3,2).
      motion_mng(l1_cmd,l2_cmd,l1,lidle,priority,detour,dt)
    )
  )
  % Send commands away, keeping priority in mind
  % This process makes sure that cmd's do not
  % result in physical deadlocks where both
  % lifts want to cross each other
  +
  (is_standby(l1) && is_standby(l2)) -> (
    (dest(l1_cmd) < dest(l2_cmd)) ->
      s_sendLift(P3,2,l2_cmd).s_sendLift(P3,1,l1_cmd).
      motion_mng(l1_cmd,l2_cmd,busy,busy,priority,detour,dt)
    +
    (dest(l1_cmd) >= dest(l2_cmd)) ->
      s_sendLift(P3,if(priority==1,2,1),
              pair(if(priority==1,dest(l1_cmd)+1,Int2Pos(dest(l2_cmd)-1)),move)).
      s_sendLift(P3,if(priority==1,1,2),
              if(priority==1,l1_cmd,l2_cmd)).
      motion_mng(if(priority==1,l1_cmd,pair(Int2Pos(dest(l2_cmd)-1),move)),
              if(priority==2,l2_cmd,pair(dest(l1_cmd)+1,move)),
              busy,busy,if(priority==1,2,1),
              if(priority==1,l2_cmd,l1_cmd),
```

```
                if(priority==1,2,1))
  )
  +
  (is_standby(l1) && is_lidle(l2)) -> (
    (dest(l1_cmd) < dest(l2_cmd)) ->
      s_sendLift(P3,1,l1_cmd).
      motion_mng(l1_cmd,l2_cmd,busy,l2,priority,detour,dt)
    +
    (dest(l1_cmd) >= dest(l2_cmd)) ->
      s_sendLift(P3,2,pair(dest(l1_cmd)+1,move)).s_sendLift(P3,1,l1_cmd).
      motion_mng(l1_cmd,pair(dest(l1_cmd)+1,move),busy,busy,2,detour,dt)
  )
  +
  (is_standby(l1) && is_busy(l2) && dest(l1_cmd) < dest(l2_cmd)) -> (
      s_sendLift(P3,1,l1_cmd).
      motion_mng(l1_cmd,l2_cmd,busy,l2,priority,detour,dt)
  )
  +
  (is_standby(l2) && is_lidle(l1)) -> (
    (dest(l2_cmd) > dest(l1_cmd)) ->
      s_sendLift(P3,2,l2_cmd).
      motion_mng(l1_cmd,l2_cmd,l1,busy,priority,detour,dt)
    +
    (dest(l2_cmd) <= dest(l1_cmd)) ->
      s_sendLift(P3,1,pair(Int2Pos(dest(l2_cmd)-1),move)).
      s_sendLift(P3,2,l2_cmd).
      motion_mng(pair(Int2Pos(dest(l2_cmd)-1),move),l2_cmd,
                busy,busy,1,detour,dt)
  )
  +
  (is_standby(l2) && is_busy(l1) && dest(l2_cmd) > dest(l1_cmd)) -> (
      s_sendLift(P3,2,l2_cmd).
      motion_mng(l1_cmd,l2_cmd,l1,busy,priority,detour,dt)
  )
;

init
allow({startmotor,stopmotor,
       shuttleLoad,
       shuttleUnload,
       liftLoad,liftUnload,
       liftLoadInput,liftUnloadOutput,
       sendLift,liftArrived,
       sendLift|sendLift,
       outReady,hasInput,
       transFree,
       tsinput,tsoutput,
       sendCmds,sendCmd,cmdDone,
       incycle,outcycle,seqcycle,doublecycle,
       step,inReady,loadedInput,liftReady
       },
  comm({s_liftArrived|r_liftArrived->liftArrived,
       s_sendLift|r_sendLift->sendLift,
       s_startmotor|r_startmotor->startmotor,
       s_stopmotor|r_stopmotor->stopmotor,
       s_step|r_step->step,
       s_loadedInput|r_loadedInput->loadedInput,
       s_inReady|r_inReady->inReady,
       s_liftReady|r_liftReady->liftReady,
       s_shuttleLoad|r_shuttleLoad->shuttleLoad,
       s_shuttleUnload|r_shuttleUnload->shuttleUnload,
       r_tsoutput|s_tsoutput->tsoutput,
       r_tsinput|s_tsinput->tsinput,
       r_incycle|s_incycle->incycle,
       r_outcycle|s_outcycle->outcycle,
       r_seqcycle|s_seqcycle->seqcycle,
       r_doublecycle|s_doublecycle->doublecycle,
       r_sendCmd|s_sendCmd->sendCmd,
       r_sendCmds|s_sendCmds->sendCmds,
       r_cmdDone|s_cmdDone->cmdDone,
       r_hasInput|s_hasInput->hasInput,
       r_transFree|s_transFree->transFree,
       r_outReady|s_outReady->outReady
       },
%liftplatform(id,pos,dest,busy,action)
liftplatform(1,BOTTOM,BOTTOM,false,move) ||
liftplatform(2,LOCS,LOCS,false,move) ||
%stepper(pos1,pos2,dir1,dir2,p1,p2,s1,s2)
stepper(BOTTOM,LOCS,neutral,neutral,NO_LOC,NO_LOC,idle,idle)||
mast(lidle,lidle,[],[],BOTTOM,LOCS) ||
motion_mng(pair(BOTTOM,move),pair(LOCS,move),
          lidle,lidle,1,pair(NO_LOC,move),0) ||
lift_agent(pair(false,BOTTOM),1) ||
lift_agent(pair(false,LOCS),2) ||
shuttle_agent ||
transfer_agent({},{},{},[],false) ||
delivery
```

```
));

map inRng:  location#location#location->Bool;
var p,max,l:  location;
eqn (l<max)-> inRng(p,max,l) = if(p==l,true,inRng(p,max,l+1));
    (l==max) -> inRng(p,max,l) = p==l;

map ibCheck:  lift_id#location#Set(location)#Set(location)->location;
var id: lift_id;
    l: location; % requested location
    ib, ic: Set(location);
eqn (l!=NO_LOC) -> ibCheck(id,l,ib,ic) =
      if(!(l in ib)&&!(l in ic)&&if(id==1,l!=LOCS,l!=BOTTOM),l,NO_LOC);
    (l==NO_LOC) -> ibCheck(id,l,ib,ic) = getFree(id,ib,ic,BOTTOM); % BOTTOM or 1||3 ?
map getFree: lift_id#Set(location)#Set(location)#location -> location;
var id: lift_id;
    ib, ic: Set(location);
    l: location;
eqn (l<LOCS) -> getFree(id,ib,ic,l) =
      if(!(l in ib)&&!(l in ic)&&if(id==1,l!=LOCS,l!=BOTTOM),l,getFree(id,ib,ic,l+1));
    (l==LOCS) -> getFree(id,ib,ic,l) =
      if(!(l in ib)&&!(l in ic)&&if(id==1,l!=LOCS,l!=BOTTOM),l,NO_LOC);

map canStore: lift_id#Set(location)#Set(location)#location -> Bool;
var ib, ic: Set(location);
    id: lift_id;
    l: location;
eqn (l<LOCS) -> canStore(id,ib,ic,l) =
      if(!(l in ib)&&!(l in ic)&&
        if(id==1,l!=LOCS,l!=BOTTOM),true,canStore(id,ib,ic,l+1));
    (l==LOCS) -> canStore(id,ib,ic,l) =
      !(l in ib)&&!(l in ic)&&if(id==1,l!=LOCS,l!=BOTTOM);

map reachable: lift_id#location -> Bool;
var id: lift_id;
    destination: location;
eqn reachable(1,destination) = destination<LOCS;
    reachable(2,destination) = destination>BOTTOM;

map bothreachable: lift_id#location -> Bool;
var id: lift_id;
    destination: location;
eqn bothreachable(id,destination) =
      reachable(1,destination) &&
      reachable(2,destination);

map mereachable: lift_id#location -> Bool;
var id: lift_id;
    destination: location;
eqn mereachable(id,destination) =
      reachable(id,destination) &&
      !reachable(if(id==1,2,1),destination);

map otherreachable: lift_id#location -> Bool;
var id: lift_id;
    destination: location;
eqn otherreachable(id,destination) =
      !reachable(id,destination) &&
      reachable(if(id==1,2,1),destination);

% selection-tree algorithm of VI
map getNextout: lift_id#List(retrieval_state) -> location;
var rsl: List(retrieval_state);
    rs: retrieval_state;
    id: lift_id;
eqn getNextout(id,[]) = NO_LOC;
    (!claimed(rs)&&bothreachable(id,loc(rs))) ->
      getNextout(id,rs|>rsl) =
        getNextout2(id,rsl,loc(rs));
    (!claimed(rs)&&mereachable(id,loc(rs))) ->
      getNextout(id,rs|>rsl) =  loc(rs);
    (!claimed(rs)&&otherreachable(id,loc(rs))) ->
      getNextout(id,rs|>rsl) =
        firstOldestReachable(id,rsl);
    claimed(rs) ->
      getNextout(id,rs|>rsl) = getNextout(id,rsl);

map getNextout2: lift_id#List(retrieval_state)#location -> location;
var rsl: List(retrieval_state);
    rs: retrieval_state;
    fst_candidate: location;
    id: lift_id;
eqn getNextout2(id,[],fst_candidate) = fst_candidate;%NO_LOC; % nothing left unclaimed
    (!claimed(rs)&&bothreachable(id,loc(rs))) ->
      getNextout2(id,rs|>rsl,fst_candidate) =
        if(id==2, max(loc(rs),fst_candidate), min(loc(rs),fst_candidate)     );
    (!claimed(rs)&&mereachable(id,loc(rs))) ->
```

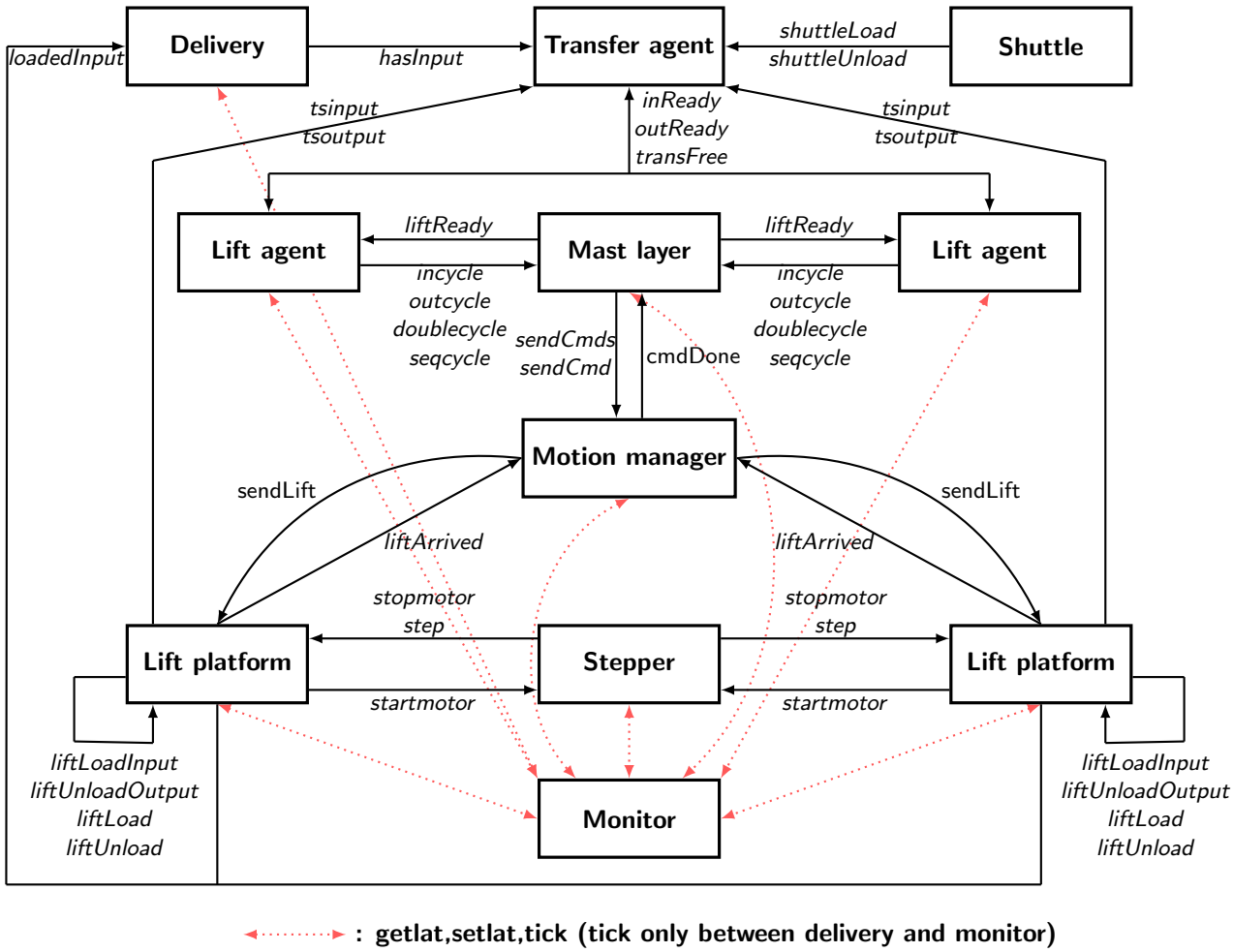
```
      getNextout2(id,rs|>rsl,fst_candidate) = loc(rs);
    (!claimed(rs)&&otherreachable(id,loc(rs))) ->
      getNextout2(id,rs|>rsl,fst_candidate) = fst_candidate;
    (claimed(rs)) ->
      getNextout2(id,rs|>rsl,fst_candidate) =
        getNextout2(id,rsl,fst_candidate);

map firstOldestReachable: lift_id#List(retrieval_state) -> location;
var rsl: List(retrieval_state);
    rs: retrieval_state;
    id: lift_id;
eqn firstOldestReachable(id,[]) = NO_LOC;
    firstOldestReachable(id,rs|>rsl) =
      if(!claimed(rs)&&mereachable(id,loc(rs)),
          loc(rs),firstOldestReachable(id,rsl));

map setClaimed: List(retrieval_state)#location -> List(retrieval_state);
var rsl: List(retrieval_state);
    rs: retrieval_state;
    l: location;
eqn setClaimed([],l) = [];
    (l==NO_LOC) -> setClaimed(rsl,l) = rsl;
    (l!=NO_LOC) -> setClaimed(rs|>rsl,l) =
      if(l==loc(rs),pair(loc(rs),true)|>rsl,rs|>setClaimed(rsl,l));

map rmRelease: List(retrieval_state)#location -> List(retrieval_state);
var rsl: List(retrieval_state);
    rs: retrieval_state;
    l: location;
eqn rmRelease([],l) = [];
    rmRelease(rs|>rsl,l) = if(loc(rs)==l,rsl,rs|>rmRelease(rsl,l));
```

A.5 Highly Dynamic Storage system schematic overview



Appendix B

Prioritized modal formulas

FR.1-p1: $\forall_{n,m:location}[true^* \cdot step(P_5, 1, m) \cdot \forall_{l:location,id:lift\_id}(\overline{step(P_5, id, l)})^* \cdot step(P_5, 2, n)](n > m)$

$\nu X \cdot ([hasInput]X \wedge$
$((((((}$
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{m:location}.[step(P_5, 1, m)](\alpha)$
$\vee\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)$
$startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id) \vee$
$liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true)$
$)$

$\alpha :=$
$\nu Y \cdot ([hasInput]Y \wedge$
$((((((}$
$\forall_{id:lift\_id,l:location}[shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$\wedge$
$(\forall_{n'':location}.[step(P_5, 2, n'')](n'' > m)$
$\vee\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)$
$startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id) \vee$
$liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true))$

FR.1-p2: $\forall_{n,m:location}[true^* \cdot step(P_5, 2, m) \cdot \forall_{l:location,id:lift\_id}(\overline{step(P_5, id, l)})^* \cdot step(P_5, 1, n)](n < m)$

$\nu X \cdot ([hasInput]X \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{m:location}.[step(P_5, 2, m)](\alpha)$
$\vee \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)$
$startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id) \vee$
$liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true)$
$)$

$\alpha :=$
$\nu Y \cdot ([hasInput]Y \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd')]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$\wedge$
$(\forall_{n'':location}.[step(P_5, 1, n'')](n'' < m)$
$\vee \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)$
$startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id) \vee$
$liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true))$

FR.2-p1: $\forall_{id:lift\_id}[true^* \cdot liftLoadInput(P_4, id) \cdot (\overline{liftUnload(P_4, id)})^* \cdot liftLoadInput(P_4, id)]false$

$\nu X \cdot ([hasInput]X \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{id':lift\_id}.[liftLoadInput(P_4, id')](\alpha)$
$\vee\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)$
$startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id) \vee \langle true)$
$)$

$\alpha :=$
$\nu Y \cdot ([hasInput]Y \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id,id2:lift\_id} val(id2 \neq id') \rightarrow$
$[liftLoad(P_4, id) \vee liftUnload(P_4, id2) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$\wedge$
$([liftLoadInput(P_4, id')](false)$
$\vee\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)$
$startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id) \vee \langle true)$)

FR.2-p2: $\forall_{id:lift\_id}[true^* \cdot liftLoad(P_4, id) \cdot (\overline{liftUnloadOutput(P_4, id)})^* \cdot liftLoad(P_4, id)]false$

$$
\begin{array}{l}
\nu X \cdot ([hasInput]X \wedge \\
((((((\\
\forall_{id:lift\_id,l:location}[step(P_5,id,l) \vee shuttleLoad(P_5,l) \vee shuttleUnload(P_5,l)]X \vee \\
\exists_{id:lift\_id}\langle liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id)\rangle true) \wedge \\
\quad \forall_{id:lift\_id}[liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id)]X) \vee \\
\quad\quad \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\quad\quad \langle startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\quad\quad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee \\
\quad\quad incycle(P_3,id,l) \vee outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee \\
\quad\quad sendCmd(P_3,id,cmd) \vee sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id)\rangle true) \wedge \\
\quad\quad\quad \forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\quad\quad\quad [startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\quad\quad\quad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee incycle(P_3,id,l) \vee \\
\quad\quad\quad outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee sendCmd(P_3,id,cmd) \vee \\
\quad\quad\quad sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id)]X) \vee \\
\quad\quad\quad\quad \exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id)\rangle true \\
) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id)]X) \\
\wedge \\
(\forall_{id':lift\_id}.[liftLoad(P_4,id')](\alpha) \\
\vee \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\quad \langle transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id) \\
\quad startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\quad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee \\
\quad incycle(P_3,id,l) \vee outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee \\
\quad sendCmd(P_3,id,cmd) \vee sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id) \vee \rangle true) \\
)
\end{array}
$$

$$
\begin{array}{l}
\alpha := \\
\nu Y \cdot ([hasInput]Y \wedge \\
((((((\\
\forall_{id:lift\_id,l:location}[step(P_5,id,l) || shuttleLoad(P_5,l) \vee shuttleUnload(P_5,l)]Y \vee \\
\exists_{id:lift\_id}\langle liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id)\rangle true) \wedge \\
\quad \forall_{id:lift\_id,id2:lift\_id} val(id2 \neq id') \rightarrow \\
\quad [liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id2)]Y) \vee \\
\quad\quad \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\quad\quad \langle startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\quad\quad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee \\
\quad\quad incycle(P_3,id,l) \vee outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee \\
\quad\quad sendCmd(P_3,id,cmd) \vee sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id)\rangle true) \wedge \\
\quad\quad\quad \forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\quad\quad\quad [startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\quad\quad\quad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee incycle(P_3,id,l) \vee \\
\quad\quad\quad outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee sendCmd(P_3,id,cmd) \vee \\
\quad\quad\quad sendCmds(P_3,cmd,cmd')]Y) \vee \\
\quad\quad\quad\quad \exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id)\rangle true \\
) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id)]Y) \\
\wedge \\
([liftLoad(P_4,id')](false) \\
\vee \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\quad \langle transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id) \\
\quad startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\quad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee \\
\quad incycle(P_3,id,l) \vee outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee \\
\quad sendCmd(P_3,id,cmd) \vee sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id) \vee \rangle true))
\end{array}
$$

FR.3-p1: $\forall_{l:location}[true^* \cdot tsinput(P_3, l) \cdot (\overline{shuttleLoad(P_5, l)})^* \cdot tsinput(P_3, l)]false$

$\nu X \cdot ([hasInput]X \wedge$
$((((((($
$\forall_{id:lift\_id, l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id, l:location, l':location, cmd:lift\_cmd, cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id, l:location, l':location, cmd:lift\_cmd, cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id, l:location, l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id, l:location, l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{d:location}.[tsinput(P_3, d)](\alpha)$
$\vee \exists_{id:lift\_id, l:location, l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true)$
$)$

$\alpha :=$
$\nu Y \cdot ([hasInput]Y \wedge$
$((((((($
$\forall_{id:lift\_id, l:location, d:location} val(d \neq d') \rightarrow$
$[step(P_5, id, l) \vee shuttleLoad(P_5, d') \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id, id2:location} val(id2 \neq id') \rightarrow$
$[liftLoad(P_4, id) \vee liftUnload(P_4, id2) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id, l:location, l':location, cmd:lift\_cmd, cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id, l:location, l':location, cmd:lift\_cmd, cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]Y) \vee$
$\exists_{id:lift\_id, l:location, l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id, l:location, l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$\wedge$
$([tsinput(P_3, d)](false)$
$\vee \exists_{id:lift\_id, l:location, l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true)$
$)$

FR.3-p2: $\forall_{l:location}[true^* \cdot shuttleUnload(P_5, l) \cdot (\overline{tsoutput(P_3, l)})^* \cdot shuttleUnload(P_5, l)]false$

$\nu X \cdot ([hasInput]X \wedge$
$((((((($
$\forall_{id:lift\_id, l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id, l:location, l':location, cmd:lift\_cmd, cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id, l:location, l':location, cmd:lift\_cmd, cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id, l:location, l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id, l:location, l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{d:location}.[shuttleUnload(P_5, d)](\alpha)$
$\vee\exists_{id:lift\_id, l:location, l':location, cmd:lift\_cmd, cmd':lift\_cmd}$
$\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)$
$startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id) \vee$
$liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true)$
$)$

$\alpha :=$
$\nu Y \cdot ([hasInput]Y \wedge$
$((((((($
$\forall_{id:lift\_id, l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id, id2:lift\_id} val(id2 \neq id') \rightarrow$
$[liftLoad(P_4, id) \vee liftUnload(P_4, id2) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id, l:location, l':location, cmd:lift\_cmd, cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id, l:location, l':location, cmd:lift\_cmd, cmd':lift\_cmd, d':location} val(d' \neq d) \rightarrow$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, d') \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]Y) \vee$
$\exists_{id:lift\_id, l:location, l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id, l:location, l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$\wedge$
$([shuttleUnload(P_5, d)](false)$
$\vee\exists_{id:lift\_id, l:location, l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true)$
$)$

FR.4: $[true^* \cdot hasInput]\mu X.[\overline{loadedInput(P_3)}]X$

$$
\begin{array}{l}
\nu X \cdot ([hasInput]X \wedge \\
((((((\\
\forall_{id:lift\_id,l:location}[step(P_5,id,l) \vee shuttleLoad(P_5,l) \vee shuttleUnload(P_5,l)]X \vee \\
\exists_{id:lift\_id}\langle liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id)\rangle true) \wedge \\
\quad \forall_{id:lift\_id}[liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id)]X) \vee \\
\qquad \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\qquad \langle startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\qquad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee \\
\qquad incycle(P_3,id,l) \vee outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee \\
\qquad sendCmd(P_3,id,cmd) \vee sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id)\rangle true) \wedge \\
\qquad\quad \forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\qquad\quad [startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\qquad\quad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee incycle(P_3,id,l) \vee \\
\qquad\quad outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee sendCmd(P_3,id,cmd) \vee \\
\qquad\quad sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id)]X) \vee \\
\qquad\quad\quad \exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id)\rangle true \\
) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id)]X) \\
\wedge \\
[hasInput](\alpha) \\
)
\end{array}
$$

$$
\begin{array}{l}
\alpha := \\
\mu Y \cdot ([hasInput]Y \wedge \\
((((((\\
\forall_{id:lift\_id,l:location}[step(P_5,id,l) \vee shuttleLoad(P_5,l) \vee shuttleUnload(P_5,l)]Y \vee \\
\exists_{id:lift\_id}\langle liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id)\rangle true) \wedge \\
\quad \forall_{id:lift\_id}[liftLoad(P_4,id) \vee liftUnload(P_4,id) \vee liftLoadInput(P_4,id) \vee liftUnloadOutput(P_4,id)]Y) \vee \\
\qquad \exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\qquad \langle startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\qquad loadedInput(P_3) \vee liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee \\
\qquad incycle(P_3,id,l) \vee outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee \\
\qquad sendCmd(P_3,id,cmd) \vee sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id)\rangle true) \wedge \\
\qquad\quad \forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd} \\
\qquad\quad [startmotor(P_3,id,l) \vee stopmotor(P_3,id) \vee sendLift(P_3,id,cmd) \vee liftArrived(P_3,id) \vee \\
\qquad\quad liftReady(P_3,id) \vee tsoutput(P_3,l) \vee tsinput(P_3,l) \vee incycle(P_3,id,l) \vee \\
\qquad\quad outcycle(P_3,id,l) \vee seqcycle(P_3,id,l,l') \vee doublecycle(P_3,id,l) \vee sendCmd(P_3,id,cmd) \vee \\
\qquad\quad sendCmds(P_3,cmd,cmd') \vee cmdDone(P_3,id)]Y) \vee \\
\qquad\quad\quad \exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id)\rangle true \\
) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2,l,id,l') \vee outReady(P_2,id,l) \vee inReady(P_2,id)]Y) \\
)
\end{array}
$$

FR.5: $\forall_{id:lift\_id}[true^* \cdot (\forall_{l:location} startmotor(P_3, id, l))]\mu X.[\overline{stopmotor(P_3, id)}]X$

$\nu X \cdot ([hasInput]X \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{id':lift\_id,l:location}.[startmotor(P_3, id', l)](\alpha)$
$\vee \exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true)$
$)$

$\alpha :=$
$\mu Y \cdot ([hasInput]Y \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd,id2:lift\_id} val(id2 \neq id') \rightarrow$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id2) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$)$

FR.6-p1: $\forall_{id:lift\_id}[true^* \cdot (\forall_{l:location} doublecycle(P_3, id, l))]\mu X.[\overline{liftReady(P_3, id)}]X$

$\nu X \cdot ([hasInput]X \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{id':lift\_id,l:location}.[doublecycle(P_3, id', l)](\alpha)$
$\vee\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true)$
$)$

$\alpha :=$
$\mu Y \cdot ([hasInput]Y \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd,id2:lift\_id} val(id2 \neq id') \rightarrow$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id2) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$)$

FR.6-p2: $\forall_{id:lift\_id}[true^* \cdot (\forall_{l,l':location} seqcycle(P_3, id, l, l'))]\mu X.[\overline{liftReady(P_3, id)}]X$

$\nu X \cdot ([hasInput]X \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{id':lift\_id,l:location,l':location}.[seqcycle(P_3, id', l, l')](\alpha)$
$\vee \exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true)$
$)$

$\alpha :=$
$\mu Y \cdot ([hasInput]Y \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd,id2:lift\_id} val(id2 \neq id') \rightarrow$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id2) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$)$

FR.6-p3: $\forall_{id:lift\_id}[true^* \cdot (\forall_{l:location} incycle(P_3, id, l))]\mu X.[\overline{liftReady(P_3, id)}]X$

$\nu X \cdot ([hasInput]X \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{id':lift\_id,l:location}.[incycle(P_3, id', l)](\alpha)$
$\vee\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true)$
$)$

$\alpha :=$
$\mu Y \cdot ([hasInput]Y \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd,id2:lift\_id} val(id2 \neq id') \rightarrow$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id2) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$)$

FR.6-p4: $\forall_{id:lift\_id}[true^* \cdot (\forall_{l:location}\, outcycle(P_3, id, l))]\mu X.[\overline{liftReady(P_3, id)}]X$

$\nu X \cdot ([hasInput]X \wedge$
$((((((}$
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{id':lift\_id,l:location}.[outcycle(P_3, id', l)](\alpha)$
$\vee\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true)$
$)$

$\alpha :=$
$\mu Y \cdot ([hasInput]Y \wedge$
$((((((}$
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd,id2:lift\_id}\, val(id2 \neq id') \rightarrow$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id2) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$)$

FR.7: $\forall_{id:lift\_id}[true^* \cdot (\forall_{cmd:lift\_cmd} sendCmd(P_3, id, cmd))]\mu X.[\overline{cmdDone(P_3, id)}]X$

$\nu X \cdot ([hasInput]X \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{id':lift\_id,cmd:lift\_cmd}.[sendCmd(P_3, id', cmd)](\alpha)$
$\vee \exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true)$
$)$

$\alpha :=$
$\mu Y \cdot ([hasInput]Y \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd,id2:lift\_id} val(id2 \neq id') \rightarrow$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id2)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$)$

FR.8: $\forall_{id:lift\_id}[true^* \cdot (\forall_{cmd:lift\_cmd} sendLift(P_3, id, cmd))]\mu X.[\overline{liftArrived(P_3, id)}]X$

$\nu X \cdot ([hasInput]X \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]X \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]X) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]X)$
$\wedge$
$(\forall_{id':lift\_id,cmd:lift\_cmd}.[sendLift(P_3, id', cmd)](\alpha)$
$\vee \exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true)$
$)$

$\alpha :=$
$\mu Y \cdot ([hasInput]Y \wedge$
$((((((($
$\forall_{id:lift\_id,l:location}[step(P_5, id, l) \vee shuttleLoad(P_5, l) \vee shuttleUnload(P_5, l)]Y \vee$
$\exists_{id:lift\_id}\langle liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)\rangle true) \wedge$
$\forall_{id:lift\_id}[liftLoad(P_4, id) \vee liftUnload(P_4, id) \vee liftLoadInput(P_4, id) \vee liftUnloadOutput(P_4, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd}$
$\langle startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee$
$incycle(P_3, id, l) \vee outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee$
$sendCmd(P_3, id, cmd) \vee sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)\rangle true) \wedge$
$\forall_{id:lift\_id,l:location,l':location,cmd:lift\_cmd,cmd':lift\_cmd,id2:lift\_id} val(id2 \neq id') \rightarrow$
$[startmotor(P_3, id, l) \vee stopmotor(P_3, id) \vee sendLift(P_3, id, cmd) \vee liftArrived(P_3, id2) \vee$
$loadedInput(P_3) \vee liftReady(P_3, id) \vee tsoutput(P_3, l) \vee tsinput(P_3, l) \vee incycle(P_3, id, l) \vee$
$outcycle(P_3, id, l) \vee seqcycle(P_3, id, l, l') \vee doublecycle(P_3, id, l) \vee sendCmd(P_3, id, cmd) \vee$
$sendCmds(P_3, cmd, cmd') \vee cmdDone(P_3, id)]Y) \vee$
$\exists_{id:lift\_id,l:location,l':location}\langle transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)\rangle true$
$) \wedge \forall_{id:lift\_id,l:location,l':location}[transFree(P_2, l, id, l') \vee outReady(P_2, id, l) \vee inReady(P_2, id)]Y)$
$)$