Systems Engineering Group
Department of Mechanical Engineering
Eindhoven University of Technology
PO Box 513
5600 MB Eindhoven
The Netherlands
http://se.wtb.tue.nl/

SE Report: Nr. 2008-07

Simulation Study of Miniload-Workstation Order Picking System

R. Andriansyah, W.W.H. de Koning, R. Jordan,
L.F.P. Etman, J.E. Rooda

ISSN: 1872-1567

SE Report: Nr. 2008-07
Eindhoven, September 2008
SE Reports are available via http://se.wtb.tue.nl/sereports

Abstract

This report provides a detailed elaboration on the simulation study of a miniload-workstation order picking system, which was carried out by the Systems Engineering Group at TU/e under the FALCON project. The study is regarded as a starting point towards creating a fast, simple and accurate model for performance analysis based on simulation models.

The purpose of this simulation study is to create a detailed simulation model that represents an operating, industrial scale distribution center (DC). As a reference case, an existing DC was selected. The main characteristic of the reference case DC is the use of state-of-the-art Automated Storage/Retrieval System (AS/RS), which is becoming a common practice for large scale DC. The type of AS/RS used in this DC is referred to as the miniload-workstation order picking system, or the end-of-aisle system.

Our approach is to create a flexible and modular model architecture such that the model is not restricted to be used only for the reference case. The proposed architecture allows different system structures to be modeled by adding slight changes to the current architecture. The simulation model is built using a process algebra based simulation language χ. The proposed model is structured into three areas and four layers. Furthermore, clustered subsystems and decentralized controls are applied to the model architecture.

We validated the proposed model and performed some experiments to evaluate the performance of the DC in terms of flowtime and throughput. Furthermore, we show that different system configurations can be modeled using the proposed architecture. We conclude that our model has covered sufficient details from the reference case and hence can be used further as the base model for evaluating the performance of aggregation methods under development.

Glossary of terms

The following glossary gives a description for some terms that are used in this report.

1. SKU	:	Stock Keeping Unit, an article number that is uniquely identified.
2. Orderline	:	An orderline is the reference number under which a delivery of goods for one specific product is requested. An orderline can be split into multiple orderline-splits. An order line always refers to one SKU, but the number of individual (one) item may vary.
3. Order	:	A request to deliver specified quantities of goods or to render specific services. An order is the reference under which a delivery of goods from a distribution center is requested. An order is comprised of one or more orderlines.
4. Suborder	:	A part of an order. A suborder also consists of one or more orderlines.
5. Miniload	:	A type of Automated Storage and Retrieval System (AS/RS) that handles small items that are typically contained in small containers, totes or trays.
6. Tote	:	A (re-useable) box-shaped container used to convey a collection of items.
7. Product tote	:	A tote that contains items of an SKU, from which items are picked at the workstation.
8. Order tote	:	A tote that contains items that have been picked from several SKUs.

Contents

Chapter 1
Introduction

Contributor: W.W.H. de Koning

FALCON (Flexible Automated Logistic CONcepts) [FAL07] is a joint research project of three Dutch technical universities, the Embedded Systems Institute, and Vanderlande Industries. The FALCON project considers the design of a new generation of distribution centers and warehouses where a considerable amount of item picking functions will be automated by means of robotized systems. Falcon aims to develop methods and tools for the design of such automated distribution centers or warehouses regarding the integrated logistic system, the (robotized) hardware functionality, and the software-based control.

The contribution of the Systems Engineering group at TU/e is on model-based optimization and model-based control. Key aspect is the development of methods for aggregate model building such that aggregate models can be cast in a multi-level modeling framework for system design, optimization, and control.

In this report, a simulation model of a reference case distribution center (DC) is presented, which is needed to develop suitable aggregate models for the DC. An operating, industrial scale DC has been chosen as the reference case. The architecture of the model is such that the control and data storage is distributed over various (local) control processes, contrary to the current centralized control system architecture of the above mentioned industrial scale DC.

The main purpose of this report is to provide the readers with a detailed description of the proposed modeling technique. Throughout the chapters in this report, the readers will find how each processes is modeled using a process algebra based simulation language χ 1.0 [HR08]. The main contribution of this simulation study is developing a flexible and modular model architecture with regards to system structure and design parameters.

The remainder of this report is organized as follows. Chapter 2 elaborates the structure of the reference case DC in detail. In Chapter 3 the architecture of the simulation model is presented. In Chapters 4 and 5, detailed simulation models of the miniload system and the workstation system, respectively, are explained. Then, in Chapter 6 the collection of

real-life data is considered. In Chapter 7, a validation experiment is carried out. After that, in Chapter 8, different experiments using the simulation model presented in this report are done. Finally, conclusions are drawn and ideas for further research are presented in Chapter 9.

Chapter 2
System Description

Contributor: R. Andriansyah

In this chapter, an overview of the reference case DC is presented. We mainly focus on the physical structures and their corresponding operations.

1 Physical structure

The system structure of the reference case DC is shown in Figure 2.1. Three main areas can be distinguished, namely the *miniload, workstation,* and *conveyor*. Miniloads provide temporary storage spaces for product totes. At the workstation, items are picked from the product totes and put into the order totes. . A conveyor loop connects the miniload area to the workstation area for moving the product totes. Two other areas, namely the receiving and consolidation areas are not considered in detail.

1.1 Miniloads

Miniloads are essentially automated storage racks equipped with cranes to serve two main functions, namely the storage and retrieval of product totes. Each miniload consist of two single-deep racks with a single crane in the middle to access product totes. Each crane is capable of holding up to four product totes simultaneously. The cranes move horizontally along the aisle between the racks, while the holder of product totes move vertically to store or retrieve the totes. There are five miniloads present in the system.

1.2 Workstations

Each of the three workstations in the system consists of three input buffers and one output buffer (see figure 2.1). There are maximal three suborders active at the same time at a

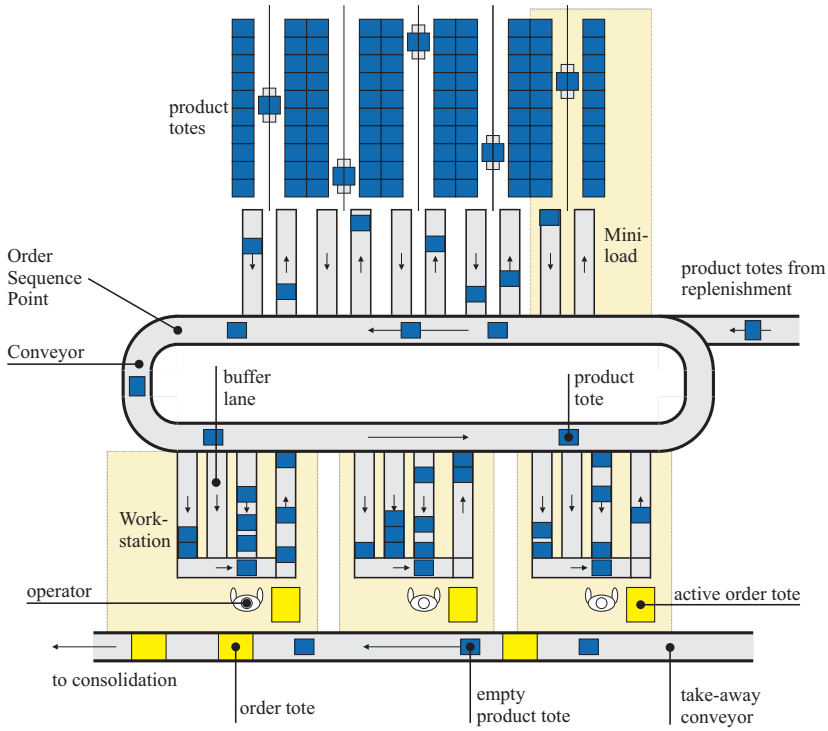


Figure 2.1: Miniload-workstation order picking system

workstation, and thus maximal nine suborders are active in the whole workstation area. An operator will work on one suborder at a time, putting all items picked from the product totes belonging to one suborder into an order tote. The operator is not allowed to start working on the next suborder when not all items for the current suborder have been picked.

1.3 Conveyor

The central conveyor loop transports product totes from the miniload area to the workstation area, and the other way around. As there are only a limited number of places for totes at the conveyor, only product totes that have successfully reserved a window are allowed to enter the conveyor. The replenishment product totes (full totes) have lower priority than the returning product totes from the workstation area (broken totes) when trying to enter the conveyor loop. This requirement is needed to ensure that there will be sufficient storage spaces at the miniload for the returning product totes.

2 Operations

The operation of the miniload-workstation order picking system is triggered by orders that enter the system at any time. An order consists of several suborders. Each suborder can contain up to 316 order lines. An order line represents an SKU type and the required amount of items for that SKU. In total, 1624 SKUs are handled in this order picking system.

2.1 Retrieval

Retrieval takes place at the miniload and starts when the miniload controller has chosen the next suborder to be completed from a list of all arriving suborders. After a suborder is selected, the inventory position of each SKU is updated. The inventory position serves as the base for the replenishment process, that is, ordering additional items from the outside suppliers. In this system, an order-up-to level replenishment policy (s, S) is used, where a number of replenishment items are ordered just enough to bring the inventory position back to a pre-determined level S, when the current inventory position has reached or dropped below a certain threshold value s.

The chosen suborder will be further divided into jobs, which specify the SKU type and the required number of items to be picked. These jobs are then assigned to the five available miniloads. As a rule, a job will be assigned to the miniload that stores the oldest tote for the SKU required by the job. A job corresponds to one or more product totes to be retrieved, since it is possible that one product tote does not contain enough items to fulfill the job. Subsequently, a list of totes to be retrieved is generated, and the retrieval action is executed if and only if there are at least four totes present in the list or a certain time has elapsed. Following a stochastic retrieval time, the retrieved totes are then put on the output buffer of the miniload, waiting to get access to the central conveyor loop to be sent to one of the workstations.

2.2 Item picking

Once a product tote has reached its destination workstation, an operator will pick the required amount of items from the tote and put the item(s) into an order tote. We assume that an order tote corresponds to one suborder, and a suborder can have more than one order tote. In the real system, however, it is possible that an order tote contains items from different suborders. When all items required for a suborder are picked into the order tote, a new order tote for the next suborder is prepared. The finished order tote is moved to the take-away conveyor.

After item picking, the operator checks whether the product totes becomes empty. If this is the case, the empty product tote will be put on the take-away conveyor along with the finished order totes to be sent to a consolidation area. Alternatively, if the product tote still contains any items left, the tote will be put on the central conveyor loop to be stored again in one of the miniloads. This product tote is referred to as a returning product tote.

2.3 Storage

The destination miniload for a returning product tote is not necessarily the same miniload from which it was retrieved. One selection criterium for the destination miniload is the miniload having available storage space with the least amount of items for the SKU type contained in the returning product tote. After the destination miniload is determined, the product totes travel to the input buffer of the destination miniload, waiting for the miniload crane to store them into the miniload racks. Similar to the retrieval, a storage action will only be executed once the number of totes waiting in the miniload input buffer reaches four or a certain time has elapsed. A stochastic storage time then applies.

Product totes for storage may also come from the replenishment process. In this case, the incoming product totes are new totes full with items. The destination miniload is determined in the same fashion as it is for the returning product totes, and the above control rule for storage also applies.

Chapter 3
Model Architecture

Contributor: W.W.H. de Koning

In the previous chapter, the system structure of the reference case DC has been explained. Next, we model the reference case DC using χ 1.0. The architecture of the simulation model is presented in this chapter.

1 Complete architecture

Figure 3.1 depicts the complete model architecture with all processes involved. Similar to the physical structure of the system, the three areas consist of a miniload, conveyor, and workstation area. Note that the conveyor in Figure 3.1 spans from `TMin` to `TWout` in the lowest level of the material flow layer. In addition to these areas, four layers of operations can be distinguished in the model, namely the *order layer, global control layer, local control layer, and material flow layer*.

1.1 Order layer

The *order* layer consists of all operations that are related to the administration of demand and supply. These operations include the creation of new customer orders by order generator `GO` and the placement of inventory replenishment orders by replenishment planner `PR`. The arriving customer orders are delivered to the miniload area by miniload planner `PM`.

1.2 Control layer

The *control* layer contains processes that record all relevant information that is used for decision-making in each area within the system. This layer is further divided into *global control* and *local control*. The main difference between the two is the scope of information that

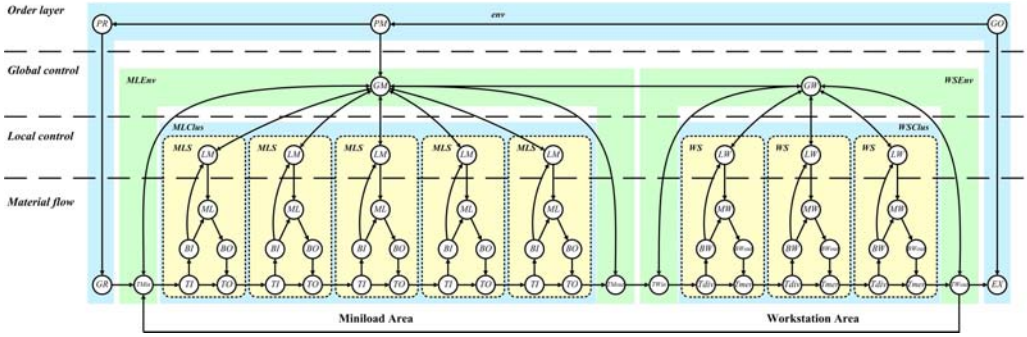


Figure 3.1: Model architecture

is accessible in each layer.

The *global controller* holds information over *all* subsystems beneath its supervision. In this model, the miniload global controller (`GM`) possess (simpified) information about all five miniload subsystems (`MLS`). Similarly, the workstation global controller (`GW`) has access to (abstracted) information of the three available workstation subsystems (`WS`).

The *local controller* contains information pertaining to the specific subsystem within its scope. A miniload local controller (`LM`), for example, has access to information *only* from the physical miniload (`ML`) under its supervision. As such, a local controller is not aware of the presence of other local controllers in the system. The same condition applies to the workstation local controllers (`LW`).

1.3 Material flow layer

The *material flow* layer represents the physical material (product totes) movement. Processes that belong to this layer includes the input and output (I/O) buffers (`BI`, `BO`, `BW`, and `BWout`), I/O conveyor windows (`TI`, `TO`, `Tdiv`, `Tmer`), and the physical miniload and workstation (`ML` and `MW`, respectively).

Conveyor windows `TI`, `TO`, `Tdiv` and `Tmer` in the model altogether form the conveyor area. Note that the conveyor area is treated differently than the other two areas. The controller for the conveyor area is actually integrated with the controller for the miniload and workstation areas. The conveyor windows require information about the destination miniload/workstation for the totes. This information, which is provided by the miniload/workstation controller, is already contained in the totes themselves. As such, there is no need to model a separate controller for the conveyor. Note that in other systems, it might be beneficial to model the controller for the conveyor area separately.

2 Simplified architecture

In Figure 3.2, an simplified overview of the simulation model is presented. In this figure, it can be seen that the model consists of the following components:

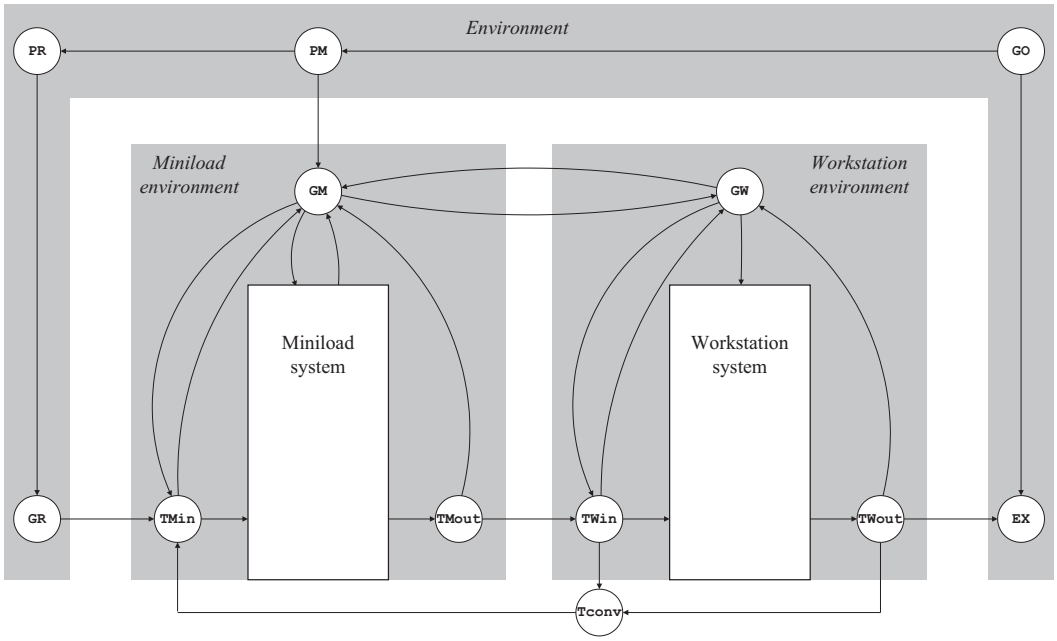


Figure 3.2: Overview of the χ model

- Total environment;
- Miniload environment;
- Workstation environment;
- Miniload system;
- Workstation system.

The total environment contains processes that take care of the generation of orders, replenishment, and consolidation of orders. The miniload (workstation) environment makes sure that the miniload (workstation) system is controlled and behaves correctly. Furthermore, the actual miniload (workstation) system is implemented in the miniload (workstation) model. The total environment, miniload environment, and workstation environment are considered in detail in this chapter. The miniload system and workstation system are explained in the following chapters.

3 Data types

The data types that are used in the χ simulation model are presented in the following lines of code.

```
type line   = ( sku:     nat      // ordered product type
              , qty:     nat      // ordered quantity
              )
   , subord = ( id:      nat      // belonging to order identity
```

```
            , seq:    nat     // sequence number
            , list:   [line]  // order line
            )
  , item   = ( sku:    nat     // product type
            , qty:    nat     // quantity
            )
  , ptote  = ( id:     nat     // tote identifier
            , timeIn: real    // starttime of tote
            , sku:    nat     // type of items
            , qty:    nat     // number of items in tote
            )
  , ttote  = ( tote:   ptote   // product tote information
            , ord:    nat     // belonging to order (99 = no info)
            , seq:    nat     // belonging to suborder (99 = no info)
            , src:    nat     // Source (99 = no info)
            , des:    nat     // Destination (99 = no info)
            , req:    nat     // Items to pick (99 = no info)
            )

  , field  = 3 * [nat]         // buffer lane fill
```

In this code, it can be seen that a suborder `subOrd` consists of identity `subOrd.id`, sequence number `subOrd.seq`, and list of products ordered `subOrd.list`. This list contains items of type `line`, which each consist of SKU number `line.sku` and quantity ordered `line.qty`. Furthermore, `field` is used for the contents of the workstation buffer lanes.

The data types used for physical elements are `item`, `pTote`, and `tTote`. As stated earlier, an item, which can be picked by an operator, does not necessarily consist of one single product. It can, for instance, represent multiple toothbrushes in one box. Therefore, `item` consist of SKU number `item.sku` and quantity `item.qty`. A product tote, which is represented by `pTote`, contains tote identifier `pTote.id`, a time stamp indicating the moment at which the product tote enters the system `pTote.timeIn`, SKU number `pTote.sku`, and number of items in the tote `pTote.qty`.

When a product tote is transported from the miniload system to the workstation system and vice versa, its information is stored in a transport tote `tTote`. The variable `tTote.pTote` indicates the product tote to be transported. Information about order and suborder to which the tote belongs is contained in the variables `tTote.ord` and `tTote.seq` respectively. `tTote.src` and `tTote.des` show the source and destination of the tote, respectively. Finally, the number of items to be picked from the tote is represented by `tTote.req`.

Constant identifiers can be found throughout the model. These identifiers are used to define the constant values repeatedly used in many parts of the model, for example the number of miniloads, workstations, batch size, etc. As we will see later on, constant identifiers simplify adjustment to the model structure. The following lines of codes define all constant identifiers.

```
const NML: nat = 5         // no. of miniloads
    , NWS: nat = 3         // no. of workstations
    , NTOTE: nat = 5503    // no. of totes after initialization
    , NCELL: nat = 6250    // no. of storage cells in a miniload
    , NWIP: nat = 100      // threshold no. of totes
    , NSKU: nat = 1624     // no. of SKUs in the system
    , MAXSUBORD: nat = 9   // no. of maximum active suborders
```

```
    , NBATCH: nat = 4     // batch size at miniload
```

As constant identifiers are not yet implemented in χ 1.0 during the time of model creation, we have used a pre-processor [APH+07] instead.

4 Total environment

As stated earlier, the environment takes care of the generation and consolidation of orders, and replenishment of the system. In Figure 3.2, it can be seen that the environment consists of order generator process GO, a miniload order process PM, replenishment process PR, tote generator GR, and exit process EX. The behavior and implementation of these processes are explained in detail in this section.

4.1 Order generator GO

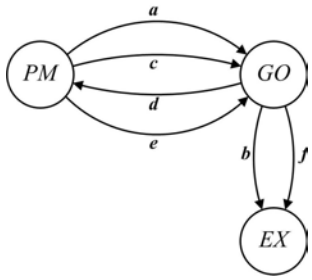


Figure 3.3: Communication in GO

Order generator GO is implemented in the following lines of code.

```
proc GO(chan a?: void, b!, c?: (nat, real), d!: subord, e?: void
           , f!: void) =
|[ var dm: -> real = uniform(0.0,1.0)
     , j,k,n: nat, p: [line], x: (nat,nat,real), xs: [(nat,nat,real)]
     , dr: -> real = uniform(0.0,1.0), r: real
     , dq: -> nat = poisson(0.22), q: nat
     , i: nat = 0, t: real
     , skuvec: [(nat,nat,real)]
:: e?; f!
 ; *( a?
      ; skuvec:= initskus()
      ; p:= []
      ; j:= numLines(sample dm)
      ; j > 0
        *> ( r:= sample dr
             ; k:= searchSKU(r,skuvec)
             ; q:= 1 + sample(dq)
             ; n:= numreq(q,k,skuvec) min 5
             ; p:= p ++ [(k,n)]
             ; skuvec:= updatesku(k,n,skuvec)
```

```
                ; j:= j - 1
              )
        ; d!(i,i,p); i:= i + 1
    )
|| *( c?(n, t); b!(n, t) )
]|
```

GO generates suborders (which consist of orderlines) based on customer requests to the DC. These suborders are generated when the initialization phase of the miniload is finished. The initialization phase is the period when the miniloads are filled with totes according to a certain replenishment policy, assuming that the miniloads are empty at the beginning. Once the initialization phase is finished, a signal is received from PM via channel e and forwarded to EX via channel f. Upon receiving this signal, the time instant for performance analysis at EX is recorded and suborders are allowed to enter the system.

Suborders are generated using a *pull* framework. That is, GO will only generate new suborders if a signal via channel a is received from PM. After initialization of variables skuvec and p, the number of orderlines j in the suborder is determined using the function numLines. For each orderline, the SKU type k and the required number of items n is defined according to the data provided (see Chapter 6). Once all orderlines in the suborder is defined, the suborder is then sent to PM via channel d.

Parallel to the above statements, GO receives information via channel c from PM and forwards this information to EX to determine the suborder flowtime.

4.2 Miniload order process PM

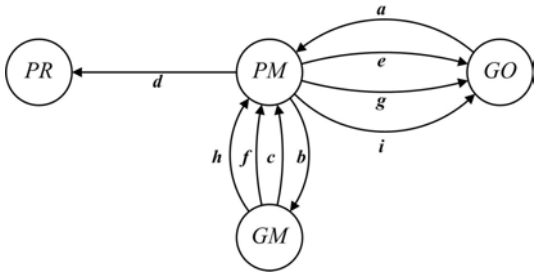


Figure 3.4: Communication in PM

Miniload order process PM is implemented in the following lines of code.

```
proc PM( chan a?, b!, c?, d!: subord, e!: (nat, real), f?, g!: void
           , h?, i!: void ) =
|[ var p, q: subord, ps: [subord] = [], qs: [nat] = []
:: *( a?p; ps:= ps ++ [p]
    | len(ps) > 0 -> b!hd(ps); ps:= tl(ps)
    | c?q; d!q; qs:= qs ++ [q.seq]
    | len(qs) > 0 -> e!(hd(qs),time); qs:= tl(qs)
    | f?; g!
    | h?; i!
    )
]|
```

In this code, it can be seen that PM receives suborders from GO over channel a, which are forwarded to the miniload environment over channel b. Furthermore, information about a suborder that is retrieved by the miniload system is received over channel c. This information is then forwarded to replenishment process PR over channel d. Channel e sends the starting time to GO for calculation of flowtime for suborder that has just been chosen by GM. Channel f receives a void signal that requests a new suborder to be released into the system. This signal will be forwarded to GO via channel g, which will create a new suborder each time the void signal is received. Similarly, channel h receives a void signal that marks the end of initialization phase and that the generation of suborders can be started. This void signal is also forwarded to GO via channel i.

4.3 Replenishment process PR

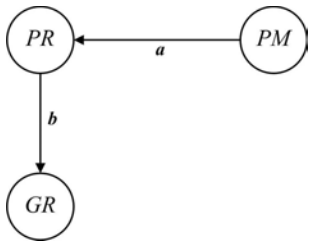


Figure 3.5: Communication in PR

Replenishment process PR decides if items of a certain SKU number should be replenished using a (*s*, *S*) ordering policy [SPP98]. Using this policy, a number of replenishment items are ordered just enough to bring the inventory position back to a pre-determined level (*S*), when the current inventory position has reached or dropped below a certain threshold value (*s*). Process PR is implemented in the following lines of code.

```
proc PR( chan a?: subord, b!: line ) =
|[ var p: subord
     , IPs: [nat], IP: %NSKU * nat
     , mns: [nat], mn: %NSKU * nat
     , mxs: [nat], mx: %NSKU * nat
     , s : line, Qty, skunr: nat
:: IP:= list2vec(IPs); IPs:= initIP()
 ; mns:= initMin(); mn:= list2vec(mns)
 ; mxs:= initMax(); mx:= list2vec(mxs)
 ; *( a?p
      ; len(p.list) > 0
        *> ( s:= hd(p.list); p.list:= tl(p.list); skunr:= s.sku
             ; IP.(skunr):= IP.(skunr) - s.qty
             ; ( IP.(skunr) <= mn.(skunr)
                 -> Qty:= mx.(skunr) - IP.(skunr)
                  ; b!(skunr,Qty)
                  ; IP.(skunr):= IP.(skunr) + Qty
               | IP.(skunr) > mn.(skunr)
                 -> skip
               )
           )
    )
]|
```

In this code, it can be seen that for each line of a suborder, which is received from channel a, the inventory position of the corresponding SKU number, IP.(s.sku) is updated. If this inventory position drops below the reorder point r.(s.sku), a replenishment order, of which the quantity is q.(s.sku), is sent to tote generator GR over channel b.

4.4 Tote generator GR

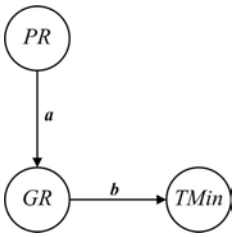


Figure 3.6: Communication in GR

Tote generator GR is implemented in the following lines of code.

```
proc GR( chan a?: line, b!: ttote, v!: void ) =
|[ var q, r: line, qs: [line] = []
     , tcs: [nat], tc: %NSKU * nat
     , i,k: nat = (0,0)
     , Qty: nat
     , IPs: [nat], IP: %NSKU * nat
:: IPs:= initIP(); IP:= list2vec(IPs)
 ; tcs:= inittote(); tc:= list2vec(tcs)
 ; k < %NSKU
   *> ( IP.k > 0
        *> ( Qty:= IP.k min tc.k
             ; b!((i,time,k,Qty),99,99,99,99,99)
             ; i:= i + 1; IP.k:= IP.k - Qty
             ; delay 10.0
           )
        ; k:= k + 1
      )
 ; *( a?q; qs:= qs ++ [q] )
|| *( len(qs) > 0 -> r:= hd(qs); qs:= tl(qs)
     ; r.qty > 0
       *> ( Qty:= r.qty min tc.(r.sku)
            ; b!((i,time,r.sku,Qty),99,99,99,99,99)
            ; v!
            ; i:= i + 1; r.qty:= r.qty - Qty
            ; delay 10.0
          )
    )
]|
```

As can be seen in this code, GR first makes sure that the system is filled. After that, GR is able to receive replenishment orders q from PR via channel a. A number of totes is then sent into the system, such that the quantity ordered is satisfied.

4.5 Exit EX

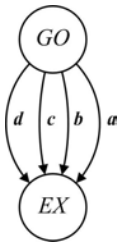


Figure 3.7: Communication in EX

Exit EX is implemented in the following lines of code.

```
proc EX( chan a?: ttote, b?: (nat, real), c?: (nat,real), d?: void
           , u!, v!: (nat, real), w!: real ) =
|[ var x: ttote
     , i: nat, j, k: nat = (0,0), t, f: real, fs: [real] = []
     , ps: [(nat, nat, real)] = []
     , mf: real = 0.0, tstart: real
     , s2phi: real = 0.0, th: real
:: *( a?x; k:= k + 1; u!(k,(time - tstart) / 3600)
    | b?(i, t); ps:= ps ++ [(i, t)]
    | c?(i, t); j:= j + 1; (f,ps):= detFlowTime(ps, i)
      ; f:= t - f; fs:= fs ++ [f]
      ; v!(j,(time - tstart) / 3600)
      ; ( j > 1  -> s2phi:= s2phi * (j - 2) / (j - 1)  + (1 / j) * (f - mf)^2
        | j <= 1 -> s2phi:= 0.0
        )
      ; mf:= mf * ( (j - 1) / j ) + f / j
      ; th:= j / (time - tstart)
      ; ( j mod 100 = 0 -> !!mf, "\t", th, "\t", s2phi, "\n"
        | j mod 100 > 0 -> skip
        )
      ; w!mf
    | d?; tstart:= time
    | j >= 10000 -> skip; delay -1.0
    )
]|
```

As can be seen in this code, EX receives empty totes from the workstation over channel a. Furthermore, EX receives information about suborders that are retrieved and suborders that are finished over channel b and channel c, respectively. Using this information, EX is able to calculate performance indicators, such as suborder throughput and suborder flow time. A signal is received via channel d that marks the end of initialization phase, after which the performance analysis of the system can be started. We determine that the simulation will be terminated once 10,000 suborders are finished.

5 Miniload environment

As stated earlier, the miniload environment makes sure that the miniload system is controlled and behaves correctly. As can be seen in Figure 3.2, the miniload environment consists of a miniload inbound transporter `TMin`, a miniload outbound transporter `TMout`, and a global miniload controller `GM`. The behavior and implementation of these processes are explained in detail in this section.

5.1 Global miniload controller `GM`

`GM` is the global controller of the miniload processes, which has as its main task to divide suborders among the miniloads and to determine in which miniload an arriving tote is stored. One of the main objectives to separate the control of the miniload system into local controllers and a global controller, is to save as much information as possible locally. Therefore, the only information saved in `GM` is the minimum amount of information needed for `GM` to successfully perform its tasks.

Because `GM` plays a central role in the system, as can be seen in Figure 3.2, its behavior is explained in detail in this section. First of all, the communication of `GM` with its surrounding processes is explained. After that, the data administration in `GM` is presented. Then, assigning an arriving tote to one of the miniloads is explained. Finally, the process of dividing suborders among the miniloads is considered.

Communication with surrounding processes

The communication of `GM` with its surrounding processes is presented in Figure 3.8, and implemented in the following lines of code.

```
proc GM( chan a!: subord                  // to PM
           , b?: subord                   // from PM
           , c!: (nat,nat)                // (qNo,mlNo)
           , d?: (line,bool)              // ((skuId,qty),prior)
           , e!: %NML # (nat,nat,nat,nat)// (skuId,qtyReq,subordId,subordSeq)
           , fIn?: %NML # (real,nat,nat) // (timeIn,skuId,toteQty)
           , fRes?: %NML # (real,nat)    // (newTimeIn,toteQty)
           , fOut?: %NML # nat           // retTotes
           , g?: void                    // leaving tote
           , h!: (subord, nat)           // (suborder, #totes)
           , i?: void                    // request
           , l!: void                    // add suborder
           , m!: void                    // start flowtime calculation
           , u!: nat                     // visualization
           , v!: %NML * nat              // visualization
           , w!: void                    // visualization
           , w2!: nat                    // visualization
           , w3!: (nat, nat)             // visualization
           , w4!: (real, real)           // visualization
       ) =
|[ var k: nat
     , p, pTemp: subord, ps: [subord] = []
     , r: nat = 0
     , totetimeIn: real, toteSkuId, toteQty, toteId: nat
```

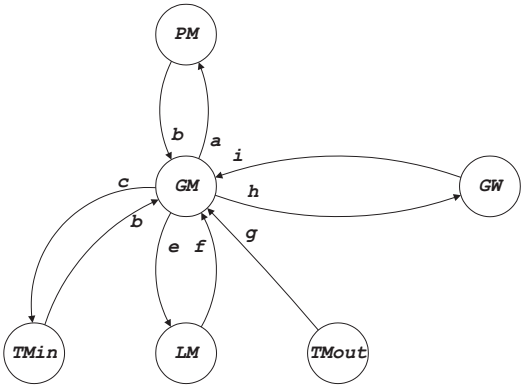


Figure 3.8: Communication of GM

```
, calc: bool = false
, zs: [(real,nat,nat)]
, z: (real,nat,nat), z1,z2: nat
, zsa: %NSKU * [(real,nat,nat)]
, x: line, xs: [line] = [], xsP: [line] = []
, ya: %NML * nat = yaInit()
, yat: nat = yatInit()
, yatin: nat = 0
, wa: %NSKU * (%NML * nat)
, MLno, retTotes, n: nat
, prior: bool
, pline: line
, timeIn: real
, MLdstr: -> nat = uniform(0,5*4*3*2*1)
, send: bool = false, Ntote, np: nat = 0
, start: bool = false
, sublen: nat = 0, msublen: real = 0.0, varsublen: real = 0.0
```

In this code, it can be seen that GM receives suborders from PM over channel b. When a suborder is retrieved, information about this suborder is sent back to PM via channel a. Process GM communicates with inbound miniload transporter TMin over channels c and d. Information about totes that must be retrieved by the miniloads are sent to the miniload system over channel e. Information is received from the miniload system when a tote is stored, retrieved, or reserved for a suborder. Therefore, channel f consists of three different channels: fIn, fOut, and fRes. Furthermore, GM receives information about departing totes from TWout via channel g. GM also sends information about a suborder that is going to be retrieved to global workstation controller GW over channel h, and receives request to retrieve a new suborder from GW over channel i. Channel l is used to send a void signal in order to request GO to create a new suborder into the system. Finally, channel m are used to mark the end of initialization phase to process PM, respectively. The initialization phase is ended when the last tote for the initial filling of the miniloads has been stored in the destination miniload.

Data administration

As stated earlier, the data stored in GM is the minimum data needed for GM to correctly perform its tasks. This data is stored in three different administrations: a physical administration, a store administration, and an order administration. These administrations, which

contain different data, are needed for different tasks, and updated after different events.

The physical administration is used to determine in which miniload an arriving tote can be stored. As a tote can only be stored in a miniload that has at least one empty position, the physical administration contains the number of totes stored in, and assigned to, each miniload. A tote is added to the physical administration when it passes `TMin`, where it is assigned to a miniload. A tote is removed from the physical administration when it is retrieved by the miniload crane. As only information about the number of totes stored in, and assigned to, each miniload is required, the data type used in the χ model is a vector `ya` containing `NML` natural numbers, where `NML` denotes the number of miniloads and initialized by `ya: NML * nat = yaInit()`.

A different physical administration is used to determine if a new tote, i.e. a tote arriving from tote generator `GR`, is allowed to enter the system. This administration, variable `yat` in the χ model, is a natural number representing the total number of totes in the system. If the value `yat` is smaller than the total miniload capacity, a new tote can be allowed to enter the system.

The store administration is used to determine in which miniload an arriving tote should be stored. In order to make sure that items are available in multiple miniloads, an arriving tote of a certain SKU should be stored in the miniload that contains the fewest items of this SKU. As a consequence, the items of a tote are added to the store administration when the tote passes `TMin`, where it is assigned to a miniload. A tote is removed from the store administration when it is reserved for a suborder, as it can then no longer be used to serve other suborders. The data type used for the store administration is an array `wa` which, for each SKU, contains a vector of `NML` natural numbers containing the number of items of the corresponding SKU in each miniload. In the χ model, array `wa` is initialized by `wa: NSKU * (NML * nat)`. Here, the constant variable `NSKU` represents the number of SKU contained in the system.

The order administration is used to determine if a suborder can be served, and, if so, which items needed for the suborder are retrieved from which miniload. Because the system contains products that are ordered rarely, i.e. slow movers, always the tote containing the oldest items of a certain SKU should be retrieved when items of this SKU are needed for a suborder. Therefore, the order administration contains data about the age of the oldest item in each miniload, and, consequently, knows in which miniload the oldest item of a certain SKU is stored. In the χ model the order administration is saved in vector of lists `zsa`, which is of the following type: `zsa: NSKU * [(real, nat, nat)]`. In this vector, for each SKU a list is saved containing tuples of type `(real, nat, nat)`. In these tuples, the first element is the age of the oldest tote in a miniload, the second element is the corresponding miniload id, and the third element is the total number of items in this miniload.

Assign an arriving tote to a miniload

As stated earlier, one of the main tasks of the global miniload controller is to assign arriving totes to one of the miniloads, which is done when a tote arrives at the inbound transporter `TMin`. This process is implemented in the following lines of code.

```
| d?(x,prior)
  ; (     prior -> xsP:= xsP ++ [x]
    | not prior -> xs:= xs ++ [x]
    )
| len(xsP) > 0
  -> x:= hd(xsP); xsP:= tl(xsP); k:= x.sku
```

```
        ; MLno:= MLassign(ya,wa.k,sample MLdstr)
        ; c!(1,MLno); ya.MLno:= ya.MLno + 1
        ; wa.k.MLno:= wa.k.MLno + x.qty
      | len(xsP) = 0 and len(xs) > 0 and yat < %NML * %NCELL
        -> x:= hd(xs); xs:= tl(xs)
        ; k:= x.sku
        ; MLno:= MLassign(ya,wa.k,sample MLdstr)
        ; c!(0,MLno); ya.MLno:= ya.MLno + 1
        ; wa.k.MLno:= wa.k.MLno + x.qty
        ; yat:= yat + 1
```

As can be seen in this code, the process of assigning an arriving tote to one of the miniloads starts with a request from `TMin` over channel `d`, which contains tote information `x` and a boolean variable `prior` that is true in case of a returning tote, i.e. a priority tote. If `prior` is true, the tote is always allowed to re-enter the system. If, on the other hand, `prior` is false, the tote can only enter the system if `yat`, i.e. the total number of tote in the system, is smaller than the total miniload capacity, as explained earlier.

The destination of an arriving tote is determined by function `MLassign`, which is implemented in the following lines of code.

```
func MLassign( val ya: %NML * nat, wa: %NML * nat, x: nat ) -> nat =
|[ var j: nat = 0
     , xs: [(nat,nat)] = []
:: j < %NML
   *> ( ( ya.j <  %NCELL -> xs:= MLList(j,wa.j,xs)
        | ya.j >= %NCELL -> skip
        )
        ; j:= j + 1
      )
 ; ret hd(drop(xs,x mod len(xs))).0
]|
```

As stated earlier, an arriving tote should be stored in the miniload that contains the fewest items of the corresponding SKU number. In the code presented above, it can be seen that `MLassign` checks in which miniloads the tote can be stored, i.e. the miniloads of which `ya`, the number of totes stored in the miniload, is smaller than the maximum capacity `NCELL`. For these miniloads, function `MLList` is used to determine which miniload contains the fewest items of the SKU number under consideration.

Divide suborders among miniloads

`GM` is responsible in assigning workloads to all miniloads, by means of dividing the suborders. In order to do this, `GM` requires three inputs. First, there must be a suborder available to be processed. New suborders are received from `PM` via channel `b`. Second, there must be a request from `GW` that indicates a workstation is able to receive a new suborder. This request is received via channel vti. Third, there should be sufficient amount of items in the miniloads to fulfill the number of required items for each SKU in the suborder. With this regard, updates regarding additional items in the miniload is essential. As such, when one or more arriving totes are stored in a miniload, `GM` receives data about the tote via channel `fIn`, after which

the order administration zsa is updated. If one of these three events happens, a boolean variable calc becomes true and GM is allowed to divide the suborders among the miniloads. The following lines of code is used to model the above behavior.

```
; ( b?p; ps:= ps ++ [p]; send:= false; calc:= true
  | ( |, j <- 0..%NML-1, fIn.j?(totetimeIn,toteSkuId,toteQty)
      ; zsa.toteSkuId:= zsUpdate(zsa.toteSkuId,j,totetimeIn,toteQty)
      ; yatin:= yatin + 1
      ; calc:= true
      ; ( Ntote < %NTOTE
            -> Ntote:= Ntote + 1
             ; ( Ntote = %NTOTE  -> start:= true; m!; w!
               | Ntote /= %NTOTE -> skip
               )
        | Ntote >= %NTOTE -> skip
        )
    )
  | i?; r:= r + 1; calc:= true
```

If calc is true, function detSubOrder determines if a new suborder can be served by checking if all items needed for a certain suborder are available in the miniloads:

```
; ( not calc -> skip
  | calc -> (calc,p):= detSubOrder(take(ps,%NWIP),r,zsa)
```

If detSubOrder returns a suborder p that can be served, GM starts to divide the items ordered among te miniloads. This process is implemented in the following lines of code.

```
          ; pTemp:= p
          ; len(pTemp.list) > 0
            *> ( pline:= hd(pTemp.list); pTemp.list:= tl(pTemp.list)
                 ; zs:= zsa.(pline.sku)
                 ; pline.qty > 0
                   *> ( z:= hd(zs); zs:= tl(zs); z1:= z.1; z2:= z.2
                        ; e.z1!(pline.sku,pline.qty,p.id,p.seq)
                        ; w3!(3,z1)
                        ; fRes.z1?(timeIn,toteQty)
                        ; w3!(6,z1)
                        ; pline.qty:= pline.qty - ( pline.qty min toteQty )
                        ; wa.(pline.sku).z1:= wa.(pline.sku).z1 - toteQty
                        ; ( z2 > toteQty -> z:= ( timeIn,z.1,z2-toteQty )
                                            ; zs:= insert(zs,z,incAge)
                          | z2 = toteQty -> skip
                          )
                        ; n:= n + 1
                      )
                 ; zsa.(pline.sku):= zs
               )
           ; h!(p, n)
           ; w3!(8,0)
           ; a!p
           ; np:= np + 1
           ; sublen:= len(p.list)
```

```
        ; ( np > 1 -> varsublen:= varsublen * (np - 2)/(np - 1)
                                + (1 / np) * (sublen - msublen)^2
          | np <= 1 -> varsublen:= 0.0
          )
        ; msublen:= msublen * (np - 1) / np + sublen / np
        ; msublen:= round(100*msublen)/100
        ; varsublen:= round(100*varsublen)/100
        ; w4!(msublen, varsublen)
    )
```

Recall that a suborder consists of a number of SKUs to be picked and the required quantity, which we refer as the suborder lines. These lines are stored in the variable `p.list`. Each of the individual suborder line (`pline`) in `p.list` is treated separately. `GM` will first look at the number of items that is ordered for a line, `pline.qty`. If the value of this variable is larger than zero, then `GM` will search for the miniload that contains the oldest item corresponding to the SKU number. This miniload is represented by `z1`. The tote containing the oldest item is then retrieved via channel `e`. Subsequently, `GM` receives updated information about the miniload via channel `fRes` and updates the store administration `wa` and order administration `zsa`. These tasks are repeated until the required number of items for the suborder line has been met. The few last lines in the above code calculates the average and variance of the suborder length, that is, the number of order lines contained in a suborder.

5.2 Miniload inbound transporter `TMin`

As can be seen in Figure 3.2, `TMin` receives both new totes from generator `GR` and returning totes from `TWout`. Process `TMin` is implemented in the following lines of code.

```
proc TMIn( chan a?: 2 # ttote, b!: ttote, c!: (line,bool)
             , d?: (nat,nat) ) =
|[ var x: ttote, xs: 2 * [ttote] = <[],[]>
    , qNo, mlNo: nat
    , toteToSend: ttote
:: *( ( |, j <- 0..1, a.j?x; xs.j:= xs.j ++ [x]
      ; c!((x.tote.sku,x.tote.qty), j = 1) )
    | d?(qNo,mlNo); toteToSend:= hd(xs.qNo); xs.qNo:= tl(xs.qNo)
      ; toteToSend.des:= mlNo; b!toteToSend
    )
]|
```

In this code, it can be seen that `TMin` receives returning (new) totes over channel `a.1` (`a.0`), after which the tote information is sent to `GM` over channel `c`. Furthermore, if a tote can be sent to the miniload system, information about the destination is received from `GM` over channel `d`, after which the correct tote is sent.

5.3 Miniload outbound transporter `TMout`

Outbound transporter `TMout`, is implemented in the following lines of code.

```
proc TMOut( chan a?, b!: ttote, c!: void ) =
|[ var x: ttote
:: *( a?x
```

```
      ; ( x.tote.qty = x.req -> c!
        | x.tote.qty > x.req -> skip
        )
      ; b!x
    )
]|
```

As can be seen in this code, `TMout` receives totes from the miniload system over channel `a`. As the number of items in the tote and the number of items to pick are known, `TMout` can determine if the tote will become empty after item picking. If so, a notification is sent to `GM` over channel `c`, after which `GM` updates its physical administration `yat`. Furthermore, the tote is forwarded to workstation inbound transporter `TWin` via channel `b`.

6 Workstation environment

As stated earlier, the workstation environment makes sure that the workstation system is controlled and behaves correctly. As can be seen in Figure 3.2, the workstation environment consists of a workstation inbound transporter `TWin`, a global workstation controller `GW`, and a workstation outbound transporter `TWout`. The behavior of these processes is explained in detail in this section.

6.1 Global workstation controller `GW`

GW is the global controller of the workstation system, which has as its main tasks to divide suborders and arriving totes among the workstations, and to send a request to global miniload controller `GM` when a new suborder can be served.

Communication with surrounding processes

The communication of `GW` with its surrounding processes is presented in Figure 3.9, and implemented in the following lines of code.

```
proc GW( chan a!: void
           , b?: (subord, nat)
           , c!: (nat, nat, bool)
           , d?: ttote
           , e!: %NWS # (subord, nat)
           , f?: (nat, bool)
           , v!: nat
           , val maxTotes: nat
       ) =
|[ var x: ttote
     , p: subord
     , ps: [(subord, nat)]
     , k: nat, n: nat = 0, m: nat = 0, wip: nat = 0, i: nat = 0
     , ns: %NWS * nat = initWS()
     , maxSubord: nat = 9
     , request: bool = true
```

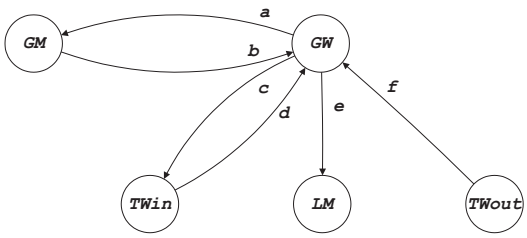


Figure 3.9: Communication of GW

```
, wsId, des, sortId, j: nat
, flag, new: bool
, occ: %NWS * nat = initWS()
, ts: %NWS * [(subord, nat)] = inittsNWS()
```

In this code and Figure 3.9, it can be seen that GW sends GM requests for new suborders over channel a, and that information about these suborders is received over channel b. To determine to which workstation an arriving tote must be send, GW communicates with TWin over channels c and d. If a suborder is assigned to a certain workstation, GW sends information about this suborder to the workstation system via channel e. TWout notifies GW when a tote leaves the workstation system via channel f. Finally, channel v is used only for visualization purposes.

An important property of the workstation system is that totes belonging to at most three different suborders can be arriving at a workstation simultaneously, due to the fact that each workstation has three inbound buffer lanes. If totes belonging to more different suborders arrive simultaneously, a workstation cannot store them in such a way that the operator is able to handle totes belonging to a suborder subsequently. Consequently, a new suborder can be allowed to arrive at a workstation when all totes belonging to one of the previous suborders have arrived.

Process GW is implemented in the following lines of code.

```
:: *( wip < maxSubord and n < maxTotes and request
      -> a!; wip:= wip + 1; request:= false
    | b?(p, k); ps:= ps ++ [(p, k)]; n:= n + k; request:= true
    | d?x; m:= m + 1; v!m
      ; (new, k, p, ps):= newSubOrder(x, ps)
      ; (     new -> i:= i + 1
                     ; wsId:= detWSid(ns,occ)
                     ; ns.wsId:= ns.wsId + k
                     ; occ.wsId:= occ.wsId + 1
                     ; e.wsId!(p, i)
                     ; ts.wsId:= ts.wsId ++ [(p, i)]
        | not new -> skip
        )
      ; (ts, des, sortId, flag):= arrtote(ts, x)
      ; (     flag -> occ.des:= occ.des - 1
        | not flag -> skip
        )
      ; c!(des, sortId, flag)
    | f?(j, flag); n:= n - 1; m:= m - 1; v!m; ns.j:= ns.j - 1
      ; (     flag -> wip:= wip - 1
```

```
        | not flag -> skip
        )
    )
||
```

In order to correctly send requests for new suborders, variables `maxSubord` and `maxTotes` are used. Variable `maxSubord` is used to define the maximum number of suborders in and due to arrive at the workstation system. Variable `maxTotes` is used to define the maximum number of totes in and due to arrive at the workstation system. If both the number of suborders, `wip`, and the number of totes, `n`, in and due to arrive at the workstation is less than these maximum values, a request for a new suborder can be sent.

When information about an arriving tote is received from `TWin` over channel `d`, function `newSubOrder` determines if this tote is the first tote of a certain suborder. If so, function `detWSid` determines to which workstation the corresponding suborder is allocated. After this, function `arrtote` determines a sorting identity `sortId`, which is needed at the workstation system, and a boolean variable `flag`, which is true if the tote is the last tote of a suborder. This information is sent back to `TWin` over channel `c`. Subsequently, `GW` checks whether the incoming tote is the last tote of a suborder (where, `flag = true`). If this is the case, then the number of suborders occupying a workstation (`occ`) can be subtracted by one.

`GW` may also receive a signal via channel `f` that a tote has left the workstation area. If the tote is the last tote of a suborder, then the number of active suborder (`wip`) is subtracted by one. This indicates there is a room for one suborder in the workstation area.

6.2 Workstation inbound transporter `TWin`

As can be seen in Figure 3.2, `TWin` receives totes arriving from the miniload area. Process `TWin` is implemented in the following lines of code.

```
proc TWIn( chan a?: ttote
             , c!: (ttote,nat,bool)
             , d!: ttote
             , e?: (nat,nat,bool)
             , var dt: real
         ) =
|[ var ys: [((ttote, nat, bool), real)] = []
     , x: ttote
     , flag: bool
     , wsid, sortId: nat
:: *( a?x; d!x; e?(wsid, sortId, flag); x.des:= wsid
      ; ys:= ys ++ [((x, sortId, flag),time+dt)]
    )
|| *( len(ys) > 0
      -> skip; delay ( hd(ys).1 - time ) max 0.0; c!hd(ys).0; ys:= tl(ys)
    )
||
```

In this code, it can be seen that totes are received over channel `a`. After that, its information is sent to `GW` over channel `d`, and information about its destination is received over channel `e`. Finally, totes are sent to the workstation system over channel `c`.

6.3 Workstation outbound transporter TWout

Workstation outbound transporter TWout is implemented in the following lines of code.

```
proc TWOut( chan a?: (ttote, nat, bool), b!, c!: ttote, d!: (nat, bool)
                , e!: (nat,real) ) =
|[ var x: ttote, sortId: nat, flag: bool
:: *( a?(x, sortId, flag)
      ; ( x.tote.qty > 0 -> b!x
        | x.tote.qty = 0 -> c!x
        )
      ; d!(x.src, flag)
      ; (     flag -> e!( x.seq, time-(2-x.src)*2.0 )
        | not flag -> skip
        )
    )
]|
```

As can be seen in this code, totes are received from the workstation system over channel a. If the tote is nonempty, it is sent back to the miniload area over channel b. If, on the other hand, the tote is empty, it is sent to exit EX over channel c. Furthermore, if the tote is the last tote of a suborder, i.e. flag is true, information is sent to EX over channel e that a suborder has been finished.

7 Verification

In this chapter, it is verified that the simulation model behaves correctly. For this purpose, a scaled down model, containing only 10 different SKU numbers, is considered.

7.1 Filling the system

As stated earlier, replenishment generator GR makes sure that the system is filled with the right number of totes. As explained earlier, GR sends totes into the system until the number of items in the system is equal to the initial inventory position, which in this case is defined by the following function initIP.

```
func initIP() -> 10 * nat =
|[ ret <500,500,500,500,500,500,500,500,500,500> ]|
```

The number of items per tote for each SKU number is defined by the following function initS.

```
func initS() -> 10 * nat =
|[ ret <10,10,10,10,10,10,10,10,10,10> ]|
```

Consequently, after filling the system, it should contain 50 totes per SKU number, each containing 10 items.

Using the model presented in the previous chapters, the following data is obtained after filling the system.

```
sum zsa: <500,500,500,500,500,500,500,500,500,500>
ya: <100,100,100,100,100>
yat: 500
```

As explained earlier, `zsa` contains a list of totes present in each miniload, for each SKU. Variable `sum zsa` in the data presented above is, for each SKU number, the sum of the items over all totes in all miniloads. It can be seen that this is equal to the initial inventory position defined by function `initIP`. Furthermore, from `ya` and `yat` it can be concluded that each miniload contains 100 totes. Consequently, it can be concluded that the process of filling the system is implemented correctly.

8 Processing a suborder

To check if the model behaves correctly when processing a suborder, the following suborder is sent into the system.

```
p = ( 0, 0, [(0,3),(1,4),(2,3),(3,4),(4,3),(5,4),(6,3),(7,4),(8,3),(9,4)] )
```

Doing this, the following output is obtained.

```
GM      new suborder is processed, suborder number: 0

TWIn    receive tote sku id 0, source 4, dest 0, qty: 10, req: 3
TWIn    receive tote sku id 1, source 3, dest 0, qty: 10, req: 4
TWIn    receive tote sku id 5, source 2, dest 0, qty: 10, req: 4
TWIn    receive tote sku id 8, source 4, dest 0, qty: 10, req: 3
TWIn    receive tote sku id 3, source 1, dest 0, qty: 10, req: 4
TWIn    receive tote sku id 2, source 3, dest 0, qty: 10, req: 3
TWIn    receive tote sku id 4, source 0, dest 0, qty: 10, req: 3
TWIn    receive tote sku id 7, source 2, dest 0, qty: 10, req: 4
TWIn    receive tote sku id 9, source 3, dest 0, qty: 10, req: 4
TWIn    receive tote sku id 6, source 1, dest 0, qty: 10, req: 3

TMIn    receive tote, source: 0, sku no. 0, qty: 7
TMIn    receive tote, source: 0, sku no. 1, qty: 6
TMIn    receive tote, source: 0, sku no. 5, qty: 6
TMIn    receive tote, source: 0, sku no. 8, qty: 7
TMIn    receive tote, source: 0, sku no. 3, qty: 6
TMIn    receive tote, source: 0, sku no. 2, qty: 7
TMIn    receive tote, source: 0, sku no. 4, qty: 7
TMIn    receive tote, source: 0, sku no. 7, qty: 6
TMIn    receive tote, source: 0, sku no. 9, qty: 6
TMIn    receive tote, source: 0, sku no. 6, qty: 7
```

```
sum zsa: <497,496,497,496,497,496,497,496,497,496>
ya: <100,100,100,100,100>
yat: 500
```

In this output, it can be seen that workstation inbound transporter `TWin` receives totes needed for the suborder from different miniloads, and that the suborder is assigned to workstation o. Furthermore, it can be seen that required quantity for each tote corresponds with the number of items ordered. After the totes are served by the workstation, the are sent to miniload inbound transporter `TMin`. In the output, it can be seen that the right amount of items has been picked from the totes. Furthermore, it can be seen that the number of items in the system is updated correctly in `sum zsa`. Consequently, it can be concluded that the system behaves correctly when processing a suborder.

In this section, the architecture of the simulation model has been explained. In the next sections, the miniload system and the workstation system are discussed in detail.

Chapter 4
Miniload Model

Contributor: R.Andriansyah

The miniload area is one of the most important aspects of the end-of-aisle AS/RS system. It represents the temporary storage spaces from which totes are retrieved to fulfill a (part of) suborder and to which the totes are returned after item picking at the workstation area. There are five miniloads present in the system and each miniload is modeled using several processes. A complete miniload system is referred as `MLS`, which is depicted as follows.

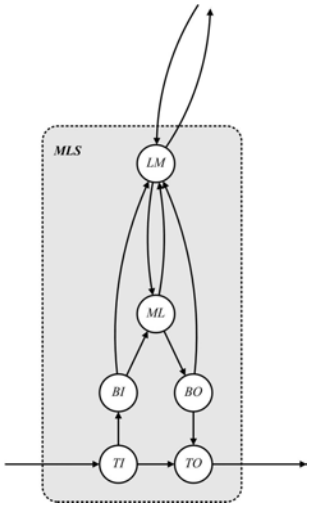


Figure 4.1: Miniload system

A miniload system `MLS` contains local miniload controller `LM`, physical miniload `ML`, input buffer `BI`, output buffer `BO`, input conveyor `TI` and output conveyor `TO`. We will explain in detail each of the component of the miniload system in the following sections.

1 Local miniload controller LM

The local miniload controller LM is responsible for making decisions based on the information contained in one particular miniload. There are in total five independent local miniload controllers, each of which communicates with the global miniload controller (GM)). In this section, we will describe each processes in LM separately to cover sufficient details in the processes.

1.1 Communication with surrounding processes

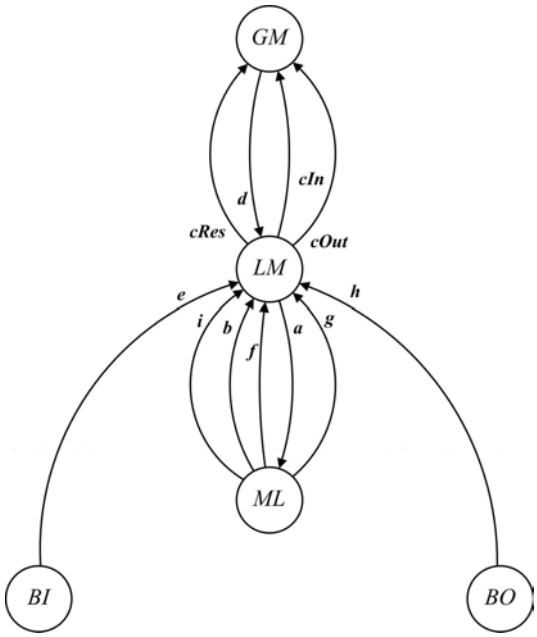


Figure 4.2: Communication in LM

The communication that takes place in LM is depicted in the following lines of code.

```
proc LM (chan a!: (bool,[ttote])       // send job assignment to ML
            , b?: [ttote]              // stored totes from ML
            , cIn!: (real,nat,nat)     // update stored totes to GM
            , cRes!: (real,nat)        // update unreserved totes to GM
            , cOut!: nat               // update retrieved totes to GM
            , d?: (nat,nat,nat,nat)    // retrieve job assignment from GM
            , e?: ttote                // update totes in BI
            , f?: bool                 // update crane position
            , g?: nat                  // update no. of totes in ML
            , h?: void                 // update no. of totes in BO
            , i?: void                 // receive trigger for time out at ML
            , v!: 3*[ttote]            // visualization
            , val k: nat               // miniload index
        ) =
|[ var skuV: %NSKU * [ptote]
         , x: ptote, ys: [ttote] = [], zs: [ttote] = [], z: ttote
```

```
, sVec: %NSKU * nat, ms: [bool], m: nat, rs: [ttote], r: ttote
, go: bool = false, calc: bool, crIn: bool = false
, inputlist: [(real,nat,nat)] = [], outputlist: nat = 0
, nBO: nat = 0, B: nat, ytake: nat, o: nat, os: [nat] = []
, ya: 3*[ttote] = <[],[],[]>, y: ttote
, row: nat = 0, skuNo, pId, pSeq, qtyReq: nat
```

In general, LM communicates with three other processes namely GM, ML, BI, and BO. Communication with GM concerns the updating of totes either after storage, retrieval, or reservation of totes (via channel cIn, cOut, and cRes, respectively). This update is essential for GM, since GM assigns retrieval jobs to the miniloads based on the information about tote in each of the miniload. LM receives the new job assignment from GM via channel d. This job will then be forwarded to the physical miniload ML via channel a. After a storage or retrieval action has been carried out by ML, then ML will send a signal to LM in order to update some parameter values that is important for decision making by LM. The update includes the crane position at ML, totes in the input and output buffer (BI and BO). The channels that are involved in the updates are channels b, e, f, g, and h. Furthermore, LM may occasionally receive a time out signal from ML via channel i. More about this time out will be explained in section 2.

1.2 Initialization

During the initialization phase, two variables are created, namely the available product totes per SKU in the miniload (variable skuV) and the total number of items per SKU in the miniload (variable sVec). The initialization is modeled as follows.

```
:: skuV:= initskuV()
 ; sVec:= initNatVec()
```

Since the simulation is started with empty miniloads, these two variables are empty as well. The following functions are used to initialize the two variables.

```
func initskuV() -> %NSKU * [ptote] =
|[ var vec: %NSKU * [ptote], i: nat = 0
:: i < %NSKU
   *> ( vec.i:= []; i:= i + 1 )
 ; ret vec
]|

func initNatVec() -> %NSKU * nat =
|[ var vec: %NSKU * nat, i: nat = 0
:: i < %NSKU *> ( vec.i:= 0; i:= i + 1 )
 ; ret vec
]|
```

1.3 Receiving new assignments

```
 ; *( calc:= true
      ; ( d?(skuNo,qtyReq,pId,pSeq)
          ; x:= hd(skuV.skuNo); skuV.skuNo:= tl(skuV.skuNo)
          ; y:= (x, pId, pSeq, k, 99, x.qty min qtyReq)
          ; ya:= arrangeSubOrd(ya, y)
```

```
; v!ya
; sVec.skuNo:= sVec.skuNo - x.qty
; ( len(skuV.skuNo) > 0 -> cRes!(hd(skuV.skuNo).timeIn, x.qty)
  | len(skuV.skuNo) = 0 -> cRes!(0.0, x.qty)
  )
```

LM receives new assignments from GM via channel d. These assignments are essentially orderlines that form a suborder. Information that is provided by GM includes the SKU type (skuNo), required quantity (qtyReq), suborder ID (pID), and orderline sequence number (pSeq). Based on these information, LM will decide which product totes should be use to fulfill this assignment. LM will always select the oldest product tote with the required SKU type in the miniload. Next, the selected tote is said to be *reserved* for this particular assignment. This means the selected tote cannot be used for other assignments. Since the tote is now reserved, the quantity of items for the particular SKU in variable sVec is updated by subtracting the original amount by the total number of items in the selected tote. An update is then sent to GM via channel cRes to inform the miniload status after the tote reservation for an assignment. Note that at this point no storage or retrieval has been carried out yet.

A boolean variable called calc is introduced in the statement. The initial value of this variable is true, which implies that LM should evaluate whether a storage or retrieval should be performed. If in the subsequent statements the value of variable calc is changed to false, then there is no need for LM to perform the evaluation.

1.4 Receiving update of totes to be stored

```
| e?z; zs:= zs ++ [z]
```

Updating the totes waiting to be stored is crucial since it influences the decision whether to send storage or retrieval signal to ML. This is done via channel e as showed above, where z is the arriving tote to be stored in the miniload. LM will subsequently put the arriving tote into a list zs.

1.5 Receiving update of retrieved totes

```
| g?m
  ; outputlist:= outputlist + m
  ; calc:= false
| outputlist > 0
  -> cOut!outputlist
   ; outputlist:= 0
   ; calc:= false
```

When a tote is already retrieved at ML, then ML will inform LM about the number of totes that has been retrieved. This information is represented by the variable outputlist, which will subsequently be forwarded to GM. Eventually, GM will use the updated information for the tote allocation to the miniloads. Note that since this statement does not serve as the basis for deciding whether to perform storage or retrieval, the variable calc is set to false.

1.6 Receiving update of stored totes

```
| b?rs
  ; len(rs) > 0
    *> ( r:= hd(rs); rs:= tl(rs)
         ; skuV.(r.tote.sku)
           := insert(skuV.(r.tote.sku), (r.tote), pred)
         ; sVec.(r.tote.sku):= sVec.(r.tote.sku) + r.tote.qty
         ; inputlist:= inputlist
           ++ [(hd(skuV.(r.tote.sku)).timeIn, r.tote.sku
               , r.tote.qty)]
       )
  ; calc:= false
| len(inputlist) > 0
  -> cIn!hd(inputlist); inputlist:= tl(inputlist); calc:= false
```

In the same way with retrieval, when a tote is already stored into the miniload, ML will inform LM about the totes that have been stored. Subsequently, LM will update its list of available totes. The information about available totes (skuV) are then sorted according to the age of the totes, where the oldest totes for an SKU will be put early in the list. The number of items for the SKU of the tote that has just been stored is also updated in the variable sVec. Finally, LM will forward the updated information about the oldest tote of a particular SKU in the miniload after the storage operation has been executed.

1.7 Receiving update of crane position and number of totes at the output buffer

```
| f?crIn
| h?; nBO:= nBO - 1
```

After ML executes a storage or retrieval of totes, the crane position at ML will change. Since the crane position is also one of the crucial factors for LM to decide whether to send a storage or retrieval signal, an update of the crane position is clearly mandatory. This update is done by receiving a boolean variable crIn via channel f. More discussion on the crane position will be provided next.

Another important update concerns the number of totes at the output buffer. Each time a retrieved tote leaves the output buffer to be put on the output conveyor TO, a signal is sent by BO to LM. This information is taken into account when deciding whether to do retrieval or not. Since the output buffer BO has a limited capacity of 12 totes, retrieval of totes can only be executed when there is sufficient space at the output buffer BO for the retrieved totes.

1.8 Storage/retrieval decision

```
| go and crIn
  -> a!(crIn, sendlist(ya, (B - nBO), row))
   ; (ya, ytake, row):= updateYA(ya, (B - nBO), row)
   ; v!ya
   ; nBO:= nBO + ytake
   ; go:= false
| go and not crIn
  -> a!(crIn,take(zs,%NBATCH)); zs:= drop(zs,%NBATCH); go:= false
)
```

Five variables serve as the basis for storage/retrieval decision, namely `go`, `crIn`, `nBO`, `ya`, and `zs`. The function `sendlist` is used to choose the next totes to be retrieved, while the function `updateYA` updates the list of available totes to retrieve. The variable `NBATCH` is a constant variable whose value can be easily determined using a constant identifier. This variable represents the batch size for storage/retrieval totes.

The boolean variable `go` decides whether a storage or retrieval action can be executed. One of the two statements will only be executed if `go` returns true. The value of `go` is determined by the function `inOut` that is evaluated each time the variable `calc` returns true. This process is modeled as follows.

```
; (     calc -> go:= inOut(len(zs),(len(ya.0)+len(ya.1)+len(ya.2)))
  | not calc -> skip
  )
```

Furthermore, the following specification represents the function `inOut`. This function will return `true` if the number of totes to be stored/retrieved has reached `NBATCH` totes.

```
func inOut(val x: nat, y: nat) -> bool =
|[ ( ret x >= %NBATCH or y >= %NBATCH ) ]|
```

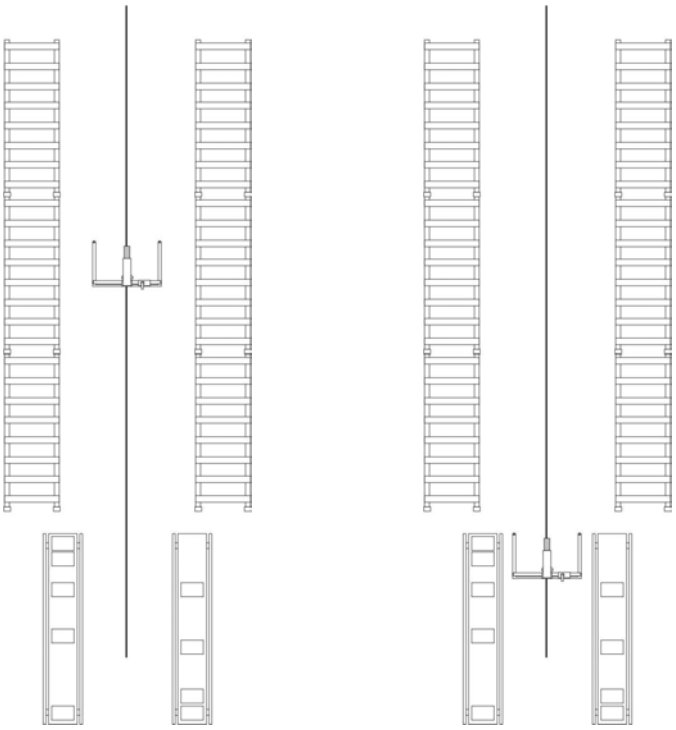
The totes to be stored are physically present in the *storage queue* (q_s) of the miniload input buffer. The totes to be retrieved, on the contrary, are contained in a virtual *retrieval queue* (q_r) that is available in the miniload controller. In principle, a storage or retrieval can take place if a batch of four totes has been formed in either queues or if the miniload crane has waited (Δ) for more than 120 seconds ($q_s \geq 4 \vee q_r \geq 4 \vee \Delta \geq 120$). In this case, we set the value `NBATCH` to 4.

The decision whether a storage or retrieval will be executed depends on the value of the variable `crIn`. If *crIn* is true, then the crane is currently *inside* the miniload, ready to *retrieve* totes. On the contrary, if *crIn* is false, then the crane is currently *outside* the miniload, ready to *store* totes.

In the case where *go* is true ($q_r \geq 4 \vee q_s \geq 4$) and the crane is *inside* the miniload (*crIn* is true), then *LM* will send four totes to be retrieved by the miniload *only if* there is sufficient space in the miniload output buffer. If the remaining space in the miniload output buffer is less than four totes, then `LM` will only retrieve a number of totes that still can be put in the output buffer (that is, $B - nBO$). In other words, the miniload will send the minimum of *four totes* and *remaining space at the output buffer*. The minimum of these two variables and the total available assignments in `LM` is defined as the variable `ytake`. On the contrary, if `go` is true and the crane is *outside* the miniload (`crIn` is false), then *LM* will send four totes to be stored by the miniload.

It is also possible that while the crane is *inside* the miniload, the number of totes to be *stored* has formed a batch of four, thus returning a true for `go`. In this case, because the crane is not at the right position to immediately store the batch of four totes, the crane will first travel to outside the miniload by taking the all the available totes (four totes or less) even if the number of totes has not reached four. This action is captured using the command `take`. The same action will also take place when the crane is *outside* the miniload but the number of totes to be *retrieved* has reached four totes.

An overview of miniload storage/retrieval operation is provided in Figure 4.4.



(a) Crane inside miniload (b) Crane outside miniload

Figure 4.3: Positions of miniload crane

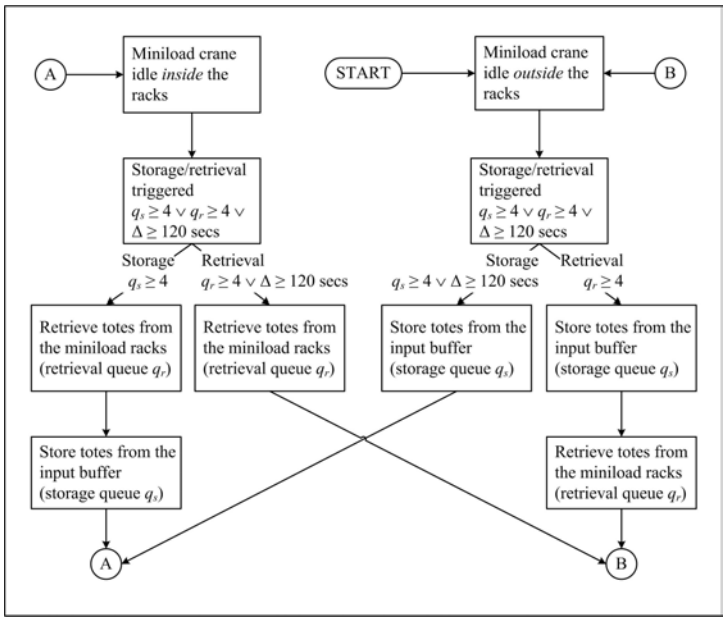


Figure 4.4: Miniload operation

2 Physical miniload ML

The physical miniload ML does the physical storage and retrieval of totes, which is triggered by LM.

2.1 Communication with surrounding processes

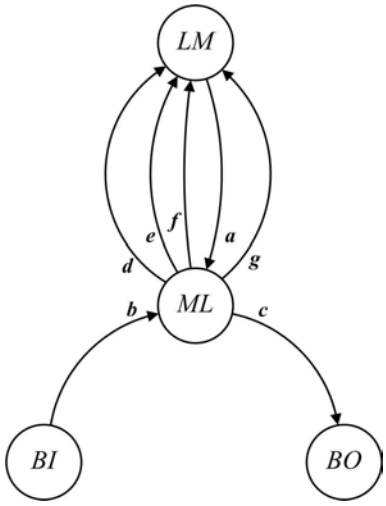


Figure 4.5: Communication in ML

```
proc ML (chan a?: (bool,[ttote]), b?: [ttote], c!: [ttote], d!: bool
           , e!: nat, f!: [ttote], g!: void, v!: bool
           , val k: nat) =
|[ var ps: [ttote], p: ttote, xs: [ttote]
       , crIn: bool = false, t1: real, t2: -> real = constant(120.0)
       , retrieve: bool, timeout: real = 120.0
```

ML communicates with three other components in the miniload system MLS, namely the local controller LM, the input buffer BI, and the output buffer BO. Via channel a, a signal is received containing the action that must be done by the physical miniload (retrieve) and the list of totes to be retrieved (ps). When ML is signaled to do a storage, then it will receive the list of totes to be stored via channel b. Retrieved totes are sent to the output buffer BO via channel c. Once the storage or retrieval is done, an update of the crane position is sent to the local controller LM via channel d. Updates regarding the status of totes after retrieval or storage is done via channels e and f, respectively. In the case where timeout has occurred, then ML will send a signal via channel g to trigger a storage or retrieval action, even if the required batch size of 4 totes has not been reached yet. More on this time out behavior will be explained next. Finally, channel v is used for visualization purposes.

2.2 Storage/retrieval action

```
:: *( a?(retrieve,ps)
    ; v!false
    ; ( retrieve
```

```
      -> skip
       ; ( len(ps) = 0 -> t1:= 23.6
         | len(ps) > 0 -> t1:= 16.5 + ((len(ps) - 1) * 7.0 )
         )
       ; delay t1
       ; crIn:= false; d!crIn
       ; c!ps
       ; e!len(ps)
       ; timeout:= time + sample t2
    | not retrieve
      -> b?xs
       ; ( len(xs) = 0 -> t1:= 23.6
         | len(xs) = 1 or len(xs) = 2 -> t1:= 30.6
         | len(xs) = 3 or len(xs) = 4 -> t1:= 37.5
         )
       ; delay t1
       ; crIn:= true; d!crIn
       ; f!xs
       ; timeout:= time + sample t2
    )
  ; v!true
```

The storage/retrieval action is started with receiving a signal with action to be accomplished, either storage or retrieval, as reflected by the variable `retrieve`. If this variable returns true, then a retrieval will be carried out. In this case, a list of totes to be retrieved is already provided by the variable `ps`. The delay that occurs due to the retrieval action depends on the number of totes to be retrieved. This is based on the real data from the reference case DC as will be discussed next in chapter 6. Subsequently the position of the crane will be updated by setting the value of variable `crIn` to false. This implies the crane position after retrieval is outside the miniload (see Figure 4.3(b)).

On the contrary, if the variable `retrieve` returns false, then a storage will be executed. `ML` will allow the input buffer `BI` to send the list of totes to be stored, which is done via channel `b`. After a certain delay, the storage operation is finished and the boolean variable `crIn` that represents the crane position is updated to true. This is due to the fact that the crane will be located inside the miniload following a storage operation. Note that the delay due to the storage operation is also dependant upon the number of totes involved, as in the case of retrieval operation. The parameter value for the delay is provided in chapter 6.

2.3 Time out behavior

```
| delay ((timeout - time) max 0.0)
  ; g!
  ; a?(retrieve,ps)
  ; v!false
  ; ( retrieve
      -> skip
       ; ( len(ps) = 0 -> t1:= 23.6
         | len(ps) > 0 -> t1:= 16.5 + ((len(ps) - 1) * 7.0 )
         )
       ; delay t1
       ; crIn:= false; d!crIn
       ; c!ps
       ; e!len(ps)
```

```
      ; timeout:= time + sample t2
    | not retrieve
      -> b?xs
      ; ( len(xs) = 0 -> t1:= 23.6
        | len(xs) = 1 or len(xs) = 2 -> t1:= 30.6
        | len(xs) = 3 or len(xs) = 4 -> t1:= 37.5
        )
      ; delay t1
      ; crIn:= true; d!crIn
      ; f!xs
      ; timeout:= time + sample t2
  )
; v!true
```

In the real system under study, the miniload crane will wait until a batch of four totes has been formed for storage or retrieval action. If after a certain delay this batch of four totes is not yet formed, then a *time out* the miniload crane will take all totes that is currently available and perform the storage or retrieval action, depending on the current position of the miniload crane as explained in previously. In the above code, the miniload will wait for a delay of `(timeout - time)`, where the variable `timeout` is set to 120 seconds after the previous storage/retrieval action.

From the modeling perspective, a void signal will be sent to `LM` via channel `g` when a time out is triggered. `LM` will receive this signal and overrules the function `inOut` that requires a batch of 4 totes to be formed before a storage or retrieval can be performed. `ML` will then receive a signal from `LM` about the list totes to store or retrieve, which is less than four totes. The rest of the statements are similar to the normal storage/retrieval action as is described in the previous section.

3 Miniload input transporter `TI`

The miniload input transporter `TI` is responsible for sending totes to the correct miniload for storage.

3.1 Communication with surrounding processes

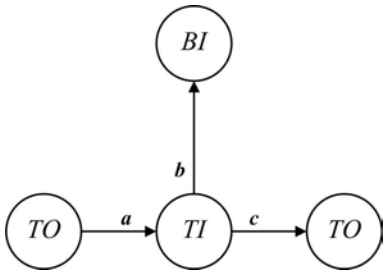


Figure 4.6: Communication in `TI`

```
proc TI (chan a?: ttote, b!: ttote, c!: ttote, val k: nat) =
```

```
|[ var x: ttote, xs: [(ttote,real)] = [], t: real
```

BI communicates with two processes, namely the input buffer BI and the output transporter TO (refer to Figure 4.1). BI receives totes from the previous TO via channel a. After a fixed delay of 1.0 second, the destination of the totes is determined. If the tote is destined for the current miniload, then the tote will be sent to the input buffer BI of the current miniload via the channel b. However, if the destination miniload of the tote is not the current tote, then TI will send the tote to the output transporter TO via channel c. The value k indicates the current miniload index.

3.2 Receiving a tote

```
:: *( a?x
      ; t:= time + 1.0; xs:= xs ++ [(x,t)]
      ; len(xs) > 0
        *> ( a?x
             ; t:= time + 1.0; xs:= xs ++ [(x,t)]
           | delay ((hd(xs).1 - time) max 0.0)
             ; ( hd(xs).0.des  = k -> b!hd(xs).0
               | hd(xs).0.des /= k -> c!hd(xs).0
               )
             ; xs:= tl(xs)
           )
```

The first statement in TI concerns the receiving of tote from the previous transporter. Afterwards, a fixed delay of 1.0 second is applied, after which the tote is ready to be sent to one of the two available destinations: either the input buffer BI or the output transporter TO of the current miniload. In a way, one can see TI and TO as separate conveyor windows: if the destination miniload is not the current miniload, then the tote is transferred to the next conveyor window until the destination miniload is reached. Only after the destination miniload for the tote is reached will the tote be put on the input buffer BI.

4 Miniload input buffer BI

The miniload input buffer BI holds the transport totes that are waiting to be stored into the miniload. In this model, we assume an infinite capacity of BI.

4.1 Communication with surrounding processes

```
proc BI (chan a?: ttote, b!: [ttote], c!: ttote, val k: nat) =
|[ var xs: [ttote], x: ttote
```

BI receives totes from the input transporter TI via channel a. Channel b is used to send totes to ML for storage. Channel c is used to send an update about the current totes that is available at BI. Finally, the value k is used to note the miniload index.

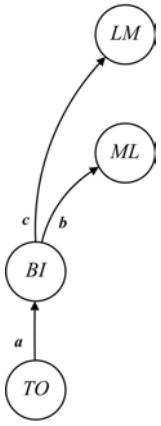


Figure 4.7: Communication in BI

4.2 Receiving and sending totes

```
:: *( a?x; xs:= xs ++ [x]; c!x
    | b!take(xs,%NBATCH) ; xs:= drop(xs,%NBATCH)
    )
```

At the moment BI receives a new transport tote x, this tote is put into the list xs and a signal containing the tote is sent to LM via channel c. If a storage action is signaled by LM, then BI will be able to send a list of four totes to ML via channel b.

5 Miniload output buffer BO

The miniload output buffer BO provides space for temporary storage of totes that have just been retrieved from the miniload and are ready to be sent to the workstation. The buffer capacity is finite with a maximum of 12 totes.

5.1 Communication with surrounding processes

```
proc BO (chan a?: [ttote], b!: ttote, c!: void, val k: nat) =
|[ var ys: [ttote], xs: [ttote] = []
```

BO receives retrieved totes from ML via channel a. These totes are then sent to the output transporter TO via channel b. Channel c is used to send an update information about the number of totes currently present in BO after each time a tote is sent to TO.

5.2 Receiving and sending totes

```
:: *( a?ys; xs:= xs ++ ys
    | len(xs) > 0 -> b!hd(xs); xs:= tl(xs); c!
    )
```

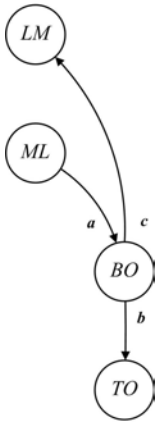


Figure 4.8: Communication in BO

BO receives totes that are retrieved by ML, which is represented by the variable ys, and put them into the list of totes present at BO by using variable xs. Subsequently, the totes will be transferred to TO and a signal will be sent to LM via channel c to indicate the change in the number of totes currently present in BO.

6 Miniload output transporter TO

The miniload output transporter TO can be regarded as the conveyor window in front of the miniload output buffer BO. It receives totes from two sources, namely from the TI and the BO (refer to Figure 4.1).

6.1 Communication with surrounding processes

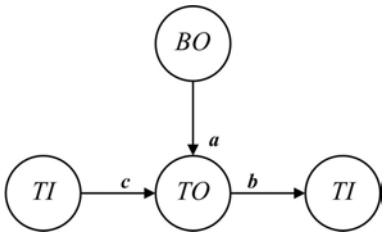


Figure 4.9: Communication in TO

```
proc TO (chan a?: ttote, b!: ttote, c?: ttote, val k: nat) =
|[ var x: ttote, n,i: nat, ws: bool, priority: bool = false
     , j: nat = 0, xs: [(ttote,real)] = [], t: real
```

TO receives totes from BO via channel a. It may also receive a tote from TI via channel c. After a certain delay, the totes will be transferred to the next input transporter TI via channel b. The value k is used to note the miniload index.

6.2 Receiving and sending totes

```
:: *( a?x; xs:= xs ++ [(x,time + 1.0)]
      ; len(xs) > 0
        *> ( a?x; xs:= xs ++ [(x,time + 1.0)]
           | c?x; xs:= xs ++ [(x,time + 1.0)]
           | delay ((hd(xs).1 - time) max 0.0)
             ; b!hd(xs).0; xs:= tl(xs)
           )
    | c?x; xs:= xs ++ [(x,time + 1.0)]
      ; len(xs) > 0
        *> ( a?x; xs:= xs ++ [(x,time + 1.0)]
           | c?x; xs:= xs ++ [(x,time + 1.0)]
           | delay ((hd(xs).1 - time) max 0.0)
             ; b!hd(xs).0; xs:= tl(xs)
           )
    )
```

As mentioned earlier, the `TO` can receive totes (`x`) from two different sources, namely from the output buffer `BO` (via channel `a`) and the input transporter `TI` (via channel `c`). The totes are then put into a list `xs`. A delay is of 1.0 second is then applied to the totes to model the transport time of the totes on the transporter. After this delay, the tote is forwarded to the next `TI` in the system.

Chapter 5
Workstation Model

Contributor: R.Jordan

In this section, the discrete-event simulation model of the workstation area is explained. In the following subsections the most important processes are explained in full detail. There are three workstations present in the system, where a complete workstation system is referred as `WS`.

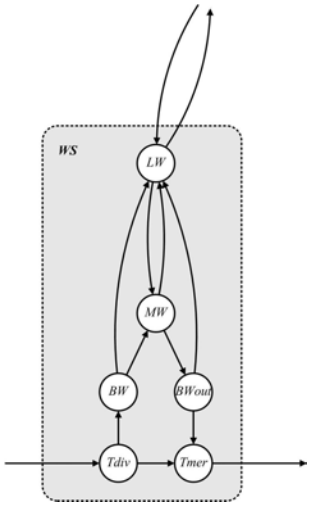


Figure 5.1: Workstation system

All processes belonging to a single workstation are connected in a single process called `WS`, which is depicted in Figure 5.1. A `WS` contains conveyor processes `Tdiv` and `Tmer`, input buffer `BW` and output buffer `BWout`, workstation operator process `MW` and its local controller `LW`.

```
proc WS(chan a?: (subord, nat), b?, c!: (ttote, nat, bool), val Id: nat) =
|[ chan BW2MW, MW2BWOut, BWOut2Tmer, Tdiv2Tmer, Tdiv2BW: (ttote, nat, bool)
      , BW2LW: (nat, bool)
      , LW2BW: nat
      , MW2LW: (nat, nat)
      , mw2viz: bool
      , lw2viz: (field, nat, [nat])
:: Tdiv(b, Tdiv2Tmer, Tdiv2BW, Id)
|| BW(Tdiv2BW, BW2MW, BW2LW, LW2BW)
|| LW(a, LW2BW, MW2LW, BW2LW, lw2viz, Id)
|| MW(BW2MW, MW2BWOut, MW2LW, mw2viz, Id)
|| BWOut(MW2BWOut, BWOut2Tmer)
|| Tmer(BWOut2Tmer, Tdiv2Tmer, c, Id)
|| viz4(mw2viz, lw2viz, Id)
]|
```

1 Workstation transporter `Tdiv` and `Tmer`

The input conveyor windows `Tdiv` and output conveyor windows `Tmer`, are similar to those used at the miniload. `Tdiv` is connected to each workstation and only allow totes with the correct destination identifier to enter the workstation, while `Tmer` merges the outgoing tote-flow from a workstation with the passing conveyor tote-flow (See Figure 5.1). At this moment blocking is not considered and thus totes can always be put on the conveyor.

2 Workstation buffer `BW`

```
proc BW(chan a?: (ttote, nat, bool), b!: (ttote, nat, bool)
           , c!: (nat, bool), d?: nat) =
|[ var xs: [(ttote, nat, bool)] = [], x: (ttote, nat, bool)
     , sortId: nat
:: *( a?x; xs:= xs ++ [x]; c!(x.1, x.2)
    | d?sortId; (xs, x):= findTTote(xs, sortId); b!x
    )
]|
```

The workstation buffer `BW` performs the actual storage of the totes, although the sorting is performed at the local controller `LW`, Section 4. `BW` receives totes from `Tdiv` via channel `a`, stores the totes in list `xs` and sends the totes `sortId` and the `flag` (modeled as `x.1` and `x.2` in the the above code, respectively) to `LW`. These two variables are the only information needed by `LW` to effectively sort the totes in the buffer. The actual tote information contained in `ttote`, is only kept at the buffer.

`BW` receives a signal from `LW` via channel `d` to send a tote with `sortId` to the operator. `BW` calls function `findTTote`, which finds the first tote in the buffer list `xs` containing the correct `sortId`. This tote is then sent to the operator. Function `findTTote` also removes the tote from the list `xs`.

Figure 5.2 shows `BW` and its surrounding processes. The chosen construction of communications in `BW` is prone to deadlock. In order to prevent this deadlock from appearing, `LW` should

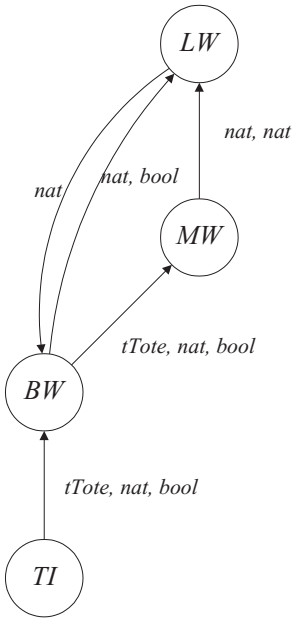


Figure 5.2: Communication in BW

always be able to receive signals from BW. LW is explained in Section 4.

3 Workstation operator MW

```
proc MW(chan a?, b!: (ttote, nat, bool), c!: (nat, nat), v!: bool
          , val id: nat) =
|[ var t: -> real = triangle (9.5, 12.5, 15.5 )
     , x: (ttote, nat, bool)
     , s: real
:: *( a?x; v!false; s:= sample t; delay s
      ; x.0.tote.qty:= x.0.tote.qty - x.0.req
      ; x.0.src:= id
      ; b!x
      ; c!(x.0.tote.sku, x.0.req)
      ; v!true
    )
]|
```

MW represents the operator of the workstation. Figure 5.3 shows the communication between MW with its surrounding processes. In MW the items are picked from the totes. MW receives a tote from BW via channel a. The process time of MW is triangularly distributed with a mean of 12.5 seconds, a minimum of 9.5 seconds and a maximum of 15.5 seconds.

When a tote is received, a sample of the process time distribution is taken and a the process delays for this time interval. Next the required number of items are removed from the tote and the tote is sent to the output buffer via channel b. The information about the required number of items to pick from a tote is included in the tote itself. Subsequently, the informa-

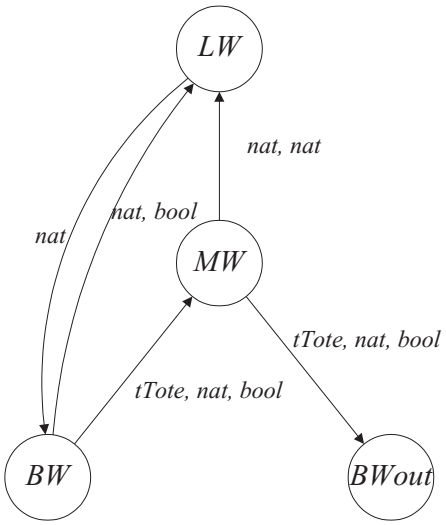


Figure 5.3: Communication in MW

tion of the SKU and the number of items to pick is sent to LW via channel c so it can update its active suborder.

4 Local workstation controller LW

A task of LW is to sort incoming totes in the workstation buffer. This sorting is required because it is not necessarily that all totes belonging to a single suborder fit in a single buffer lane. In order to sort the totes, LW uses an identifier called sortId of each tote. This identifier links a tote to a suborder. All the totes belonging to the same suborder have the same sortId. Besides sortId, LW uses a number of variables to effectively sort the totes in the buffer. These variables are as follows:

```
proc LW(chan a?: (subord, nat), b!: nat, d?: (nat, nat), c?: (nat, bool)
          , v!: (field, nat, [nat]), val WSID: nat) =
|[ var ps: [(subord, nat)] = [], p: (subord, nat)
     , fld: field = <[], [], []>, xs: [nat] = [], x: nat
     , top: 3 * nat = <0, 0, 0>
     , bottom: 3 * nat = <0, 0, 0>
     , length: 3 * nat = <0, 0, 0>
     , arriving: [nat] = []
     , curP: subord = (0, 0, [])
     , currId, skuId, n, lane: nat
     , nTotes: nat = 0
     , flag: bool
```

fld is a vector of three lists of naturals. Each list represents the filling of totes on each buffer lane. The head of the list is the tote at bottom, the tail of the list is the tote at top. top (bottom) represents the sortId of the top (bottom) totes in each buffer lane. length is a vector of three natural values that represent the number of totes in each of the three corresponding buffer lane. curP is the current suborder being processed by the operator

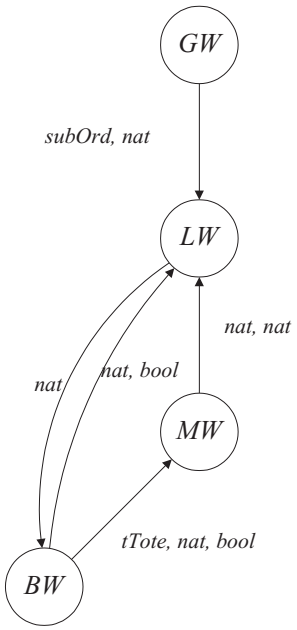


Figure 5.4: The local controller process

and `currId` is the identity of the current suborder. `arriving` is a list of identifiers of all suborders assigned to a particular workstation. This list only contains identifiers (`sortId`) of suborders, which have not yet completely arrived in the buffer. `nTotes` keeps track of the number of totes that have been sent to the operator. `nTotes` can be either one or zero.

The communication between `LW` and its surrounding processes are depicted in Figure 5.4. `LW` controls the stacking of totes in the workstation buffer `BW`. `LW` also decides which totes to send to the operator `MW`. For this purpose, `LW` receives the information about which suborders to process from `GW`. When the operator `MW` has finished a tote, a signal is sent `LW`.

`LW` can perform a number of tasks in parallel namely receiving a suborder from GW, finishing current suborder and start a new one, updating the current suborder, sorting an incoming tote into one of the buffer lanes, and sending a signal to the buffer to move a tote to the operator. Note that as in other controller processes (`GM`, `LM`, and `GW`), these tasks are executed in a timeless instant.

4.1 Receive a suborder from `GW`

```
:: *( a?p; ps:= ps ++ [p]; arriving:= arriving ++ [p.1]
```

As we can see in Figure 5.4 `LW` receives from `GW` a suborder and a natural, representing a new suborder being assigned to this workstation. The natural data type is the `sortId` belonging to that suborder. Both the suborder and its `sortId` are stored in the list `ps`. The list `arriving`, which is used to identify the sequence of arriving suborders, is extended with the `sortId` from this suborder.

4.2 Finish current suborder and start a new one

```
| len(curP.list) = 0 and len(ps) > 0 -> (curP, currId):= hd(ps); ps:= tl(ps)
```

When the current suborder does not contain orderlines anymore (that is, `len(curP.list) = 0`), a new suborder should be started at the workstation. When there are still suborders assigned to this workstation, these are stored in `ps` by the task mentioned above. Thus, `curP` and `currId` are updated and the `hd(ps)` is removed from `ps`.

4.3 Update current suborder

```
| d?(skuId, n); nTotes:= nTotes - 1; curP:= updActiveorder(curP, skuId, n)
```

When the operator has finished working on a tote, `MW` sends the information of the tote orderline to `LW`. This information contains the SKU identifier and the number of items removed from that tote. `LW` now calls the function `updActiveOrder` to remove an orderline from a suborder. This function is discussed in Section 6. Now `nTotes` can be updated since a tote has just left the operator.

4.4 Sort an incoming tote into one of the buffer lanes

```
| c?(x, flag); lane:= detInLane(top, length, arriving, x)
  ; fld.lane:= fld.lane ++ [x]; length.lane:= length.lane + 1
  ; top.lane:= x
  ; ( not flag -> skip
    |     flag -> arriving:= updArriving(arriving, x)
    )
  ; bottom:= updBottom(bottom, fld)
  ; v!(fld, lane, arriving)
```

The local controller receives information from `BW` when a tote arrives in the buffer. The information received by `LW` (x, $flag$) contains the `sortId` of the tote that arrived and the `flag`. The `sortId`, here `x`, is the input for the function `detInLane` together with the variables `top`, `length` and `arriving`. The function `detInLane` is discussed in Section 5. The function returns a natural: `lane`, which identifies the lane in which the new tote should be put in. The local controller does not keep track of the contents of the totes, because this information is not needed for the sorting.

When `lane` is known, the variables `top`, `length`, `bottom` and `fld` are updated. If the tote is the last one of the suborder, this tote has a "$flag = true$" attached to it. The `sortId` belonging to this tote should now be removed from `Arriving`. Function `updArriving` updates the `arriving` list. If the tote is not flagged, `arriving` should not be updated.

The function `UpdArriving` simply removes the identity `x` from the list `arriving`. Both these variables are input requirements for the function. The function returns the updated list `arriving`, without changing the order of identifiers in the list.

Finally when the buffer is updated, `fld`, `lane` and `arriving` are communicated to `VizWS`, which writes the necessary information to the visualization tools.

4.5 Send a signal to the buffer to move a tote to the operator

```
| len(curP.list) > 0 and nTotes < 1 and (or, j <- 0..2, bottom.j = currId )
  -> lane:= detOutLane(bottom, length, currId)
   ; xs:= xs ++ [hd(fld.lane)]; fld.lane:= tl(fld.lane)
   ; length.lane:= length.lane - 1
   ; ( length.lane  = 0 -> top.lane:= 0
     | length.lane /= 0 -> skip
     )
   ; bottom:= updBottom(bottom, fld); nTotes:= nTotes + 1
   ; v!(fld, lane, arriving)
| len(xs) > 0 -> b!hd(xs); xs:= tl(xs)
```

There are two alternative statements in the above code, as represented by the sign |. `LW` determines which type of tote to send to the operator `MW` and stores this information in `xs`. The information is sent to the buffer `BW` via channel `b` in the second alternative statement. This two-statement construction is created to keep the communication between `LW` and `BW` deadlock free.

The first line of the code above sets the criteria for sending a new tote to the operator. First `nTotes` should be smaller than one. `nTotes` indicates if there is a tote at the operator or not. `nTotes` is updated every time the operator finishes a tote, as explained earlier. Initially `nTotes` is equal to zero, to indicate there are no totes at the operator.

As long as `curP` still contains orderlines, there should still be totes belonging to this suborder in the buffer `BW`. Of course it could be possible that there are no totes belonging to this suborder in the buffer. In that case the totes are still somewhere between the miniload and the workstation. Whether or not there are actually totes available in the buffer, is checked by the statement (line 1): `(or, j <- 0..2, bottom.j = currId)`

This statement checks if one of the elements of `bottom` contains a tote belonging to the current suborder. As long as there are no totes belonging to the current suborder in the buffer, none can be sent to the operator.

If the buffer does contain totes belonging to the current suborder, these totes should have been sorted in such a way that at least one of the elements of `bottom` contains a tote for this suborder, which is done using a stacking algorithm. When there is at least one tote available at the `bottom` of the buffer, the function `detOutLane` is called to determine from which lane a tote should be picked.

The function `detOutLane` chooses the tote in the lane with the greatest length from the totes belonging to the current suborder. This is done to keep the possibility of buffer lane overflow as low as possible. Function `detOutLane` returns a natural value: `lane`, which indicates from which buffer lane a tote should be removed.

When the right `lane` is already determined, the variables `fld`, `bottom` and `length` are updated. `Top` will only be updated when the `length` of the current lane is zero (lines 5 and 6), because now there is no more tote present in this buffer lane. Variable `nTotes` is increased by one, so `LW` knows it has already picked a tote to send to the operator. Therefore this statement may not execute again.

5 Function detInLane

```
func detInLane(val top, length: 3 * nat, arriving: [nat], x: nat) -> nat =
|[ var i: nat = 0
     , n: nat = 0
     , NLane: nat = 3
     , arr: [nat] = arriving
     , se: [(nat, nat)] = []
     , sg: [(nat, nat, nat)] = []
```

This function applies the stacking rules to the incoming totes. Function detInLane uses the variables top, length and arriving to sort a tote belonging to a certain suborder identified as x. Variable arriving is a list of suborder identifiers, in the sequence in which they were assigned to the workstation. Arriving can contain more than three sortId's due to the fact that a new suborder is already assigned at the entry of the workstation area, yet only the totes with identifiers at the first three positions of arriving can actually arrive at the workstation.

Variable NLanes is the variable which scales this function to the number of buffer lanes, in this case three. This function will also work when the number of buffer lanes is increased or decreased. Keep in mind that the number of incomplete suborders sent to the workstation as determined at GW, should be no greater than the number of buffer lanes.

```
:: len(arr) > 0
   *> ( x  = hd(arr) -> arr:= []
      | x /= hd(arr) -> n:= n + 1; arr:= tl(arr)
      )
```

When the function detInLane is called, it first determines which position arriving the current tote (x) belongs to (lines 6 through 9). Depending on the position in arriving a different rule applies, variable n is used to represent the place in arriving. When x belongs to the first suborder in arriving, n is equal to zero. When x belongs to the second suborder in arriving, n is equal to one, etc.

```
 ; i < NLane
   *> ( ( x < top.i -> skip
        | x = top.i -> se:= se ++ [(i, length.i)]
        | x > top.i -> sg:= sg ++ [(i, x - top.i, length.i)]
        )
        ; i:= i + 1
      )
```

Now the top of each buffer lane is verified versus x. As the above code suggests, there are three possible outcomes.

1. The identity of x is smaller than top. In this case, x can not be placed on this lane. When a tote is put on another tote which has a higher sortId, the tote is blocked. The suborders should be processed in increasing order. Since totes cannot be rearranged within a lane, a low sortId may never be placed on a higher sortId.

2. The identity of x is equal to top. As such, x can always be placed in this lane. A list se is made which contains the lane number and the length of the lane which has a top equal to x.
3. The identity of x is greater than top. For this case, a list sg is created which stores the lane number, the difference between x and the top of the lane and the length of the lane.

```
 ; ( len(sg) <= n -> skip
   | len(sg) >  n -> i:= hd(sort(sg, inc12)).0
     ; se:= se ++ [(i, length.i)]
   )
 ; ret hd(sort(se, inc1)).0
]|
```

Now that all buffer lanes are evaluated and the lists sg and se have been created, the above final part of the function can run. In this final part there are two possibilities: there are either more items in the list sg than the value of n or there are less or equal items in sg.

If there are no more items in sg than the value of n, the current tote x can only be placed on a row with the same identifier. One of the elements of se should be chosen. List se is sorted in such a way, that the first element of se contains the lane with the shortest length. The sorting is performed by function inc1. When two lane are of equal length, the most upstream lane is put before the more downstream one.

The first element of list se now contains the lane in which the tote should be sorted. The lane identifier is returned to LW.

```
func inc1(val x, y: (nat, nat)) -> bool =
|[ ( x.1 /= y.1 -> ret x.1 < y.1
   | x.1  = y.1 -> ret x.0 < y.0
   )
]|
```

When there are more items in sg than the value of n, this means the current tote x is allowed to claim a (extra) lane for itself. List sg is sorted using function inc12. This function first sorts sg on the smallest difference between x and top. In this case the smallest value of (x - top). If this sorting results in two lanes having a similar difference, the shortest lanes is chosen. If this also does not generate a single result, the most upstream lane is chosen.

```
func inc12(val x, y: (nat, nat, nat)) -> bool =
|[ ( x.1 /= y.1 -> ret x.1 < y.1
   | x.1  = y.1 -> skip
     ; ( x.2 /= y.2 -> ret x.2 < y.2
       | x.2  = y.2 -> ret x.0 < y.0
       )
   )
]|
```

When the function inc12 has returned a candidate lane from the list sg, this lane identifier and its length are added to the list se. List se now contains all the possible lanes in which to put the current tote. Now like above, list se is sorted using function inc1 and the first element of se is returned to LW.

6 Function updActiveOrder

Function updActiveOrder is called by LW when the operator finished working on a tote. Since LW does not keep track of the contents of totes, it has to receive this information from MW when it has finished a tote.

```
func updActiveorder(val s: subord, skuid, n: nat) -> subord =
|[ var p: subord = s
     , q: subord = (s.id, s.seq, [])
     , r: line
:: len(p.list) > 0
   *> ( r:= hd(p.list); p.list:= tl(p.list)
        ; ( r.sku /= skuid -> q.list:= q.list ++ [r]
          | r.sku  = skuid -> skip
            ; ( n <  r.qty -> r.qty:= r.qty - n; q.list:= q.list ++ [r]
              | n >= r.qty -> skip
              )
            ; q.list:= q.list ++ p.list; ret q
          )
      )
]|
```

UpdActiveOrder searches through the suborder it is presented with (s) to find the orderline, with the matching SKU. When the correct orderline is found (r.sku = skuid), the right amount of items are removed from this line. Recall that in order to satisfy an orderline, sometimes multiple totes holding the same SKU have to be processed at the workstation. As such, it is possible that only the requested number of items has to be updated (n < r.qty) or the complete orderline has to be removed from the suborder (n >= r.qty). Once the update is done, the suborder is returned to LW.

Chapter 6
Data Collection

Contributor: R.Andriansyah

The complexity of the simulation model can be seen from the data structure that is used as input for the model. Two types of data are distinguished in the model, namely *product-related data* and *equipment-related data*. Product-related data are mainly obtained from the real data of the reference case distribution center, while the equipment-related data are derived from the assumptions used in a simulation study that was done earlier by Vanderlande Industries at the same distribution center.

1 Product-related data

Figure 6.1 depicts the distribution of suborder length. The distribution is used to determine the number of orderlines contained in each of the suborder generated by GO.

For the (*s, S*) policy for the item replenishment, a reorder point (*s*) of 2-day inventory is used. The order-up-to level (*S*) is chosen as 8-day inventory. As such, whenever the number of items of an SKU in the miniloads drops below the 2-day inventory level (*min*), replenishment items for the SKU are ordered such that the the number of items is raised to 8-day inventory (*max*). This policy applies for all 1624 SKUs in the system, where each SKU has a different *min* and *max* level.

Another product-related data is the number of items in a full tote for each SKU type. This data is based on the assumption used in. Specifically, two discrete uniform distributions are used, where the values are spread from 6 to 12 and from 12 to 24. The realizations are generated twice as much from the first distribution as from the second distribution. Eventually, an average of 12 items in a full tote is obtained.

Summarizing, the product-related data consists of the distribution of suborder length (Figure 6.1), the demand/week for each SKU, the *min* and *max* value of the number of items in

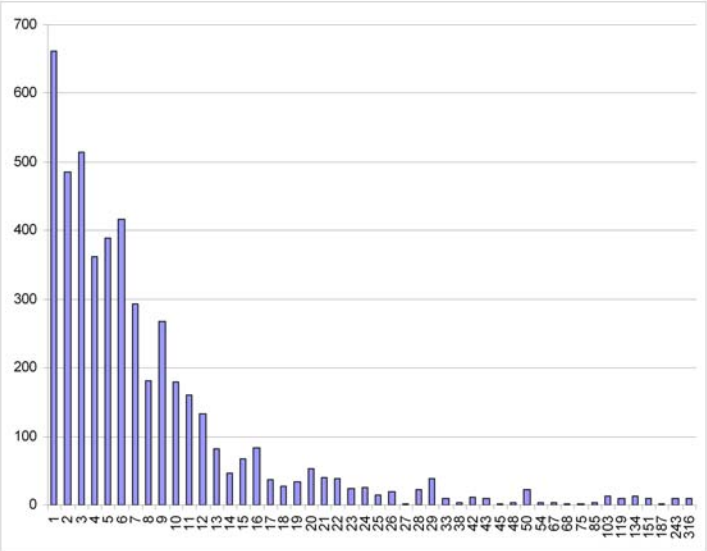


Figure 6.1: Suborder length distribution

the miniload for each SKU, and the number of items in a full tote for each SKU.

2 Equipment-related data

The equipment-related data represents the various processing time and data concerning the physical attributes of the system. A summary of the data is presented in Table 2. From this table it can be seen that no data is available regarding the retrieval time for 0 tote, which is the time needed for an empty crane movement from the retrieval position. This is due to the fact that it is not possible for the crane to move without retriving any totes while in the retrieval position, since there are always totes to be retrieved available in the retrieval queue at the miniload.

Table 6.1: Equipment related data

No	Name	Value
1	Storage time (seconds)	
	0 tote (empty crane movement)	23.6
	1 or 2 totes	30.6
	3 or 4 totes	37.5
2	Retrieval time (seconds)	
	1 tote	16.5
	2 tote	23.5
	3 tote	30.5
	4 tote	37.5
3	Operator picking time per item (seconds)	triangular (9.5, 12.5, 15.5)
4	Transport time for one conveyor window (seconds)	1.0
5	Number of storage location per miniload	6250
6	Number of buffer places per buffer lane at one workstation	14

Chapter 7
Validation

Contributor: R.Andriansyah

To validate the proposed simulation model, an experiment is carried out to compare the performance of the model to that of the real system. As a performance measure, we use the number of totes per hour that leaves the Order Sequencing Point (OSP), or `TWin` in our simulation model. From a previous discussion with the systems engineer at the reference case DC, we know that there are approximately 700 totes that pass the OSP per hour.

In the real system, the buffer places at each of the workstation has a capacity of $3 \times 14 = 42$ totes. Since there are three workstations in the system, the total number of totes that can be put into the buffer places simultaneously is 126 totes. Since there will also be totes that are still on the conveyor loop both in the miniload or the workstation area, it is reasonable to vary the threshold number of totes in the system between 110 to 150 totes. Note that the actual number of totes in the system may exceed the threshold. A more precise definition about the threshold will be given in the next chapter. The result from our validation experiment is as follows.

Table 7.1: Validation experiment

Threshold no. of totes	Throughput (totes/hour)	95% CI	
110	691.01	679.48	702.55
120	702.91	691.39	714.43
130	713.12	701.82	724.41
140	723.19	712.84	733.54
150	730.78	720.84	740.72

The results indicates that the throughput of totes from the OSP or the `TWin` in the model is close to the real system.

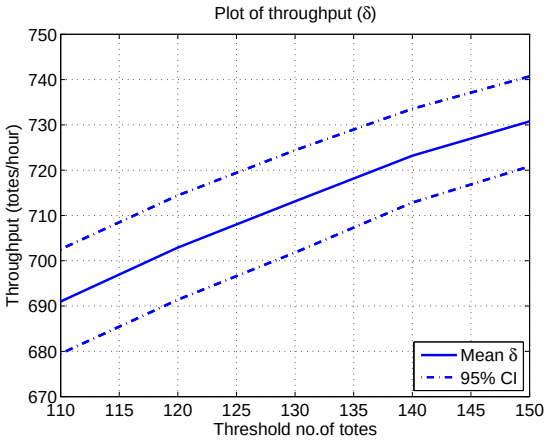


Figure 7.1: Tote throughput at Order Sequencing Point (OSP)

Chapter 8
Experiments

Contributor: R.Andriansyah

A set of experiments are conducted to understand the behavior and to measure the performance of the system. In particular, we are interested in system's throughput (δ) and flowtime. We define suborder as the unit of measurement for both throughput and flowtime. Recall that a suborder, which is generated by GO, consists of one or more order lines. Each of the order lines consist of the SKU type and its required quantity. An order line corresponds with one or more product totes of a particular SKU type.

1 Effect of system traffic

One of the design parameters in the model is the maximum number of totes at the workstation area for suborder release. This parameter is defined due to the high variability of the suborder length. As mentioned earlier, a suborder may consist up to 316 order lines. In order to avoid a very heavy traffic of totes in the workstation area, a *threshold* is applied such that *a new suborder will be processed if and only if the number of totes currently present in the workstation area is lower than the threshold.* This threshold value is defined in the design variable.

In the experiment, the threshold number of totes is varied from 10 to 200 with an increment of 10. Additionally, we also include values between 1 to 10. For each value of this threshold, ten simulation replications are conducted. The simulation is terminated once the total number of finished suborder reaches 10,000 suborders. This amount of suborder is selected because at that point the simulation has reached a steady state condition, as observed in several preliminary experiments. Table 1 shows the results from the experiments.

Figure 8.1 shows the plot of throughput against the threshold number of totes. There is clearly an asymptotic relation between the two variables. The throughput of the system increases by allowing more totes to be sent to the workstation area for item picking. However, at a certain value of threshold number of totes, there is no further additional increase of throughput.

Table 8.1: Effect of threshold number of totes

Threshold no. of totes	Mean flowtime (seconds/suborder)	Mean throughput (suborders/hr)	Mean no. of totes
1	320.86	11.15	7.89
2	322.97	13.14	9.31
3	321.54	15.10	10.74
4	319.22	17.12	12.24
5	314.33	19.10	13.72
6	308.93	21.09	15.24
7	303.34	23.01	16.75
8	297.90	24.85	18.19
9	292.30	26.67	19.61
10	287.47	28.28	20.98
20	258.16	41.26	31.94
30	268.24	48.57	40.25
40	294.43	52.49	47.12
50	322.33	54.63	52.91
60	347.57	56.11	57.24
70	368.52	56.88	60.93
80	384.82	57.81	63.14
90	399.17	58.19	65.83
100	411.26	58.72	67.46
110	423.95	58.70	69.66
120	435.39	58.80	71.19
130	443.51	59.11	72.29
140	453.21	59.20	73.50
150	463.39	59.12	74.42
160	470.35	59.28	75.01
170	476.41	59.39	75.86
180	483.84	59.38	76.26
190	489.38	59.35	77.03
200	495.58	59.21	78.34

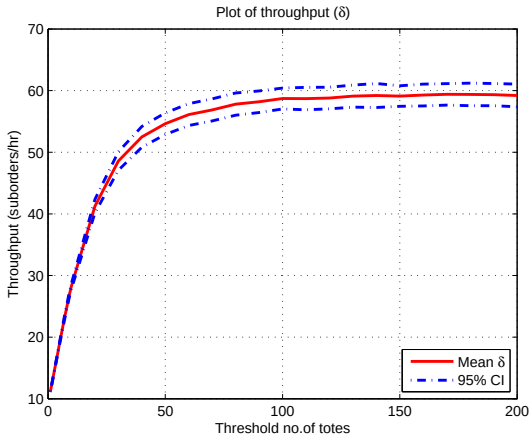


Figure 8.1: Plot of suborder throughput

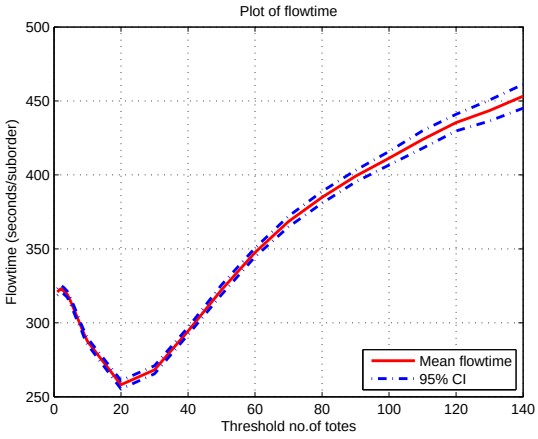


Figure 8.2: Plot of average suborder flowtime

This maximum throughput suggests the system capacity under the given input variables of threshold number of totes.

Another interesting result is obtained for the plot of mean flowtime against the threshold number of totes (Figure 8.2). The figure can be divided into two parts, where a declining trend characterizes the first part and an increasing asymptotic trend is identified in the second part.

When the threshold number of totes is very low (*i.e.* smaller than 20), the average flowtime shows a declining trend. This can be explained by the time-out behavior of the miniloads. Recall that a miniload waits until a batch of 4 totes to be stored or retrieved is formed. However, if the batch size has not reach 4 totes after 2 minutes, the miniload will take all totes that are currently available and subsequently performs storage or retrieval, depending on the current position of the miniload crane. As such, if the threshold number of totes is set very low, then a batch of 4 totes will hardly be formed. This will lead to a very frequent occurrence of time-out at the miniload, which eventually increases the suborder flowtime. Allowing more totes to be sent to the workstation area will reduce the number of time-outs, hence we see the declining flowtime by increasing the threshold number of totes.

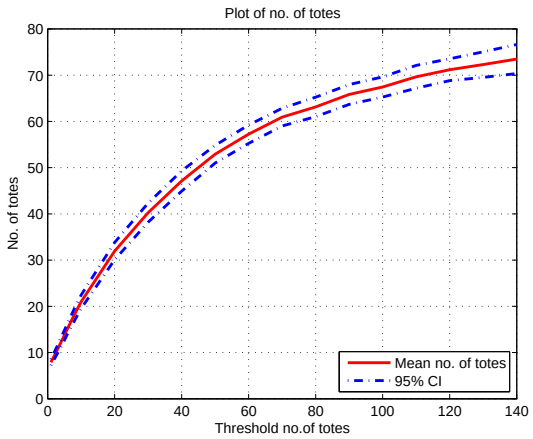


Figure 8.3: Plot of average number of totes at workstation area

When the threshold number of totes is increased more than 20 totes, the flowtime increases as well. This can be explained by the fact that increasing the threshold number of totes allows for a longer queueing at the workstation buffer. As such, the tote waiting time at the queue increases and so does the flowtime of suborder. In this case, the long queue of totes at the workstation area is mainly responsible for a high suborder flowtime.

Finally, figure 8.4 shows the plot of suborder flowtime against throughput. The effect of time-out behavior as explained previously can also be observed in this figure, namely a high flowtime for a low throughput (*i.e.* for throughput ≤ 40 suborders/hr). As expected, when the system reaches its maximum capacity, the throughput can no longer be increased and the flowtime goes to infinity.

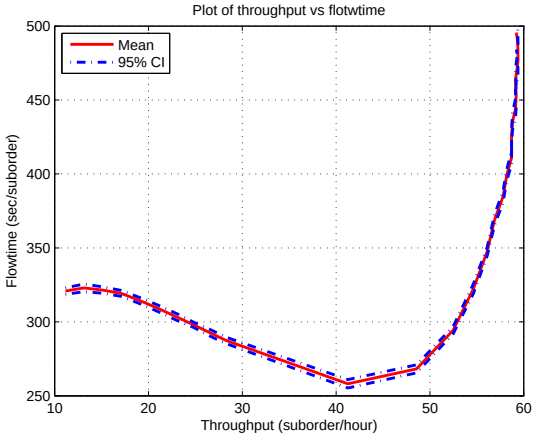


Figure 8.4: Plot of suborder flowtime against throughput

2 Effect of altering the number of miniloads

Among the central advantages of the proposed model architecture are the flexibility and modularity. Our model architecture is developed in such a way that altering the model structure can be done quite easily. The value of constant variables, for example, can be changed in using the constant identifiers. Using this feature, we are able to alter the number of miniloads and/or workstations in the system without affecting the rest of the processes in the model.

In this section, we will show how the different number of miniloads affect the performance measures namely the suborder throughput and flowtime. To change the number of miniloads, we simply alter the value of the constant variable `NML` from 5 to 3, 4, and 6. Running the simulation model using these various parameter values, we obtain the results as depicted in Figure 8.5 and Figure 8.6.

As the figures suggest, subtracting miniloads from the system causes a significant loss of throughput and a notable increase in flowtime for all values of threshold number of totes. However, adding a miniload to the system does not necessarily lead to a significant increase

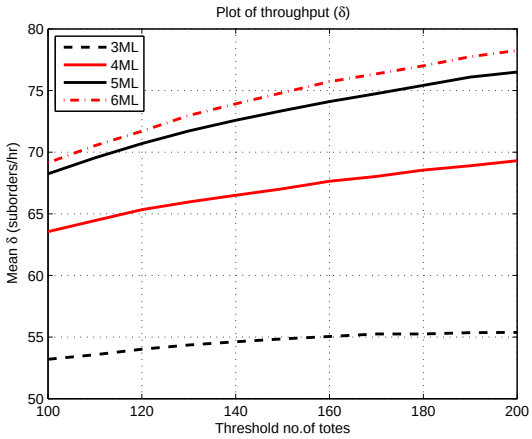


Figure 8.5: Throughput various miniloads

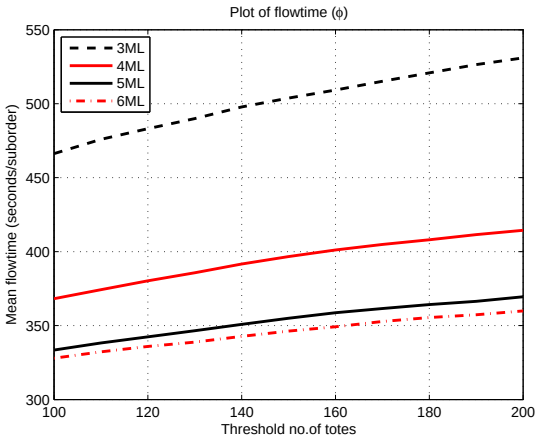


Figure 8.6: Flowtime various miniloads

of throughput or decrease of flowtime. It is obvious that adding one miniload from four to five will increase the performance less than adding one miniload from three to four miniloads. This result can further be used to justify an addition/removal of miniloads.

Chapter 9
Conclusion and Future Work

Contributor: R.Andriansyah

1 Conclusion

This report has elaborated the process of modeling an operational, industrial scale DC using χ 1.0. The reference case DC is characterized by an AS/RS, namely the miniload-workstation order picking system. The main purpose of the study was to develop a very detailed simulation model that can be used in the performance analysis of AS/RS-supported DC.

We created a model architecture that is both flexible and modular, such that different control rules, design parameters and model structures can be incorporated in a quick and simple way. The main features of the proposed model architecture include a decentralized control structure and minimized communication between processes. Autonomous controllers that are able to make independent decisions are introduced at different layers in the model. As such, different control rules can be implemented locally and changes in a certain process will not affect other processes in the model as long as the communicated data types between processes remains the same. Furthermore, the minimized communication allows an efficient use of information in all parts of the model.

The results from the validation experiment indicated that our simulation model produced results that are close to the real system with regards to the throughput totes per hour. Other experiments have shown the effect of the system's traffic intensity on the suborder flowtime and throughput. The modularity of the model architecture is highlighted by the experiment to see the effect on the system performance by involving different number of miniloads. Similar experiments involving various number of workstations are also possible. In general, we observe that the current system configuration is balanced in terms of throughput and flowtime.

2 Future Work

The resulting simulation model can be further used for several purposes. The flexibility of the model architecture allows one to investigate the effect of various control strategies at different layers both in the miniload and/or workstation area to the overall system performance. Furthermore, owing to the modularity of the model, applying a slight change to the current model architecture will allow for modeling other system structures. An example of other structures include cases in which the workstations are located far apart of one another, which is commonly encountered in large DCs. Also, we note that the system that is considered in this study is suitable for handling slow-moving products. DCs that handle fast-moving products will require different system configurations. We argue that by applying small adjustments to the proposed model architecture, other system configurations are also possible to be modeled. With this regard, performance analysis of various types of distribution centers becomes an appealing research direction.

As we mentioned earlier, this study is the first step towards developing a fast, simple, yet accurate performance analysis method based on simulation models. Our next approach is to use the concept of Effective Process Time (EPT) [HS00] to develop a method for *aggregate modeling*. EPT is generally defined as the total amount of time a job could have been, or actually was, processed on a machine [JEC+03]. An aggregate model reduces the details that need to be incorporated in the model, where all sources of variability are aggregated into a single process time distribution, EPT. Jacobs *et al.* [JEC+03] proposed an algorithm to measure EPT realizations from arrivals and departure data and applied it to a semiconductor industry. Subsequently, [KOC08] extended the use of EPT for aggregate modeling of manufacturing systems. Given the latest findings from EPT and aggregate modeling, we foresee an appealing opportunity of developing aggregate modeling techniques for AS/RS distribution centers. Once a method for aggregate modeling has been developed, a reference model is needed assess the quality of the aggregate modeling method. For this purpose, we will use the results from the detailed simulation study explained in this report.

Acknowledgement

This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute program (BSIK03021). We would like to thank Bruno van Wijngaarden, Roelof Hamberg, Jacques Verriet, Ivo Adan, and Jacques Resing for their feedback during model development.

Bibliography

[APH+07] M.F. van Amstel, E. van de Plassche, R. Hamberg, M.G.J. van den Brand, J.E. Rooda. Performance analysis of a palletizing system. SE Report 2007-09, Eindhoven University of Technology, Eindhoven, 2007.

[FAL07] FALCON website. `http://www.esi.nl/short/falcon/`, 2007.

[HR08] A.T. Hofkamp, J.E. Rooda. Chi 1.0 Reference Manual. SE Report 2008-04. Technische Universiteit Eindhoven, Department of Mechanical Engineering, Systems Engineering Group, Eindhoven, July 2008.

[HS00] W.J. Hopp, M.L. Spearman. *Factory Physics: Foundations of Manufacturing Management*. McGraw-Hill, London, second edition, 2000.

[JEC+03] J.H. Jacobs, L.F.P. Etman, E.J.J van Campen, J.E. Rooda. Characterization of operational time variability using effective process times. *IEEE Transactions on Semiconductor Manufacturing*, 16: 511-520, August 2003.

[KOC08] A.A.A Kock. *Effective Process Times for Aggregate Modeling of Manufacturing Systems*. PhD thesis, Systems Engineering Group, Technische Universiteit Eindhoven, June 2008.

[SPP98] E. Silver, D.F. Pyke, R. Peterson. *Inventory Management and Production Planning and Scheduling*. John Wiley & Sons, New York, 1998.

Appendix A

χ processes

```
//// ENVIRONMENT

from standardlib import *

DEFINE %NML 5        // no. of miniloads
DEFINE %NWS 3        // no. of workstations
DEFINE %NTOTE 5503   // no. of totes after initialization
DEFINE %NCELL 6250   // no. of storage cells in a miniload
DEFINE %NWIP 100     // threshold no. of totes
DEFINE %NSKU 1624    // no. of SKUs in the system
DEFINE %MAXSUBORD 9  // no. of maximum active suborders
DEFINE %NBATCH 4     // batch size at miniload

////////////////////////////////////////////// DATA TYPES

// identified by variables: p,q,r
type line = ( sku:    nat      // ordered product type
            , qty:    nat      // ordered quantity
            )
   , subord = ( id:     nat      // belonging to order identity
              , seq:    nat      // sequence number
              , list:   [line]   // order line
              )

// identified by variables: x,y,z
   , item = ( sku:    nat      // product type
            , qty:    nat      // quantity
            )
   , ptote = ( id:     nat      // tote identifier
             , timeIn: real     // starttime of tote
             , sku:    nat      // type of items
             , qty:    nat      // number of items in tote
             )
   , ttote = ( tote:   ptote    // product tote information
             , ord:    nat      // belonging to order (99 = no info)
             , seq:    nat      // belonging to suborder (99 = no info)
             , src:    nat      // Source (99 = no info)
             , des:    nat      // Destination (99 = no info)
             , req:    nat      // Items to pick (99 = no info)
             )

   , field = 3 * [nat]          // buffer lane fill

////////////////////////////////////////////// MODEL DECLARATION

model S( val maxTotes: nat ) =
|[ chan pm2gm, gm2pm: subord
      , gm2gw: (subord, nat)
      , ws2ex, TMIn2ml, ml2TMOut, ml2ws: ttote
      , c2TMIn: 2 # ttote
      , lm2gmIn: %NML # (real,nat,nat)
      , lm2gmRes: %NML # (real,nat)
      , lm2gmOut: %NML # nat
      , gm2lm: %NML # (nat,nat,nat,nat)
```

```
      , gw2gm: void
      , TWIn2ws, ws2TWOut: (ttote,nat,bool)
      , gw2lw: %NWS # (subord,nat)
      , TWout2ex: (nat, real)
      , gm2pm2: void
      , gm2pm3: void
:: env ( pm2gm, c2TMIn.0, ws2ex, gm2pm, TWout2ex, gm2pm2, gm2pm3 )
|| MLEnv ( gm2pm, pm2gm, c2TMIn, TMIn2ml, ml2TMOut, ml2ws
         , lm2gmIn, lm2gmRes, lm2gmOut, gm2lm, gm2gw, gw2gm
         , gm2pm2, gm2pm3
         )
|| MLClus ( TMIn2ml, ml2TMOut, lm2gmIn, lm2gmRes, lm2gmOut, gm2lm )
|| WSEnv ( ml2ws, TWIn2ws, ws2TWOut, c2TMIn.1, ws2ex, gw2lw
         , gw2gm, gm2gw, TWout2ex, maxTotes
         )
|| WSClus ( TWIn2ws, ws2TWOut, gw2lw )
||

////////////////////////////////////////////////// ENVIRONMENT

proc GO(chan a?: void, b!, c?: (nat, real), d!: subord, e?: void
          , f!: void) =
|[ var dm:-> real = uniform(0.0,1.0)
     , j,k,n: nat, p: [line], x: (nat,nat,real), xs: [(nat,nat,real)]
     , dr:-> real = uniform(0.0,1.0), r: real
     , dq:-> nat = poisson(0.22), q: nat
     , i: nat = 0, t: real
     , skuvec: [(nat,nat,real)]
:: e?; f!
 ; *( a?
      ; skuvec:= initskus()
      ; p:= []
      ; j:= numLines(sample dm)
      ; j > 0
        *> ( r:= sample dr
             ; k:= searchSKU(r,skuvec)
             ; q:= 1 + sample(dq)
             ; n:= numreq(q,k,skuvec) min 5
             ; p:= p ++ [(k,n)]
             ; skuvec:= updatesku(k,n,skuvec)
             ; j:= j - 1
           )
      ; d!(i,i,p); i:= i + 1
    )
|| *( c?(n, t); b!(n, t) )
||

proc PM( chan a?, b!, c?, d!: subord, e!: (nat, real), f?, g!: void
           , h?, i!: void ) =
|[ var p, q: subord, ps: [subord] = [], qs: [nat] = []
:: *( a?p; ps:= ps ++ [p]
    | len(ps) > 0 -> b!hd(ps); ps:= tl(ps)
    | c?q; d!q; qs:= qs ++ [q.seq]
    | len(qs) > 0 -> e!(hd(qs),time); qs:= tl(qs)
    | f?; g!
    | h?; i!
```

```
    )
]|

proc PR( chan a?: subord, b!: line ) =
|[ var p: subord
     , IPs: [nat], IP: %NSKU * nat
     , mns: [nat], mn: %NSKU * nat
     , mxs: [nat], mx: %NSKU * nat
     , s : line, Qty, skunr: nat
:: IP:= list2vec(IPs); IPs:= initIP()
 ; mns:= initMin(); mn:= list2vec(mns)
 ; mxs:= initMax(); mx:= list2vec(mxs)
 ; *( a?p
      ; len(p.list) > 0
        *> ( s:= hd(p.list); p.list:= tl(p.list); skunr:= s.sku
             ; IP.(skunr):= IP.(skunr) - s.qty
             ; ( IP.(skunr) <= mn.(skunr)
                 -> Qty:= mx.(skunr) - IP.(skunr)
                  ; b!(skunr,Qty)
                  ; IP.(skunr):= IP.(skunr) + Qty
               | IP.(skunr) > mn.(skunr)
                 -> skip
               )
           )
    )
]|

proc GR( chan a?: line, b!: ttote, v!: void ) =
|[ var q, r: line, qs: [line] = []
     , tcs: [nat], tc: %NSKU * nat
     , i,k: nat = (0,0)
     , Qty: nat
     , IPs: [nat], IP: %NSKU * nat
:: IPs:= initIP()
 ; tcs:= inittote()
 ; IP:= list2vec(IPs)
 ; tc:= list2vec(tcs)
 ; k < %NSKU
   *> ( IP.k > 0
        *> ( Qty:= IP.k min tc.k
             ; b!((i,time,k,Qty),99,99,99,99,99)
             ; i:= i + 1; IP.k:= IP.k - Qty
             ; delay 10.0
           )
        ; k:= k + 1
      )
 ; *( a?q; qs:= qs ++ [q] )
|| *( len(qs) > 0 -> r:= hd(qs); qs:= tl(qs)
      ; r.qty > 0
        *> ( Qty:= r.qty min tc.(r.sku)
             ; b!((i,time,r.sku,Qty),99,99,99,99,99)
             ; v!
             ; i:= i + 1; r.qty:= r.qty - Qty
             ; delay 10.0
           )
    )
```

```
]|

proc EX( chan a?: ttote, b?: (nat, real), c?: (nat,real), d?: void
           , u!, v!: (nat, real), w!: real ) =
|[ var x: ttote
     , i: nat, j, k: nat = (0,0), t, f: real, fs: [real] = []
     , ps: [(nat, real)] = []
     , mf: real = 0.0, tstart: real
     , s2phi: real = 0.0, th: real
:: *( a?x; k:= k + 1
      ; u!(k,(time-tstart)/3600)
    | b?(i, t); ps:= ps ++ [(i, t)]
    | c?(i, t); j:= j + 1; (f,ps):= detFlowTime(ps, i)
      ; f:= t - f; fs:= fs ++ [f]
      ; v!(j,(time-tstart)/3600)
      ; ( j > 1  -> s2phi:= s2phi * (j - 2) / (j - 1)  + (1 / j) * (f - mf)^2
       | j <= 1 -> s2phi:= 0.0
       )
      ; mf:= mf * ( (j - 1) / j ) + f / j
      ; th:= j/(time-tstart)
      ; ( j mod 100 = 0 -> !!mf, "\t", th, "\t", s2phi, "\n"
       | j mod 100 > 0 -> skip
       )
      ; w!mf
    | d?; tstart:= time
    | j >= 10000 -> skip; delay -1.0
    )
]|

proc viz2( chan a?, b?: (nat, real), c?: real, d?: void ) =
|[ var x, y: nat, z, t: real, i: nat = 0
:: !!"time:",time,"\t","dest:plot1\tline-type:1\tgraph-title:ML_1\n"
 ; !!"time:",time,"\t","dest:plot1\tline-type:2\tgraph-title:ML_2\n"
 ; !!"time:",time,"\t","dest:plot1\tline-type:3\tgraph-title:ML_3\n"
 ; !!"time:",time,"\t","dest:plot1\tline-type:4\tgraph-title:ML_4\n"
 ; !!"time:",time,"\t","dest:plot1\tline-type:5\tgraph-title:ML_5\n"
 ; !!"time:",time,"\t","dest:plot4\tline-type:1\tgraph-title:WS_1\n"
 ; !!"time:",time,"\t","dest:plot4\tline-type:2\tgraph-title:WS_2\n"
 ; !!"time:",time,"\t","dest:plot4\tline-type:3\tgraph-title:WS_3\n"
 ; *( a?(x, t); !!"time:",time,"\tdest:text4\tdata:",x,"\n"
              ; !!"time:",time,"\tdest:text7\tdata:",round(100*x/t)/100,"\n"
    | b?(y, t); !!"time:",time,"\tdest:text3\tdata:",y,"\n"
              ; !!"time:",time,"\tdest:text6\tdata:",round(100*y/t)/100,"\n"
    | c?z; !!"time:",time,"\tdest:plot3\tline-type:1\tpoints:[(",time,",", z, ")]\n"
    | d?; i:= i + 1; !!"time:",time,"\tdest:text9\tdata:",i,"\n"
    )
]|

proc env( chan a!: subord, b!, c?: ttote, d?: subord
            , e?: (nat,real), f?: void, g?: void ) =
|[ chan go2pm, pm2pr : subord
      , pm2go: (nat, real)
      , go2ex: (nat, real)
      , pr2gr: line
      , ex2viz0, ex2viz1: (nat, real), ex2viz2: real
      , pm2go2: void, pm2go3: void, go2ex2, gr2viz: void
```

```
:: GO ( pm2go2, go2ex, pm2go, go2pm, pm2go3, go2ex2 )
|| PM ( go2pm, a, d, pm2pr, pm2go, f, pm2go2, g, pm2go3 )
|| PR ( pm2pr, pr2gr )
|| GR ( pr2gr, b, gr2viz )
|| EX ( c, go2ex, e, go2ex2, ex2viz0, ex2viz1, ex2viz2 )
|| viz2 ( ex2viz0, ex2viz1, ex2viz2, gr2viz )
]|

////////////////////////////////////////////// MINILOAD ENVIRONMENT

proc GM( chan a!: subord                  // to PM
           , b?: subord                   // from PM
           , c!: (nat,nat)                // (qNo,mlNo)
           , d?: (line,bool)              // ((skuId,qty),prior)
           , e!: %NML # (nat,nat,nat,nat)// (skuId,qtyReq,subordId,subordSeq)
           , fIn?: %NML # (real,nat,nat) // (timeIn,skuId,toteQty)
           , fRes?: %NML # (real,nat)    // (newTimeIn,toteQty)
           , fOut?: %NML # nat           // retTotes
           , g?: void                    // leaving tote
           , h!: (subord, nat)           // (suborder, #totes)
           , i?: void                    // request
           , l!: void                    // add suborder
           , m!: void                    // start flowtime calculation
           , u!: nat                     // visualization
           , v!: %NML * nat              // visualization
           , w!: void                    // visualization
           , w2!: nat                    // visualization
           , w3!: (nat, nat)             // visualization
           , w4!: (real, real)           // visualization
       ) =
|[ var k: nat
     , p, pTemp: subord, ps: [subord] = []
     , r: nat = 0
     , totetimeIn: real, toteSkuId, toteQty, toteId: nat
     , calc: bool = false
     , zs: [(real,nat,nat)]
     , z: (real,nat,nat), z1,z2: nat
     , zsa: %NSKU * [(real,nat,nat)]
     , x: line, xs: [line] = [], xsP: [line] = []
     , ya: %NML * nat = yaInit()
     , yat: nat = yatInit()
     , yatin: nat = 0
     , wa: %NSKU * (%NML * nat)
     , MLno, retTotes, n: nat
     , prior: bool
     , pline: line
     , timeIn: real
     , MLdstr:-> nat = uniform(0,5*4*3*2*1)
     , send: bool = false, Ntote, np: nat = = (0,0)
     , start: bool = false
     , sublen: nat = 0, msublen: real = 0.0, varsublen: real = 0.0
:: zsa:= zsaInit()
 ; wa:= waInit()
 ; *( calc:= calc
     ; ( b?p; ps:= ps ++ [p]; send:= false; calc:= true
       | ( |, j <- 0..%NML-1, fIn.j?(totetimeIn,toteSkuId,toteQty)
```

```
        ; w3!(5, j)
        ; zsa.toteSkuId:= zsUpdate(zsa.toteSkuId,j,totetimeIn,toteQty)
        ; yatin:= yatin + 1
        ; w2!yatin
        ; calc:= true
        ; ( Ntote < %NTOTE
              -> Ntote:= Ntote + 1
               ; ( Ntote = %NTOTE -> start:= true; m!; w!
                 | Ntote /= %NTOTE -> skip
                 )
          | Ntote >= %NTOTE -> skip
          )
    )
  | i?; r:= r + 1; calc:= true
    ; w3!(9,0)
  | d?(x,prior)
    ; w3!(2,0)
    ; (     prior -> xsP:= xsP ++ [x]
      | not prior -> xs:= xs ++ [x]
     )
  | len(xsP) > 0
    -> x:= hd(xsP); xsP:= tl(xsP); k:= x.sku
     ; MLno:= MLassign(ya,wa.k,sample MLdstr)
     ; c!(1,MLno); ya.MLno:= ya.MLno + 1
     ; wa.k.MLno:= wa.k.MLno + x.qty
     ; v!ya
     ; w3!(1,MLno)
  | len(xsP) = 0 and len(xs) > 0 and yat < %NML * %NCELL
    -> x:= hd(xs); xs:= tl(xs)
     ; k:= x.sku
     ; MLno:= MLassign(ya,wa.k,sample MLdstr)
     ; c!(0,MLno); ya.MLno:= ya.MLno + 1
     ; wa.k.MLno:= wa.k.MLno + x.qty
     ; v!ya
     ; w3!(1,MLno)
     ; yat:= yat + 1; u!yat
  | ( |, j <- 0..%NML-1, fOut.j?retTotes; ya.j:= ya.j - retTotes
      ; yatin:= yatin - retTotes; w2!yatin; w3!(4,j)
    )
    ; v!ya
  | g?; yat := yat - 1; u!yat
    ; w3!(7,0)
  | start and not send and len(ps) < %NWIP -> l!; send:= true
  )
; ( not calc -> skip
  | calc -> (calc,p):= detSubOrder(take(ps,%NWIP),r,zsa)
    ; ( not calc -> skip
      | calc -> ps:= ps -- [p]; r:= r - 1
        ; n:= 0
        ; pTemp:= p
        ; len(pTemp.list) > 0
          *> ( pline:= hd(pTemp.list); pTemp.list:= tl(pTemp.list)
             ; zs:= zsa.(pline.sku)
             ; pline.qty > 0
               *> ( z:= hd(zs); zs:= tl(zs); z1:= z.1; z2:= z.2
                  ; e.z1!(pline.sku,pline.qty,p.id,p.seq)
```

```
                                ; w3!(3,z1)
                                ; fRes.z1?(timeIn,toteQty)
                                ; w3!(6,z1)
                                ; pline.qty:= pline.qty - ( pline.qty min toteQty )
                                ; wa.(pline.sku).z1:= wa.(pline.sku).z1 - toteQty
                                ; ( z2 > toteQty -> z:= ( timeIn,z.1,z2-toteQty )
                                                    ; zs:= insert(zs,z,incAge)
                                  | z2 = toteQty -> skip
                                  )
                                ; n:= n + 1
                              )
                          ; zsa.(pline.sku):= zs
                        )
                    ; h!(p, n)
                    ; w3!(8,0)
                    ; a!p
                    ; np:= np + 1
                    ; sublen:= len(p.list)
                    ; ( np > 1 -> varsublen:= varsublen * (np - 2)/(np - 1)
                                                + (1 / np) * (sublen - msublen)^2
                      | np <= 1 -> varsublen:= 0.0
                      )
                    ; msublen:= msublen * (np - 1) / np + sublen / np
                    ; msublen:= round(100*msublen)/100
                    ; varsublen:= round(100*varsublen)/100
                    ; w4!(msublen, varsublen)
                )
            )
        )
    ]|

    proc TMIn( chan a?: 2 # ttote, b!: ttote, c!: (line,bool)
                  , d?: (nat,nat) ) =
    |[ var x: ttote, xs: 2 * [ttote] = <[],[]>
        , qNo, mlNo: nat
        , toteToSend: ttote
    :: *( ( |, j <- 0..1, a.j?x; xs.j:= xs.j ++ [x]
          ; c!((x.tote.sku,x.tote.qty), j = 1) )
        | d?(qNo,mlNo); toteToSend:= hd(xs.qNo); xs.qNo:= tl(xs.qNo)
          ; toteToSend.des:= mlNo; b!toteToSend
        )
    ]|

    proc TMOut( chan a?, b!: ttote, c!: void ) =
    |[ var x: ttote
    :: *( a?x; b!x; ( x.tote.qty = x.req -> c!
                    | x.tote.qty > x.req -> skip
                    )
        )
    ]|

    proc viz( chan a?: nat, b?: 5*nat, c?: void, d?: nat
                 , e?:(real, real) ) =
    |[ var ya: 5*nat, yatin: nat, yat: nat = 0
         , av, t, t0, tstart: real
         , calc: bool = false
```

```
     , varsublen, msublen: real
:: *( a?yat
      ; !!"time:",time,"\tdest:plot2\tline-type:1\tpoints:
        [(",time,",", yat, ")]\n"
    | b?ya
      ; !!"time:",time,"\tdest:text1\tdata:"
        ,ya.0+ya.1+ya.2+ya.3+ya.4,"\n"
    | c?; calc:= true
        ; tstart:= time
    | d?yatin; !!"time:",time,"\tdest:text8\tdata:",yatin,"\n"
    | e?(msublen, varsublen)
      ; !!"time:",time,"\tdest:text10\tdata:",msublen,"\n"
      ; !!"time:",time,"\tdest:text11\tdata:",varsublen,"\n"
    )
|| *( !!"time:",time,"\tdest:plot2\tline-type:1\tpoints:
      [(",time,",", yat, ")]\n"
      ; delay 100.0
    )
]|

proc vizMSC( chan a?: (nat, nat) ) =
|[ var x, y: nat
     , i2: nat = 0, i7: nat = 0, i8: nat = 0, i9: nat = 0
:: !!"time:",time,"\tdest:msc1\tdata:START GM\n"
 ; !!"time:",time,"\tdest:msc1\tdata:START GW\n"
 ; !!"time:",time,"\tdest:msc1\tdata:START TMin\n"
 ; !!"time:",time,"\tdest:msc1\tdata:START TMout\n"
 ; !!"time:",time,"\tdest:msc1\tdata:START LM\n"
 ; *( a?(x, y)
      ; ( x = 1 -> !!"time:",time,"\tdest:msc1\tsend:GM\trecv:
                   TMin\tlabel:tote_to_ML_",y,"\n"
        | x = 2 -> i2:= (i2 + 1) mod 1000
          ; !!"time:",time,"\tdest:msc1\tsend:TMin\trecv:
                   GM\tlabel:arr_tote_",i2,"\n"
        | x = 3 -> !!"time:",time,"\tdest:msc1\tsend:GM\trecv:
                   LM\tlabel:retr_tote_ML_",y,"\n"
        | x = 4 -> !!"time:",time,"\tdest:msc1\tsend:LM\trecv:
                   GM\tlabel:4_totes_out_ML_",y,"\n"
        | x = 5 -> !!"time:",time,"\tdest:msc1\tsend:LM\trecv:
                   GM\tlabel:tote_in_ML_",y,"\n"
        | x = 6 -> !!"time:",time,"\tdest:msc1\tsend:LM\trecv:
                   GM\tlabel:tote_ass_ML_",y,"\n"
        | x = 7 -> i7:= (i7 + 1) mod 1000
          ; !!"time:",time,"\tdest:msc1\tsend:TMout\trecv:
              GM\tlabel:tote_",i7,"_to_WS\n"
        | x = 8 -> i8:= (i8 + 1) mod 1000
          ; !!"time:",time,"\tdest:msc1\tsend:GM\trecv:
              GW\tlabel:subord_",i8,"\n"
        | x = 9 -> i9:= (i9 + 1) mod 1000
          ; !!"time:",time,"\tdest:msc1\tsend:GW\trecv:
            GM\tlabel:request_",i9,"\n"
        )
    )
]|

proc MLEnv( chan a0!, a1?: subord
```

```
                , b?: 2 # ttote
                , c!: ttote
                , d?: ttote
                , e!: ttote
                , fIn?: %NML # (real,nat,nat)
                , fRes?: %NML # (real,nat)
                , fOut?: %NML # nat
                , g!: %NML # (nat,nat,nat,nat)
                , h!: (subord, nat)
                , i?: void
                , j!: void
                , k!: void
          ) =
|[ chan gm2TMIn: (nat,nat)
      , gm2viz0, gm2viz3: nat
      , gm2viz1: %NML*nat
      , gm2viz4: (nat, nat)
      , TMIn2gm: (line,bool)
      , TMOut2gm, gm2viz2: void
      , gm2viz5: (real, real)
:: GM ( a0, a1 , gm2TMIn, TMIn2gm, g, fIn, fRes, fOut, TMOut2gm
        , h, i, j, k, gm2viz0, gm2viz1, gm2viz2, gm2viz3, gm2viz4
        , gm2viz5
      )
|| TMIn ( b, c, TMIn2gm, gm2TMIn )
|| TMOut ( d, e, TMOut2gm )
|| viz ( gm2viz0, gm2viz1, gm2viz2, gm2viz3, gm2viz5 )
|| vizMSC( gm2viz4 )
]|

////////////////////////////////////////////// WORKSTATION ENVIRONMENT

proc GW( chan a!: void
            , b?: (subord, nat)
            , c!: (nat, nat, bool)
            , d?: ttote
            , e!: %NWS # (subord, nat)
            , f?: (nat, bool)
            , v!: nat
            , val maxTotes: nat
       ) =
|[ var x: ttote
     , p: subord
     , ps: [(subord, nat)]
     , k: nat, n: nat = 0, m: nat = 0, wip: nat = 0, i: nat = 0
     , ns: %NWS * nat = initNWS()
     , maxSubord: nat = 9
     , request: bool = true
     , wsId, des, sortId, j: nat
     , flag, new: bool
     , occ: %NWS * nat = initNWS()
     , ts: %NWS * [(subord, nat)] = inittsNWS()
:: *( wip < maxSubord and n < maxTotes and request
      -> a!; wip:= wip + 1; request:= false
    | b?(p, k); ps:= ps ++ [(p, k)]; n:= n + k; request:= true
    | d?x; m:= m + 1; v!m
```

```
      ; (new, k, p, ps):= newSubOrder(x, ps)
      ; (     new -> i:= i + 1
                     ; wsId:= detWSid(ns,occ)
                     ; ns.wsId:= ns.wsId + k
                     ; occ.wsId:= occ.wsId + 1
                     ; e.wsId!(p, i)
                     ; ts.wsId:= ts.wsId ++ [(p, i)]
        | not new -> skip
        )
      ; (ts, des, sortId, flag):= arrtote(ts, x)
      ; (     flag -> occ.des:= occ.des - 1
        | not flag -> skip
        )
      ; c!(des, sortId, flag)
    | f?(j, flag); n:= n - 1; m:= m - 1; v!m; ns.j:= ns.j - 1
      ; (     flag -> wip:= wip - 1
        | not flag -> skip
        )
    )
]|

proc TWIn( chan a?: ttote
             , c!: (ttote,nat,bool)
             , d!: ttote
             , e?: (nat,nat,bool)
             , var dt: real
         ) =
|[ var ys: [((ttote, nat, bool), real)] = []
     , x: ttote
     , flag: bool
     , wsid, sortId: nat
:: *( a?x; d!x; e?(wsid, sortId, flag); x.des:= wsid
      ; ys:= ys ++ [((x, sortId, flag),time+dt)]
    )
|| *( len(ys) > 0
      -> skip; delay ( hd(ys).1 - time ) max 0.0; c!hd(ys).0; ys:= tl(ys)
    )
]|

proc TWOut( chan a?: (ttote, nat, bool), b!, c!: ttote, d!: (nat, bool)
              , e!: (nat,real) ) =
|[ var x: ttote, sortId: nat, flag: bool
:: *( a?(x, sortId, flag)
      ; ( x.tote.qty > 0 -> b!x
        | x.tote.qty = 0 -> c!x
        )
      ; d!(x.src, flag)
      ; (     flag -> e!( x.seq, time-(2-x.src)*2.0 )
        | not flag -> skip
        )
    )
]|

proc viz1( chan a?: nat ) =
|[ var n: nat
:: *( a?n
```

```
      ; !!"time:",time,"\tdest:text2\tdata:",n,"\n"
    )
]|

proc WSEnv( chan a?: ttote, c!, d?: (ttote,nat,bool), e!, f!: ttote
                , g!: %NWS # (subord,nat), h!: void
                , i?: (subord, nat), j!: (nat, real)
                , val maxTotes: nat
          ) =
|[ chan gw2TWIn: (nat,nat,bool)
      , TWIn2gw: ttote
      , TWout2gw: (nat, bool)
      , gw2viz: nat
:: GW ( h, i, gw2TWIn, TWIn2gw, g, TWout2gw, gw2viz, maxTotes )
|| TWIn ( a, c, TWIn2gw, gw2TWIn, 5.0 )
|| TWOut ( d, e, f, TWout2gw, j )
|| viz1 ( gw2viz )
]|

//////////////////////////////////////////// DETAILED MINILOAD SYSTEM

proc LM (chan a!: (bool,[ttote])       // send job assignment to ML
            , b?: [ttote]              // stored totes from ML
            , cIn!: (real,nat,nat)     // update stored totes to GM
            , cRes!: (real,nat)        // update unreserved totes to GM
            , cOut!: nat               // update retrieved totes to GM
            , d?: (nat,nat,nat,nat)    // retrieve job assignment from GM
            , e?: ttote                // update totes in BI
            , f?: bool                 // update crane position
            , g?: nat                  // update no. of totes in ML
            , h?: void                 // update no. of totes in BO
            , i?: void                 // receive trigger for time out at ML
            , v!: 3*[ttote]            // visualization
            , val k: nat) =            // miniload index
|[ var skuV: %NSKU * [ptote]
         , x: ptote, ys: [ttote] = [], zs: [ttote] = [], z: ttote
         , sVec: %NSKU * nat, ms: [bool], m: nat, rs: [ttote], r: ttote
         , go: bool = false, calc: bool, crIn: bool = false
         , inputlist: [(real,nat,nat)] = [], outputlist: nat = 0
         , nBO: nat = 0, B: nat, ytake: nat, o: nat, os: [nat] = []
         , ya: 3*[ttote] = <[],[],[]>, y: ttote
         , row: nat = 0, skuNo, pId, pSeq, qtyReq: nat
:: skuV:= initskuV()
 ; sVec:= initNatVec()
 ; *( calc:= true
      ; ( d?(skuNo,qtyReq,pId,pSeq)
          ; x:= hd(skuV.skuNo); skuV.skuNo:= tl(skuV.skuNo)
          ; y:= (x, pId, pSeq, k, 99, x.qty min qtyReq)
          ; ya:= arrangeSubOrd(ya, y)
          ; v!ya
          ; sVec.skuNo:= sVec.skuNo - x.qty
          ; ( len(skuV.skuNo) > 0 -> cRes!(hd(skuV.skuNo).timeIn, x.qty)
            | len(skuV.skuNo) = 0 -> cRes!(0.0, x.qty)
            )
        | e?z; zs:= zs ++ [z]
        | g?m
```

```
          ; outputlist:= outputlist + m
          ; calc:= false
        | outputlist > 0
          -> cOut!outputlist
           ; outputlist:= 0
           ; calc:= false
        | b?rs
          ; len(rs) > 0
            *> ( r:= hd(rs); rs:= tl(rs)
                 ; skuV.(r.tote.sku)
                   := insert(skuV.(r.tote.sku), (r.tote), pred)
                 ; sVec.(r.tote.sku):= sVec.(r.tote.sku) + r.tote.qty
                 ; inputlist:= inputlist
                   ++ [(hd(skuV.(r.tote.sku)).timeIn, r.tote.sku
                       , r.tote.qty)]
               )
          ; calc:= false
        | len(inputlist) > 0
          -> cIn!hd(inputlist); inputlist:= tl(inputlist); calc:= false
        | f?crIn
        | h?; nBO:= nBO - 1
        | i?
          ; ( crIn
              -> a!(crIn, sendlist(ya, (B - nBO), row))
               ; (ya, ytake, row):= updateYA(ya, (B - nBO), row)
               ; v!ya
               ; nBO:= nBO + ytake
               ; go:= false
            | not crIn
              -> a!(crIn,take(zs,%NBATCH)); zs:= drop(zs,%NBATCH); go:= false
            )
        | go and crIn
          -> a!(crIn, sendlist(ya, (B - nBO), row))
           ; (ya, ytake, row):= updateYA(ya, (B - nBO), row)
           ; v!ya
           ; nBO:= nBO + ytake
           ; go:= false
        | go and not crIn
          -> a!(crIn,take(zs,%NBATCH)); zs:= drop(zs,%NBATCH); go:= false
        )
      ; (     calc -> go:= inOut(len(zs),(len(ya.0)+len(ya.1)+len(ya.2)))
        | not calc -> skip
        )
    )
]|

proc TI (chan a?: ttote, b!: ttote, c!: ttote, val k: nat) =
|[ var x: ttote, xs: [(ttote,real)] = [], t: real
:: *( a?x
      ; t:= time + 1.0; xs:= xs ++ [(x,t)]
      ; len(xs) > 0
        *> ( a?x
             ; t:= time + 1.0; xs:= xs ++ [(x,t)]
           | delay ((hd(xs).1 - time) max 0.0)
             ; ( hd(xs).0.des  = k -> b!hd(xs).0
               | hd(xs).0.des /= k -> c!hd(xs).0
```

```
              )
            ; xs:= tl(xs)
          )
    )
]|

proc BI (chan a?: ttote, b!: [ttote], c!: ttote, val k: nat) =
|[ var xs: [ttote], x: ttote
:: *( a?x; xs:= xs ++ [x]; c!x
    | b!take(xs,%NBATCH) ; xs:= drop(xs,%NBATCH)
    )
]|

proc ML (chan a?: (bool,[ttote]), b?: [ttote], c!: [ttote], d!: bool
           , e!: nat, f!: [ttote], g!: void, v!: bool
           , val k: nat) =
|[ var ps: [ttote], p: ttote, xs: [ttote]
      , crIn: bool = false, t:-> real = constant(120.0)
      , retrieve: bool, timeout: real = 120.0
:: *( a?(retrieve,ps)
    ; v!false
    ; ( retrieve
        -> skip
         ; ( len(ps) = 0 -> skip; delay 23.6
           | len(ps) > 0 -> skip; delay ( 16.5 + ((len(ps) - 1) * 7.0 ) )
           )
         ; crIn:= false; d!crIn
         ; c!ps
         ; e!len(ps)
         ; timeout:= time + sample t
      | not retrieve
        -> b?xs
         ; ( len(xs) = 0 -> skip; delay 23.6
           | len(xs) = 1 or len(xs) = 2 -> skip; delay 30.6
           | len(xs) = 3 or len(xs) = 4 -> skip; delay 37.5
           )
         ; crIn:= true; d!crIn
         ; f!xs
         ; timeout:= time + sample t
    )
    ; v!true
  | delay ((timeout - time) max 0.0)
    ; g!
    ; a?(retrieve,ps)
    ; v!false
    ; ( retrieve
          -> skip
           ; ( len(ps) = 0 -> skip; delay 23.6
             | len(ps) > 0 -> skip; delay ( 16.5 + ((len(ps) - 1) * 7 ) )
             )
           ; crIn:= false; d!crIn
           ; c!ps
           ; e!len(ps)
           ; timeout:= time + sample t
      | not retrieve
          -> b?xs
```

```
            ; ( len(xs) = 0 -> skip; delay 23.6
              | len(xs) = 1 or len(xs) = 2 -> skip; delay 30.6
              | len(xs) = 3 or len(xs) = 4 -> skip; delay 37.5
              )
          ; crIn:= true; d!crIn
          ; f!xs
          ; timeout:= time + sample t
        )
      ; v!true
    )
]|

proc BO (chan a?: [ttote], b!: ttote, c!: void, val k: nat) =
|[ var ys: [ttote], xs: [ttote] = []
:: *( a?ys; xs:= xs ++ ys
    | len(xs) > 0
     -> b!hd(xs); xs:= tl(xs)
       ; c!
    )
]|

proc TO (chan a?: ttote, b!: ttote, c?: ttote, val k: nat) =
|[ var x: ttote, n,i: nat, ws: bool, priority: bool = false
     , j: nat = 0, xs: [(ttote,real)] = [], t: real
:: *( a?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
      ; len(xs) > 0
        *> ( a?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
           | c?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
           | delay ((hd(xs).1 - time) max 0.0)
             ; b!hd(xs).0; xs:= tl(xs)
           )
    | c?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
      ; len(xs) > 0
        *> ( a?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
           | c?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
           | delay ((hd(xs).1 - time) max 0.0)
             ; b!hd(xs).0; xs:= tl(xs)
           )
    )
]|

proc viz3( chan a?: bool, b?: 3*[ttote], val k: nat )=
|[ var t: real = 0.0, s: bool, u: real = 0.0, cur: real
     , ya: 3*[ttote], tstart, tcur, av: real, first: bool = true, num: nat
     , yam: (real,real,nat) = (0.0,0.0,0)
:: *( a?s; cur:= time
      ; ( cur = 0.0 -> skip
        | cur > 0.0 -> skip
          ; ( not s -> u:= t / cur * u
            | s -> u:= t / cur * u + (cur - t) / cur
            )
          ; t:= cur
        )
      ; !!"time:",time,"\tdest:plot1\tline-type:",k+1,"\tpoints:[(",cur,",", u, ")]\n"
    | b?ya; num:= len(ya.0) + len(ya.1) + len(ya.2)
      ; !!"time:",time,"\tdest:bar1\tcur_",k,":",num,"\n"
```

```
      ; tcur:= time
      ; (     first -> tstart:= tcur; first:= false
        | not first -> av:= ( yam.0 * (yam.1 - tstart) + yam.2 * (tcur - yam.1) )
                             / (tcur - tstart)
                     ; !!"time:",time,"\tdest:bar1\tavg_",k,":",av,"\n"
        )
      ; yam:= (av, time, num)
    )
]|

proc MLS (chan a?, b!: ttote, cIn!: (real,nat,nat), cRes!: (real,nat)
             , cOut!: nat, d?: (nat,nat,nat,nat), e!: [ttote]
             , f!: [ttote], val k: nat, te: real, tr: real) =
|[ chan lm2ml: (bool,[ttote]), ml2lm: [ttote], bi2lm: ttote
      , ti2bi: ttote, bi2ml: [ttote], ml2bo: [ttote]
      , ml2lm2: bool, ml2lm3: nat, bo2to: ttote, ti2to: ttote, ml2util: bool
      , bo2lm: void, ml2lm4: void
      , lm2viz: 3*[ttote]
:: LM(lm2ml,ml2lm,cIn,cRes,cOut,d,bi2lm,ml2lm2,ml2lm3,bo2lm,ml2lm4,lm2viz,k)
|| BI(ti2bi,bi2ml,bi2lm,k)
|| ML(lm2ml,bi2ml,ml2bo,ml2lm2,ml2lm3,ml2lm,ml2lm4,ml2util,k)
|| BO(ml2bo,bo2to,bo2lm,k)
|| TO(bo2to,b,ti2to,k)
|| TI(a,ti2bi,ti2to,k)
|| viz3(ml2util,lm2viz,k)
]|

proc MLClus( chan a?, b!: ttote, cIn!: %NML # (real,nat,nat)
                  , cRes!: %NML # (real,nat), cOut!: %NML # nat
                  , d?: %NML # (nat,nat,nat,nat) ) =
|[ chan mm: %NML-1 # ttote, ti2vis: %NML # [ttote], to2vis: %NML # [ttote]
:: MLS (    a, mm.0, cIn.0, cRes.0, cOut.0, d.0, ti2vis.0, to2vis.0, 0, 3.0, 1.0 )
|| ( || , j <- 0..%NML-3, MLS(mm.j, mm.(j+1), cIn.(j+1), cRes.(j+1), cOut.(j+1)
     , d.(j+1), ti2vis.(j+1), to2vis.(j+1), j+1, 3.0, 1.0))
|| MLS ( mm.(%NML-2),    b, cIn.(%NML-1), cRes.(%NML-1), cOut.(%NML-1)
         , d.(%NML-1), ti2vis.(%NML-1), to2vis.(%NML-1), (%NML-1), 3.0, 1.0 )
]|


////////////////////////////////////// DETAILED WORKSTATION SYSTEM

proc LW(chan a?: (subord, nat), b!: nat, d?: (nat, nat), c?: (nat, bool)
           , v!: (field, nat, [nat]), val WSID: nat) =
|[ var ps: [(subord, nat)] = [], p: (subord, nat)
     , fld: field = <[], [], []>, xs: [nat] = [], x: nat
     , top: 3 * nat = <0, 0, 0>
     , bottom: 3 * nat = <0, 0, 0>
     , length: 3 * nat = <0, 0, 0>
     , arriving: [nat] = []
     , curP: subord = (0, 0, [])
     , currId, skuId, n, lane: nat
     , nTotes: nat = 0
     , flag: bool
:: *( a?p; ps:= ps ++ [p]; arriving:= arriving ++ [p.1]
    | len(curP.list) = 0 and len(ps) > 0 -> (curP, currId):= hd(ps); ps:= tl(ps)
    | len(curP.list) > 0 and nTotes < 1 and (or, j <- 0..2, bottom.j = currId )
```

```
      -> lane:= detOutLane(bottom, length, currId)
       ; xs:= xs ++ [hd(fld.lane)]; fld.lane:= tl(fld.lane)
       ; length.lane:= length.lane - 1
       ; ( length.lane  = 0 -> top.lane:= 0
         | length.lane /= 0 -> skip
         )
       ; bottom:= updBottom(bottom, fld); nTotes:= nTotes + 1
       ; v!(fld, lane, arriving)
    | len(xs) > 0 -> b!hd(xs); xs:= tl(xs)
    | c?(x, flag); lane:= detInLane(top, length, arriving, x)
      ; fld.lane:= fld.lane ++ [x]; length.lane:= length.lane + 1
      ; top.lane:= x
      ; ( not flag -> skip
        |     flag -> arriving:= updArriving(arriving, x)
        )
      ; bottom:= updBottom(bottom, fld)
      ; v!(fld, lane, arriving)
    | d?(skuId, n); nTotes:= nTotes - 1; curP:= updActiveorder(curP, skuId, n)
    )
]|

proc BW(chan a?: (ttote, nat, bool), b!: (ttote, nat, bool)
           , c!: (nat, bool), d?: nat) =
|[ var xs: [(ttote, nat, bool)] = [], x: (ttote, nat, bool)
     , sortId: nat
:: *( a?x; xs:= xs ++ [x]; c!(x.1, x.2)
    | d?sortId; (xs, x):= findTTote(xs, sortId); b!x
    )
]|

proc MW(chan a?, b!: (ttote, nat, bool), c!: (nat, nat), v!: bool
           , val id: nat) =
|[ var t: -> real = triangle (9.5, 12.5, 15.5 )
     , x: (ttote, nat, bool)
     , s: real
:: *( a?x; v!false; s:= sample t; delay s
      ; x.0.tote.qty:= x.0.tote.qty - x.0.req
      ; x.0.src:= id
      ; b!x
      ; c!(x.0.tote.sku, x.0.req)
      ; v!true
    )
]|

proc BWOut(chan a?, b!: (ttote, nat, bool) ) =
|[ var xs: [(ttote, nat, bool)] = [], x: (ttote, nat, bool)
:: *( a?x; xs:= xs ++ [x]
    | len(xs) > 0  -> b!hd(xs); xs:= tl(xs)
    )
]|

proc Tdiv(chan a?, b!, c!: (ttote, nat, bool), val id: nat) =
|[ var xs: [((ttote, nat, bool), real)] = []
     , x: (ttote, nat, bool)
     , t: real
:: *( a?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
```

```
      ; len(xs) > 0
        *> ( a?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
           | delay (hd(xs).1 - time) max 0.0
             ; ( hd(xs).0.0.des = id -> c!hd(xs).0
               | hd(xs).0.0.des /= id -> b!hd(xs).0
               )
             ; xs:= tl(xs)
           )
    )
]|

proc Tmer(chan a?, b?, c!: (ttote, nat, bool), val id: nat) =
|[ var xs: [((ttote, nat, bool), real)]
     , x: (ttote, nat, bool)
     , t: real
:: *( a?x; x.0.src:= id; x.0.des:= 99; t:= time + 1.0; xs:= xs ++ [(x, t)]
      ; len(xs) > 0
        *> ( a?x; x.0.src:= id; x.0.des:= 99; t:= time + 1.0; xs:= xs ++ [(x, t)]
           | b?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
           | delay (hd(xs).1-time) max 0.0; c!hd(xs).0; xs:= tl(xs)
           )
    | b?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
      ; len(xs) > 0
        *> ( a?x; x.0.src:= id; x.0.des:= 99; t:= time + 1.0; xs:= xs ++ [(x, t)]
           | b?x; t:= time + 1.0; xs:= xs ++ [(x,t)]
           | delay (hd(xs).1-time) max 0.0; c!hd(xs).0; xs:= tl(xs)
           )
    )
]|

proc viz4( chan a?: bool, b?: (field, nat, [nat]), val k: nat )=
|[ var t: real = 0.0, av, cur, tstarta: real, s: bool, u: real = 0.0
     , lane: nat, fld: field, arriving: [nat]
     , ava: 3 * (real, real, nat)
     , calc: 3*bool = <true, true, true>, first: bool = true
     , t0, tstartb: 3*real
:: *( a?s; cur:= time
      ; (     first -> tstarta:= cur; first:= false
        | not first -> skip
          ; ( not s -> u:= u * (t - tstarta) / (cur - tstarta)
            | s -> u:= (u * (t - tstarta) + (cur - t)) / (cur - tstarta)
            )
          ; !!"time:",time,"\tdest:plot4\tline-type:",k+1
            ,"\tpoints:[(",cur,",", u, ")]\n"
        )
      ; t:= cur
    | b?(fld, lane, arriving)
      ; t0.lane:= time
      ; !!"time:",time,"\tdest:buf",k,"\t",writeColumn(0, fld.0, arriving),"\n"
      ; !!"time:",time,"\tdest:buf",k,"\t",writeColumn(1, fld.1, arriving),"\n"
      ; !!"time:",time,"\tdest:buf",k,"\t",writeColumn(2, fld.2, arriving),"\n"
      ; !!"time:",time,"\tdest:bar",k+2,"\tcur_",lane,":",len(fld.lane),"\n"
      ; (     calc.lane -> tstartb.lane:= t0.lane; calc.lane:= false
        | not calc.lane
          -> av:= ( ava.lane.0 * (ava.lane.1 - tstartb.lane)
                    + ava.lane.2 * (t0.lane-ava.lane.1) )
```

```
                    / (t0.lane - tstartb.lane)
            ; !!"time:",time,"\tdest:bar",k+2,"\tavg_",lane,":",av,"\n"
          )
        ; ava.lane:= (av, t0.lane, len(fld.lane))
      )
]|

proc WS(chan a?: (subord, nat), b?, c!: (ttote, nat, bool), val Id: nat) =
|[ chan BW2MW, MW2BWOut, BWOut2Tmer, Tdiv2Tmer, Tdiv2BW: (ttote, nat, bool)
       , BW2LW: (nat, bool)
       , LW2BW: nat
       , MW2LW: (nat, nat)
       , mw2viz: bool
       , lw2viz: (field, nat, [nat])
:: Tdiv(b, Tdiv2Tmer, Tdiv2BW, Id)
|| BW(Tdiv2BW, BW2MW, BW2LW, LW2BW)
|| LW(a, LW2BW, MW2LW, BW2LW, lw2viz, Id)
|| MW(BW2MW, MW2BWOut, MW2LW, mw2viz, Id)
|| BWOut(MW2BWOut, BWOut2Tmer)
|| Tmer(BWOut2Tmer, Tdiv2Tmer, c, Id)
|| viz4(mw2viz, lw2viz, Id)
]|

proc WSClus( chan a?, b!: (ttote, nat, bool), c?: %NWS # (subord,nat) ) =
|[ chan WS02WS1, WS12WS2: (ttote, nat, bool)
:: WS(c.0, a,       WS02WS1, 0)
|| WS(c.1, WS02WS1, WS12WS2, 1)
|| WS(c.2, WS12WS2, b      , 2)
]|
```

χ functions

```
////////////////////////////////////////////// ENVIRONMENT FUNCTIONS

func initls() -> [(nat,real)] = { search.py } :: initls

func initskus() -> [(nat,nat,real)] = { search.py } :: initskus

func searchSKU(val r: real, y: [(nat,nat,real)])
-> nat = { search.py } :: searchSKU

func numreq(val q: nat, k: nat, skuvec: [(nat,nat,real)])
-> nat = { search.py } :: numreq

func updatesku(val k,n: nat, y: [(nat,nat,real)])
-> [(nat,nat,real)] = { search.py } :: updatesku

func totalN(val skus: [(nat,nat,real)])
-> nat = { search.py } :: totalN

func refreshsku(val skuvec: [(nat,nat,real)], xs:[(nat,nat,real)])
-> ([(nat,nat,real)],nat) = { search.py } :: refreshsku
```

```
func incrSKU( val a, b: (nat,nat,real) ) -> bool = |[ ret a.0 <= b.0 ]|

func numLines(val m: real) -> nat =
|[ var p: nat, q: real, ls: [(nat,real)] = initls(), i: nat = 0
:: i <= len(ls)
   *> ( (p,q):= hd(ls)
        ; ( m > q  -> skip
            | m <= q -> ret p
          )
        ; ls:= tl(ls)
      )
]|

func list2vec(val xs: [nat]) -> %NSKU * nat =
|[ var vec: %NSKU * nat, i: nat = 0
:: len(xs) > 0
     *> ( vec.i:= hd(xs); xs:= tl(xs); i:= i + 1 )
 ; ret vec
]|

func initIP() -> [nat] = { search.py } :: initIP

func initMin() -> [nat] = { search.py } :: initMin

func initMax() -> [nat] = { search.py } :: initMax

func inittote() -> [nat] = { search.py } :: inittote


func detFlowTime( val ps: [(nat, real)], i: nat )
-> ( real, [(nat,real)] ) =
|[ var pst: [(nat,real)] = ps, p: (nat,real)
     , qs: [(nat,real)] = []
     , j: real
:: len(pst) > 0
   *> ( p := hd(pst); pst:= tl(pst)
        ; ( p.0 = i  -> j:= p.1
          | p.0 /= i -> qs:= qs ++ [p]
          )
      )
; ret (j, qs)
]|

////////////////////////////// MINILOAD ENVIRONMENT FUNCTIONS

func zsaSum( val zsa: %NSKU * [(real,nat,nat)] ) -> %NSKU * nat =
|[ var zsasum: %NSKU * nat
     , i: nat = 0
:: i < %NSKU
   *> ( zsasum.i:= subLineSum(zsa.i); i:= i + 1 )
 ; ret zsasum
]|

func detSubOrder( val ps: [subord], r: nat, zsa: %NSKU * [(real,nat,nat)] )
-> (bool,subord) =
|[ var p: subord
```

```
    , pst: [subord] = ps
    , found: bool = false
    , available: bool
:: len(pst) > 0 and r > 0
   *> ( p:= hd(pst); pst:= tl(pst)
        ; available:= detSubOrderAvailable(p,zsa)
        ; ( available -> ret(true,p) | not available -> skip )
      )
 ; ret(false,(0,0,[(0,0)]))
]|

func detSubOrderAvailable( val p: subord, zsa: %NSKU * [(real,nat,nat)] )
-> bool =
|[ var available: bool = true
     , plist: [line] = p.list
     , pline: line
     , itemNo: nat
:: len(plist) > 0
   *> ( pline:= hd(plist); plist:= tl(plist)
        ; itemNo:= subLineSum(zsa.(pline.sku))
        ; ( itemNo >= pline.qty -> skip
          | itemNo <  pline.qty -> ret false
          )
      )
 ; ret true
]|

func subLineSum( val subLine: [(real,nat,nat)] ) -> nat =
|[ var subLineTemp: [(real,nat,nat)] = subLine
     , sum: nat = 0
:: len(subLineTemp) > 0
   *> ( sum:= sum + hd(subLineTemp).2; subLineTemp:= tl(subLineTemp) )
 ; ret sum
]|

func zsUpdate( val zs: [(real,nat,nat)], j: nat, timeIn: real, qty: nat )
-> [(real,nat,nat)] =
|[ var ps, qs: [(real,nat,nat)]
    , newtimeIn: real, newQty: nat
:: (ps,qs):= extract(zs,j)
 ; ( len(ps) = 0 -> qs:= insert(qs,(timeIn,j,qty),incAge)
   | len(ps) = 1 -> newtimeIn:= timeIn min hd(ps).0
     ; newQty:= hd(ps).2 + qty
     ; qs:= insert(qs,(newtimeIn,j,newQty),incAge)
   )
; ret qs
]|

func extract(val zs: [(real,nat,nat)], j: nat )
-> ([(real,nat,nat)],[(real,nat,nat)]) =
|[ var ps: [(real,nat,nat)] = zs
     , p: (real,nat,nat)
:: len(ps) > 0
   *> ( p:= hd(ps); ps:= tl(ps)
 ; ( p.1  = j -> ret([p],zs -- [p]) | p.1 /= j -> skip ) )
 ; ret([],zs)
```

```
]|

func incAge( val a, b: (real,nat,nat) ) -> bool = |[ ret a.0 <= b.0 ]|

func MLassign( val ya: %NML * nat, wa: %NML * nat, x: nat ) -> nat =
|[ var j: nat = 0
    , xs: [(nat,nat)] = []
:: j < %NML
   *> ( ( ya.j <  %NCELL -> xs:= MLList(j,wa.j,xs)
        | ya.j >= %NCELL -> skip
        )
        ; j:= j + 1
      )
 ; ret hd(drop(xs,x mod len(xs))).0
]|

func MLList( val j: nat, w : nat, xs: [(nat,nat)] ) -> [(nat,nat)] =
|[ var ys : [(nat,nat)]
:: ( len(xs) = 0 -> ys := [(j,w)]
   | len(xs) > 0 -> skip
     ; ( w < hd(xs).1 -> ys := [(j,w)]
       | w = hd(xs).1 -> ys := xs ++ [(j,w)]
       | w > hd(xs).1 -> ys := xs
       )
   )
 ; ret ys
]|

func zsaInit() -> %NSKU * [(real,nat,nat)] =
|[ var vec: %NSKU * [(real,nat,nat)], i: nat = 0
:: i < %NSKU
   *> ( vec.i:= []; i:= i + 1 )
 ; ret vec
]|

func yaInit() -> %NML * nat =
|[ var i: nat = 0, vec: %NML * nat
:: i < %NML
   *> ( vec.i:= 0; i:= i + 1 )
 ; ret vec
]|

func yatInit() -> nat =
|[ ret 0 ]|

func waInit() -> %NSKU * (%NML * nat) =
|[ var wa: %NSKU * (%NML * nat)
    , i: nat = 0, j: nat = 0
:: i < %NSKU
   *> ( j < %NML *> ( wa.i.j:= 0; j:= j + 1 ); i:= i + 1; j:= 0 )
; ret wa
]|

////////////////////////////////////////////// WORKSTATION ENVIRONMENT FUNCTIONS

func detNtot (val nw: 3 * nat ) -> nat =
```

```
|[ var i: nat = 0, Ntot: nat = 0
:: i < 3
   *> ( Ntot:= Ntot + nw.i; i:= i + 1 )
 ; ret Ntot
]|

func wslistInit( val maxwip: nat ) -> [nat] =
|[ var i: nat = 0, wslist: [nat] = []
:: i < maxwip *> ( wslist:= wslist ++ [i mod 3]; i:= i + 1 )
 ; ret wslist
]|

func arrtote(val ts: 3 * [(subord, nat)], x: ttote)
     -> (3 * [(subord, nat)], nat, nat, bool) =
|[ var ps: 3 * [(subord, nat)] = ts
    , qs: [(subord, nat)] = []
    , p: subord
    , flag: bool = false
    , found: bool = false
    , i: nat = 0
    , sortId: nat
:: i < 3
   *> ( len(ps.i) > 0
        *> ( p:= hd(ps.i).0; sortId:= hd(ps.i).1; ps.i:= tl(ps.i)
             ; ( x.ord /= p.id or  x.seq /= p.seq -> qs:= qs ++ [(p, sortId)]
               | x.ord  = p.id and x.seq  = p.seq -> skip
                 ; p:= updActiveorder(p, x.tote.sku, x.req)
                 ; found:= true
                 ; ( len(p.list) = 0 -> flag:= true
                   | len(p.list) > 0 -> qs:= qs ++ [(p, sortId)]
                   )
                 ; qs:= qs ++ ps.i; ps.i:= []
               )
          )
        ; ps.i:= qs; qs:= []
        ; ( not found -> i:= i + 1
          |     found -> ret(ps, i, sortId, flag)
          )
      )
; !!"Error in ArrTote, did not find tote in order\t",x,"\n"
]|

func updActiveorder(val s: subord, skuid, n: nat) -> subord =
|[ var p: subord = s
     , q: subord = (s.id, s.seq, [])
     , r: line
:: len(p.list) > 0
   *> ( r:= hd(p.list); p.list:= tl(p.list)
        ; ( r.sku /= skuid -> q.list:= q.list ++ [r]
          | r.sku  = skuid -> skip
            ; ( n <  r.qty -> r.qty:= r.qty - n; q.list:= q.list ++ [r]
              | n >= r.qty -> skip
              )
            ; q.list:= q.list ++ p.list; ret q
          )
      )
```

```
]|

func newSubOrder( val x: ttote, ps: [(subord, nat)] )
-> (bool, nat, subord, [(subord, nat)]) =
|[ var psT: [(subord, nat)] = ps, p: (subord, nat)
     , qs: [(subord, nat)] = [], q: (subord, nat)
     , new: bool = false
:: len(psT) > 0
   *> ( p:= hd(psT); psT:= tl(psT)
        ; ( x.seq  = p.0.seq -> q:= p; new:= true
          | x.seq /= p.0.seq -> qs:= qs ++ [p]
          )
      )
 ; ret (new, q.1, q.0, qs)
]|

func detWSid( val ns, occ: 3*nat ) -> nat =
|[ var ids: [(nat, nat)] = [], i: nat = 0
:: i < 3
   *> ( ( occ.i <  3 -> ids:= ids ++ [(i,ns.i)]
        | occ.i >= 3 -> skip
        )
        ; i:= i + 1
      )
 ; ids:= sort(ids,inc1)
 ; ret(hd(ids).0)
]|

////////////////////////////////////////////// MINILOAD SYSTEM FUNCTIONS

func pToteIncAge( val a, b: ptote ) -> bool = |[ ret a.timeIn <= b.timeIn ]|

func initskuV() -> %NSKU * [ptote] =
|[ var vec: %NSKU * [ptote], i: nat = 0
:: i < %NSKU
   *> ( vec.i:= []; i:= i + 1 )
 ; ret vec
]|

func initNatVec() -> %NSKU * nat =
|[ var vec: %NSKU * nat, i: nat = 0
:: i < %NSKU *> ( vec.i:= 0; i:= i + 1 )
 ; ret vec
]|

func pred(val x,y: ptote) -> bool = |[ ret x.timeIn <= y.timeIn ]|

func inOut(val x: nat, y: nat) -> bool =
|[ ( ret x >= %NBATCH or y >= %NBATCH ) ]|

func inct(val x, y: (ttote, real)) -> bool = |[ ret x.1 < y.1 ]|

func arrangeSubOrd(val arr: 3*[ttote], y: ttote) -> 3*[ttote] =
|[ var i: nat = 0
     , ya: 3 * [ttote] = arr, x: ttote
     , ls: [(nat, nat)] = []
```

```
     , k: nat
:: i < 3
   *> ( len(ya.i) > 0
        -> x:= hr(ya.i); ls:= ls ++ [(i, len(ya.i))]
         ; ( y.ord  = x.ord and y.seq  = x.seq -> ya.i:= ya.i ++ [y]; ret ya
           | y.ord /= x.ord and y.seq /= x.seq -> i:= i +1
           )
      | len(ya.i) = 0 -> ls:= ls ++ [(i, len(ya.i))]; i:= i + 1
      )
 ; ls:= sort(ls, inc1); k:= hd(ls).0
 ; ya.k:= ya.k ++ [y]; ret ya
]|

func sendlist(val arr: 3*[ttote], nfree: nat, row: nat ) -> [ttote] =
|[ var ya: 3 * [ttote] = arr
     , ys: [ttote] = []
     , r: nat = row
     , ytake: nat
:: ytake:= (%NBATCH min nfree) min (len(ya.0)+len(ya.1)+len(ya.2))
 ; len(ys) < ytake
   *> ( ( len(ya.r) > 0 -> ys:= ys ++ [hd(ya.r)]; ya.r:= tl(ya.r)
        | len(ya.r) = 0 -> skip
        )
      ; r:= (r + 1) mod 3
      )
 ; ret ys
]|

func updateYA(val arr: 3*[ttote], nfree: nat, row: nat )
              -> (3*[ttote], nat, nat) =
|[ var ya: 3 * [ttote] = arr
     , ys: [ttote] = []
     , r: nat = row
     , ytake: nat
:: ytake:= (%NBATCH min nfree) min (len(ya.0)+len(ya.1)+len(ya.2))
 ; len(ys) < ytake
   *> ( ( len(ya.r) > 0 -> ys:= ys ++ [hd(ya.r)]; ya.r:= tl(ya.r)
        | len(ya.r) = 0 -> skip
        )
      ; r:= (r + 1) mod 3
      )
 ; ret (ya, ytake, r)
]|

////////////////////////////////////////////// WORKSTATION SYSTEM FUNCTIONS

func incSortId( val a, b: (ttote,nat,bool) ) -> bool = |[ ret a.1 <= b.1 ]|

func inc( val a, b: nat ) -> bool = |[ ret a <= b ]|

func qPsUpdatePlus( val q: (nat,nat,bool), ps: [(nat,nat,bool)], i: nat, j: bool )
     -> ((nat,nat,bool), [(nat,nat,bool)]) =
|[ var pst: [(nat,nat,bool)] = ps
     , qs: [(nat,nat,bool)] = []
     , p: (nat,nat,bool)
:: ( i  = q.0 -> ret ((i, q.1 + 1, j), ps)
```

```
   | i /= q.0 -> skip
     ; len(pst) > 0
       *> ( p:= hd(pst); pst:= tl(pst)
            ; ( i  = p.0 -> qs:= qs ++ [(i, p.1 + 1, j)]
              | i /= p.0 -> qs:= qs ++ [p]
              )
          )
     ; (q, qs):= qPsUpdate(q, qs)
     ; ret (q, qs)
   )
]|

func qPsUpdate( val q: (nat,nat,bool), ps: [(nat,nat,bool)] )
     -> ((nat,nat,bool), [(nat,nat,bool)] ) =
|[ ( q.1 > 0 or not q.2 -> ret (q, ps)
   | q.1 = 0 and q.2 -> skip
     ; ( len(ps) > 0 -> ret (hd(ps), tl(ps))
       | len(ps) = 0  -> ret (q, ps)
       )
   )
]|

func detOutLane(val bottom, length: 3 * nat, currid: nat) -> nat =
|[ var b: 3 * nat = bottom, i: nat = 0
     , rs: [(nat, nat)] = []
:: i < 3
   *> ( ( b.i  = currid -> rs:= rs ++ [(i, length.i)]
        | b.i /= currid -> skip
        )
        ; i:= i + 1
      )
 ; ret hd(sort(rs,dec1)).0
]|

func updBottom(val bottom: 3 * nat, f: 3 * [nat]) -> 3 * nat =
|[ var b: 3 * nat = bottom, i: nat = 0
:: i < 3
   *> ( ( len(f.i) > 0 -> b.i:= hd(f.i)
        | len(f.i) = 0 -> b.i:= 0
        )
        ; i:= i + 1
      )
 ; ret b
]|

func writeColumn(val lane: nat, col, arriving: [nat]) -> string =
|[ var s: string = ""
     , as: [nat] = arriving
     , cs: [nat] = col
     , c: nat
     , i: nat = 0
     , n: 3*nat = <0, 0, 0>
:: i < 3
   *> ( ( len(as) > 0 -> n.i:= hd(as); as:= tl(as)
        | len(as) = 0 -> skip
        )
```

```
        ; i:= i + 1
      )
 ; s:= "col_" ++ n2s(lane) ++ ":["
 ; len(cs) > 0
   *> ( c:= hd(cs); cs:= tl(cs)
        ; s:= s ++ "('" ++ n2s(c)
        ; ( c = n.0 -> s:= s ++ "',0)"
          | c = n.1 -> s:= s ++ "',1)"
          | c = n.2 -> s:= s ++ "',2)"
          | c /= n.0 and c /= n.1 and c /= n.2 -> s:= s ++ "',3)"
          )
        ; ( len(cs) > 0 -> s:= s ++ ","
          | len(cs) = 0 -> skip
          )
      )
 ; ret s ++ "]"
]|

func detInLane(val top, length: 3 * nat, arriving: [nat], x: nat) -> nat =
|[ var i: nat = 0
     , n: nat = 0
     , NLane: nat = 3
     , arr: [nat] = arriving
     , se: [(nat, nat)] = []
     , sg: [(nat, nat, nat)] = []
:: len(arr) > 0
   *> ( x  = hd(arr) -> arr:= []
      | x /= hd(arr) -> n:= n + 1; arr:= tl(arr)
      )
 ; i < NLane
   *> ( ( x < top.i -> skip
        | x = top.i -> se:= se ++ [(i, length.i)]
        | x > top.i -> sg:= sg ++ [(i, x - top.i, length.i)]
        )
        ; i:= i + 1
      )
 ; ( len(sg) <= n -> skip
   | len(sg) >  n -> i:= hd(sort(sg, inc12)).0
     ; se:= se ++ [(i, length.i)]
   )
 ; ret hd(sort(se, inc1)).0
]|

func dec1(val x, y: (nat, nat)) -> bool =
|[ ( x.1 /= y.1 -> ret x.1 > y.1
   | x.1  = y.1 -> ret x.0 < y.0
   )
]|

func inc12(val x, y: (nat, nat, nat)) -> bool =
|[ ( x.1 /= y.1 -> ret x.1 < y.1
   | x.1  = y.1 -> skip
     ; ( x.2 /= y.2 -> ret x.2 < y.2
       | x.2  = y.2 -> ret x.0 < y.0
       )
   )
```

```
]|

func inc1(val x, y: (nat, nat)) -> bool =
|[ ( x.1 /= y.1 -> ret x.1 < y.1
   | x.1  = y.1 -> ret x.0 < y.0
   )
]|

func updArriving(val arriving: [nat], x: nat) -> [nat] =
|[ var as: [nat] = arriving
     , bs: [nat] = []
     , ar: nat
:: len(as) > 0
   *> ( ar:= hd(as); as:= tl(as)
        ; ( x  = ar -> ret bs ++ as
          | x /= ar -> bs:= bs ++ [ar]
          )
      )
]|

func findTTote(val xs: [(ttote, nat, bool)], n: nat)
-> ([(ttote, nat, bool)], (ttote, nat, bool)) =
|[ var ys: [(ttote, nat, bool)] = xs, y: (ttote, nat, bool)
     , zs: [(ttote, nat, bool)] = []
:: len(ys) > 0
   *> ( y:= hd(ys); ys:= tl(ys)
        ; ( y.1 /= n -> zs:= zs ++ [y]
          | y.1  = n -> ret(zs ++ ys, y)
          )
      )
]|

func initNWS() -> %NWS * nat =
|[ var i: nat = 0, vec: %NWS * nat
:: i < %NWS
   *> ( vec.i:= 0; i:= i + 1 )
 ; ret vec
]|

func inittsNWS() -> %NWS * [(subord,nat)] =
|[ var i: nat = 0, vec: %NWS * [(subord,nat)]
:: i < %NWS
   *> ( vec.i:= []; i:= i + 1 )
 ; ret vec
]|
```