An Algebraic Specification of a Model Factory, Part IV

Jacob Brunekreef
Alban Ponse

Programming Research Group, University of Amsterdam

Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

E-mail: `jacob@fwi.uva.nl` , `alban@fwi.uva.nl`

Abstract

In this paper a centralised control process for the DEC CEC model factory is specified. The control process accepts product orders from the outside world and puts the various stations of the model factory to work by a request/reply communication. Different product types require different station sequences. A sequence of station requests is generated in a *routing* process. Two routing processes are specified: one process oriented, the other data oriented.

Key Words & Phrases: Specification Language, PSF, ADT (Abstract Data Types), Process Algebra.
1985 Mathematics Subject Classification: 68Q40.
1987 CR Categories: D.1.3, D.2.1, F.1.2.

Contents

1 Introduction

The work reported on in this paper is part of a project, the overall aim of which is to specify different aspects of a model factory. The actual model factory, and therefore the context of this project is provided by the Amsterdam research site of DIGITAL EQUIPMENT CORPORATION (DEC) called COOPERATIVE ENGINEERING CENTER (CEC)[1].

The purpose of this project is twofold. A first goal is to apply the particular specification language PSF [3] in a modelling on a relatively large scale. The model factory built for and programmed at CEC suits this goal perfectly. Secondly, it is intended to investigate whether desired features can be easily specified and simulated on a symbolic level with PSF. Such an effort would then facilitate an analysis of desired features. Three previous PSF specifications of the model factory that form part of this project are described in [7, 2, 4], respectively. The idea is that the cooperation between DEC CEC and the University of Amsterdam will result in a series of PSF specifications, each of which extends the previous one and will be made by a fresh team (as to guarantee easy accessibility).

The model factory at DEC-CEC produces *Printed Circuit Boards* (PCBs). A PCB is an electrical device consisting of a (print-)*board* with a maximum of six electronic *components* placed upon it. Three kinds of PCBs are manufactured. These are called the *product types* A, B and C. The product types differ on the kind of board and on the number and kind of components used.

During manufacturing, one, two or three PCBs are operated on at the same time. Such a cluster is called a *batch*. The PCBs in a batch stay (conceptually) together during the entire production process and must be of the same product type.

We shortly describe the manufacturing of batches step by step. We adopt a somewhat simplified view on the production process, abstracting from the identity and supply of boards and components, the manual repair loop and the shipping and receiving facilities of the model factory.

Figure 1 shows the track each batch traverses in our modelling. The *Trays with P.C. boards* (TPC) consists of two stores containing empty boards; each store contains boards of one type (one for a PCB of type A, the other for B and C). The empty boards are put from the TPC onto the conveyor belt by board movers, and are further operated on as a batch. Next a batch passes the *Screen Printing Station* (SCP). At this station solder paste is put on the boards. Now the components can be placed. Some components are placed by *Component Placement Station 1* (CP1) and others by *Component Placement Station 2* (CP2). A batch for production of type A has to pass both stations CP1 and CP2, a batch for B must only pass CP1 and a batch for product type C skips CP1, but has to pass CP2 twice. Therefore C-batches have to do the whole tour twice. A batch is transported to and from a Component Placement Station by so called *Movers*, MC1 and MC2. The *Board Mover to Reflow* (BMR) transports a batch to the *Reflow and Clean Station* (RCL). The restore sequence ends in the *In Process Store* (IPS). There are three IPSs, one for each product-type. The PCBs stay here until a product order of a certain type comes in. Then the requested number of PCBs of the requested product type are taken from the IPS and transported further as a batch. The batch passes the *Board Mover to Warehouse* (BMW) to the *Inspection and Electrical Test* (IET) station. This station performs a visual test (are all components well-placed?) and an electronic test. Next the *Board Mover to Repair* (BTR) and the *Buffer to Recycle* (BRE) are passed. If the production of the batch is completed, it is sent to the *Warehouse* (WAR). A C-batch that is only halfway (it has to go through the Placement Station CP2 for the second time) passes the Warehouse and is put in a buffer, the *Second Side Buffer* (SSB). From there the C-batch can be put on the conveyor belt again by a board mover.

In this paper we present a single control process for the whole factory as described above. It is loosely based on a graphical representation of a centralised control architecture from [5]. Referring to all physical components as *stations*, the main tasks for a centralised control process are:

[1]DEC CEC, Einsteingebouw, Kabelweg 21, 1014 BA Amsterdam, The Netherlands, (+31 20 5866000)

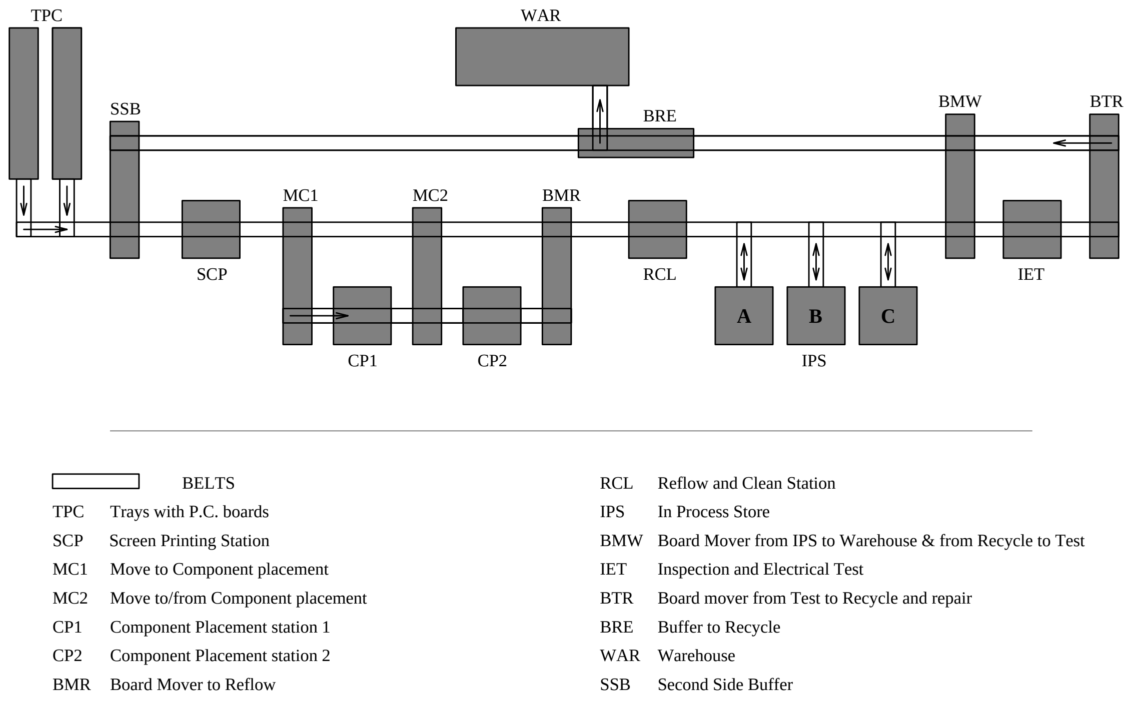

Figure 1: Flow of Products in the model factory

1. The acceptance of product orders from the outside world. In an order the product type (A, B, C) and the order size have to be specified.

2. The in-order processing of the received orders.

3. The correct routing of each order through the model factory. For different product types different routes have to be followed.

4. The optimisation of the efficiency of the model factory. Different products can be processed by different stations of the model factory at the same time.

The first task leads to a simple process that receives a product order from the outside world and stores it in an internal queue. A scheduler process can take an order from this queue, split it in several *batches* if the size is greater than three and for each batch invoke a routing process that actually handles a batch. This process activates a sequence of stations in the model factory. Each batch requires two separate sequences: a *delivery* sequence that transports the required number of PCBs (Printed Circuit Boards) from the appropriate IPS (In Process Store) to the WAR (Warehouse) and a *restore* sequence that takes care of the refill of the IPSs. The first sequence is independent of the product type, the second sequence is not, as it represents the production of a PCB. The efficiency of the model factory is increased if the handling of a subsequent batch starts as soon as possible and is not delayed until a the current batch has been handled completely. This means that the handling of a new batch can be started as soon as the first stations in the delivery sequence and restore sequence are free again.

In [2] a decentralised control is specified: the model factory is split in eight control zones, each of which has its own local control process. This decentralised control architecture reflects the implementation of the model factory at DEC CEC at the time that paper was written. Additionally, this paper contains the specification of production of batches with *variable size*. The subsequent paper [4] is a further study of control as defined in [2]: synchronisation between control and station level can be relaxed by adding a separate layer.

To facilitate the analysis of a centralised control process, the physical part of the model factory is specified in a very abstract way: a station is able to accept a request (with various parameters), after which it produces a reply indicating the successful handling of a batch. We do not deal with more specific states of a station, such as its processing activity or station errors (this is left for future refinement). A physical connection between two stations (a conveyer belt, batch mover or board mover) is modelled by a synchronisation (communication). The specification of the station level of the model factory is described in Section 2.

The centralised control process is introduced in Section 3. The core of this process is a *routing* algorithm, which determines the sequence of stations that has to be traversed for a specific product type. In this paper two alternatives for this routing process are specified. In Section 4 we present a process oriented alternative, in Section 5 a data oriented alternative is discussed. This means that in the first alternative the routing of the delivery and restore sequences is modelled by an explicit sequential composition of requests to stations and replies from stations. In the second specification the routing information is stored in data functions. In Section 6 some concluding remarks are made. The paper ends with Appendix A, each Subsection A.x of which contains the complete PSF specifications discussed in Section x below.

In order to keep the paper readable, in the coming sections only the signature and import information of the data specifications is given. For the process specifications only the import information and the process definitions are given.

2 Specification of the model factory station level

The DEC model factory consist of a number of stations, which perform various actions to produce printed circuit boards (PCBs). Three different types of PCBs are considered: A, B and C. The latter is more complex than the other two, because both sides of the board are used. The data module `Product-Types` represents the PCBs (to be). The products are specified as constants of the sort `PRODUCTTYPE`. A PCB (to be) of type C is specified by three constants, indicating the status of the product. The function `empty` is used in the start of a restore sequence. See further on.

```
data module Product-types
begin
  exports
    begin
      sorts
        PRODUCTTYPE
      functions
        A  :  ->  PRODUCTTYPE
        B  :  ->  PRODUCTTYPE
        C  :  ->  PRODUCTTYPE  -- full  C-board
        Ce :  ->  PRODUCTTYPE  -- empty C-board
        Ch :  ->  PRODUCTTYPE  -- half  C-board
        empty :   PRODUCTTYPE -> PRODUCTTYPE
    end
  .....
end Product-types
```

PCBs are produced in the model factory in so called batches of at most three boards. The data module `Batch-sizes` gives the sort `BATCHSIZE` and the constants `one`, `two` and `three`. The size of a product order is specified as a natural (see the next section). The function `n2b` is used for conversion from a natural to a batch size constant. The module `Naturals` is imported from the standard PSF library ([8]) and defines the natural numbers.

```
data module Batch-sizes
begin
  exports
    begin
      sorts
        BATCHSIZE
      functions
        one   :           ->   BATCHSIZE
        two   :           ->   BATCHSIZE
        three :           ->   BATCHSIZE
        n2b   :  NATURAL  ->   BATCHSIZE
    end
  imports
    Naturals
  ....
end Batch-sizes
```

Each station in the the model factory performs a specific operation on a batch that passes the station. The stations are referred to by a three–letter identifier conform the DEC CEC definition [1]. The data module `Station-Ids` gives the sort `STATION` and the station identifiers. The function `s2n` is used to map the station identifiers on naturals, which simplifies the specification of the equality function `eqs` between stations. Note that the `IPS` station is specified twice with the suffixes `r` and `d`, explicating its different functionalities: the final station in a restore sequence (IPSr), and the first station in a delivery sequence (IPSd). Furthermore, a dummy station (Dum) is specified. These peculiarities will be explained further on.

```
data module Station-Ids
begin
  exports
    begin
      sorts
        STATION
      functions
        TPC : -> STATION
        SSB : -> STATION  -- Second Side Buffer for C-half
        SCP : -> STATION
        MC1 : -> STATION
        MC2 : -> STATION
        CP1 : -> STATION
        CP2 : -> STATION
        BMR : -> STATION
        RCL : -> STATION
        IPSr: -> STATION  -- restore  facility of IPS
        IPSd: -> STATION  -- delivery facility of IPS
        BMW : -> STATION
        IET : -> STATION
        BTR : -> STATION
        WAR : -> STATION
        BRE : -> STATION
        Dum : -> STATION  -- dummy station
```

```
          eqs :    STATION # STATION -> BOOLEAN
      end
    imports
      Naturals
    functions
      s2n : STATION -> NATURAL  -- for equality function
    .....
  end Station-Ids
```

The communication between the stations and the control process is modelled by a request/reply message pair. Requests are parameterised with an additional argument, which contains more details about the requested operation. Although these parameters are not needed in our abstract specification of the station behaviour, they will be needed in a more refined specification. Therefore, in our specification we will add request parameters to station requests.

In the data module `Station-RequestParameters` the sort `REQPARAM` is specified. The request parameter constants are defined conform the DEC CEC parameters [1]. The function `reqp` gives the request parameter for a given station and a given product type.

```
  data module Station-RequestParameters
  begin
    exports
      begin
        sorts
          REQPARAM
        functions
          half     : -> REQPARAM
          forward  : -> REQPARAM  -- = up in layout schemes
          backward : -> REQPARAM  -- = down in layout schemes
          rp-nil   : -> REQPARAM  -- = no parameter
          reqp     :    STATION # PRODUCTTYPE -> REQPARAM
      end
    imports
      Product-types, Station-Ids
    .....
  end Station-RequestParameters
```

Now we are able to specify a station process. In the process module `Stations` the process `TheStations` consists of the merge of all station processes. The core of a station process is the acceptance of a request, followed by the sending of a reply. In the model factory batches are transported from one station to another by conveyer belts. This physical connection between stations is modelled by a synchronisation: a station receives a batch with the action `from-belt`, this action synchronises with the action `to-belt` from another station. The resulting communication is named `comm-belt`.

A request from a routing process is received by a `read-request-RS` action. The two capital letters at the end of the action name identify the source (R, the routing process) and the destination (S, the station process) of the communication action. This naming convention for read–, send– and comm–actions is used throughout this paper. The last parameter of the `read-request-RS` action tells a station to which station the batch has to be transported. This global information about the physical lay out of the model factory is not present at the level of the station processes. The action `send-reply-SR` models the reply from a station process to a routing process. Various data modules are imported for the required sorts and functions. In the definition of a station process in the process module `Stations`, a distinction is made between first stations in a sequence (the stations `TPC` and `IPSd`), final stations in a sequence (the stations `IPSr` and `WAR`), and other stations. First stations have no incoming belt, final stations have no outgoing belt.

```
process module Stations
  .....
  imports
    Station-Ids, Product-types, Batch-sizes, Station-RequestParameters
  .....
  definitions
    TheStations = merge(i  in STATION, Station-process(i))

    Station-process(i) = [or(eqs(i,IPSd),eqs(i,TPC)) = true] ->
        sum(pt in PRODUCTTYPE, sum(bs in BATCHSIZE,
        sum(rp in REQPARAM, sum(ns in STATION,
        read-request-RS(i,pt,bs,rp,ns) . to-belt(ns) .  send-reply-SR(i) .
        Station-process(i)))))
      + [or(eqs(i,WAR),eqs(i,IPSr)) = true] -> from-belt(i) .
        sum(pt in PRODUCTTYPE, sum(bs in BATCHSIZE,
        sum(rp in REQPARAM, read-request-RS(i,pt,bs,rp,Dum) .
        send-reply-SR(i) . Station-process(i))))
      + [not(or(or(eqs(i,IPSd),eqs(i,TPC)),or(eqs(i,WAR),eqs(i,IPSr)))) = true] ->
        from-belt(i) . sum(pt in PRODUCTTYPE,
        sum(bs in BATCHSIZE, sum(rp in REQPARAM, sum(ns in STATION,
        read-request-RS(i,pt,bs,rp,ns) . to-belt(ns) .
        send-reply-SR(i) . Station-process(i)))))
end Stations
```

In the next section the specification of the centralised control process is introduced.

3 A centralised control process

In this section the centralised control process, except for the routing process, is introduced. Two alternatives for the routing process are given in the following sections.

We start with the interface with the outside world: the acceptance of orders and the in–order processing of the received orders. Orders are constructed from two values: the desired PCB type and the number of the desired boards. This is specified in the data module `Orders` with the sort `ORDER` and the function `order`. The constant `order-nil` is required in the binding of the parameter of the queue data structure in the next module.

```
data module Orders
begin
  exports
    begin
      sorts
        ORDER
      functions
        order     : PRODUCTTYPE # NATURAL -> ORDER
        order-nil :                       -> ORDER
    end
  imports
    Naturals, Product-types
end Orders
```

In the process module `OrderQueue` an order is received from the outside world and is stored in a FIFO queue. This is modelled by the action `read-order-EO` (from Environment to OrderQueue). The order at the head of the queue can be sent to a Scheduler process (see below) by the action `send-order-OSC`. (from OrderQueue to SCheduler). The data module `Queues` is imported from the standard PSF library [8].

```
process module OrderQueue
  .....
  imports
    Orders, Queues {Queue-Parameters bound by [Q-ELEMENT -> ORDER,
                                    queue-nil -> order-nil] to Orders}
  .....
  definitions
    OrderQueue(oq) = sum(pt in PRODUCTTYPE, sum(os in NATURAL,
          read-order-EO(pt,os) .  OrderQueue(enqueue(order(pt,os),oq)) ))
     + [gt(length(oq),nat(^0)) = true] -> send-order-OSC(serve(oq)) .
          OrderQueue(dequeue(oq))
end OrderQueue
```

A scheduler process `Scheduler` takes an order from the head of the queue and starts a routing process for a batch of at most three PCBs. If the order size is greater than three, subsequent routing processes are started before the next order is taken from the queue. A new routing process is started as soon as the start stations in the delivery sequence and the restore sequence are free. This situation is achieved by a "permit" synchronisation with a separate process `Start-Permission` (see below). This synchronisation is modelled by an action `read-permit-PS` from `Scheduler` and an action `send-permit-PS` from `Start-Permission`. Here the suffix PS abbreviates: from Permission to Scheduler.

```
process module Scheduler
  .....
  imports
    Orders, Batch-sizes, Routing
```

```
  .....
  definitions
    Scheduler = sum(pt in PRODUCTTYPE, sum(os in NATURAL,
                   read-order-OSC(order(pt, os)) . SubScheduler(pt, os)))
    SubScheduler(pt, os) = read-permit-PS .
                           ([gt(os, nat(^3)) = true] ->
                      (Routing(pt, three) || SubScheduler(pt, os - nat(^3)))
                  +         [gt(os, nat(^3)) = false] ->
                      (Routing(pt, n2b(os)) || Scheduler) )
end Scheduler
```

The scheduler process receives the permission to start the processing of a new batch from the process `Start-Permission`. After a permission is given, this process waits until the permission is returned by both the delivery sequence and the restore sequence in the routing process, before a new permission is made available. The return of a permission to the `Start-Permission` process is modelled by a synchronisation between an action `read-permit-RP` from `Start-Permission` and an action `send-permit-RP` from the routing process (RP: from Routing to Permission).

```
process module Start-Permission
  .....
  definitions
    Start-Permission = send-permit-PS.read-permit-RP.read-permit-RP.
                       Start-Permission
end Start-Permission
```

The complete specification of the model factory consists of the encapsulated parallel composition of the processes `OrderQueue`, `Scheduler`, `TheStations` and `Start-Permission`. In this module the communications between these processes are also defined. The encapsulation enforces synchronisations between the different components.

```
process module ModFac
  .....
  imports
    OrderQueue, Scheduler, Stations, Start-Permission
  .....
  definitions
    ModelFactory = encaps(H, OrderQueue(empty-queue) || Scheduler ||
                             TheStations || Start-Permission)
end ModFac
```

In the next two sections the different alternatives for the routing process will be discussed.

4 Process oriented routing

A routing process consists of the merge of a delivery process and a restore process. In a process oriented version of the routing process, the routing information for the delivery sequence and the restore sequence is explicitly coded in a sequential composition of requests and replies. As the delivery sequence is independent of the product types, one delivery process is sufficient. However, the restore sequence is different for each product type. This results in three different restore processes, as can be seen below.

The sending of a request to a station process is modelled by the action `send-request-RS`, the reception of a reply by `read-reply-SR`. The parameters of the request have been explained with the station process. The routing process returns a start permission to the process `Start-Permission` by the action `send-permit-RP`.

```
process module Routing
  .....
  imports
    Station-Ids, Station-RequestParameters, Batch-sizes, Product-types
  .....
  definitions
    Routing(pt, bs)  = Delivery(pt, bs) || Restore(empty(pt), bs)

    Delivery(pt, bs) =
    send-request-RS(IPSd,pt,bs,forward,BMW).send-permit-RP.read-reply-SR(IPSd).
    send-request-RS(BMW,pt,bs,rp-nil,IET).read-reply-SR(BMW).
    send-request-RS(IET,pt,bs,rp-nil,BTR).read-reply-SR(IET).
    send-request-RS(BTR,pt,bs,forward,BRE).read-reply-SR(BTR).
    send-request-RS(BRE,pt,bs,rp-nil,WAR).read-reply-SR(BRE).
    send-request-RS(WAR,pt,bs,forward,Dum).read-reply-SR(WAR)

    Restore(A,bs) =
    send-request-RS(TPC,A,bs,rp-nil,SCP).send-permit-RP.read-reply-SR(TPC).
    send-request-RS(SCP,A,bs,rp-nil,MC1).read-reply-SR(SCP).
    send-request-RS(MC1,A,bs,backward,CP1).read-reply-SR(MC1).
    send-request-RS(CP1,A,bs,rp-nil,MC2).read-reply-SR(CP1).
    send-request-RS(MC2,A,bs,rp-nil,CP2).read-reply-SR(MC2).
    send-request-RS(CP2,A,bs,rp-nil,BMR).read-reply-SR(CP2).
    send-request-RS(BMR,A,bs,forward,RCL).read-reply-SR(BMR).
    send-request-RS(RCL,A,bs,rp-nil,IPSr).read-reply-SR(RCL).
    send-request-RS(IPSr,A,bs,backward,Dum).read-reply-SR(IPSr)

    Restore(B,bs) =
    send-request-RS(TPC,B,bs,rp-nil,SCP).send-permit-RP.read-reply-SR(TPC).
    send-request-RS(SCP,B,bs,rp-nil,MC1).read-reply-SR(SCP).
    send-request-RS(MC1,B,bs,backward,CP1).read-reply-SR(MC1).
    send-request-RS(CP1,B,bs,rp-nil,MC2).read-reply-SR(CP1).
    send-request-RS(MC2,B,bs,forward,BMR).read-reply-SR(MC2).
    send-request-RS(BMR,B,bs,forward,RCL).read-reply-SR(BMR).
    send-request-RS(RCL,B,bs,rp-nil,IPSr).read-reply-SR(RCL).
    send-request-RS(IPSr,B,bs,backward,Dum).read-reply-SR(IPSr)

    Restore(Ce, bs) =
    send-request-RS(TPC,Ce,bs,rp-nil,SCP).send-permit-RP.read-reply-SR(TPC).
    send-request-RS(SCP,Ce,bs,rp-nil,MC1).read-reply-SR(SCP).
    send-request-RS(MC1,Ce,bs,rp-nil,MC2).read-reply-SR(MC1).
    send-request-RS(MC2,Ce,bs,backward,CP2).read-reply-SR(MC2).
```

```
    send-request-RS(CP2,Ce,bs,rp-nil,BMR).read-reply-SR(CP2).
    send-request-RS(BMR,Ce,bs,forward,RCL).read-reply-SR(BMR).
    send-request-RS(RCL,Ce,bs,rp-nil,BMW).read-reply-SR(RCL).
    send-request-RS(BMW,Ce,bs,forward,BRE).read-reply-SR(BMW).
    send-request-RS(BRE,Ce,bs,rp-nil,SSB).read-reply-SR(BRE).
    send-request-RS(SSB,Ch,bs,half,SCP).read-reply-SR(SSB).
    send-request-RS(SCP,Ch,bs,rp-nil,MC1).read-reply-SR(SCP).
    send-request-RS(MC1,Ch,bs,rp-nil,MC2).read-reply-SR(MC1).
    send-request-RS(MC2,Ch,bs,backward,CP2).read-reply-SR(MC2).
    send-request-RS(CP2,Ch,bs,rp-nil,BMR).read-reply-SR(CP2).
    send-request-RS(BMR,Ch,bs,forward,RCL).read-reply-SR(BMR).
    send-request-RS(RCL,Ch,bs,rp-nil,IPSr).read-reply-SR(RCL).
    send-request-RS(IPSr,Ch,bs,backward,Dum).read-reply-SR(IPSr)
end Routing
```

5 Data oriented routing

In a data oriented version of the routing process the routing information for batches of different product types is kept in data functions. This means that a routing process will be specified that is highly transparent with respect to a specific route (sequence of stations). The routing information for different product types is now stored in the data module `Station-Sequences`. The function `next` in this module returns for a given station and product type the next station in line.

```
data module Station-Sequences
begin
  exports
    begin
      functions
        next : STATION # PRODUCTTYPE -> STATION
    end
  imports
    Station-Ids, Product-types
  .....
end Station-Sequences
```

All routing sequences in the model factory are straightforward from station X to station Y, except the restore sequence for C products. This sequence contains a loop. The function `pmod` from the data module `Product-Modifier` is used in modelling this loop. The function takes a product type and a station identifier as inputs and returns a product type. With this function the product type of a C–batch is changed from `Ce` to `Ch` at the station `BRE`. For all other product types and stations this function is defined as the identity function.

```
data module Product-Modifier
begin
  exports
  begin
    functions
      pmod : PRODUCTTYPE # STATION -> PRODUCTTYPE
  end
  imports
    Product-types,Station-Ids
  .....
end Product-Modifier
```

As mentioned before, in this data oriented specification the routing process for a certain batch is very general. In the process module `Routing` the routing process is specified as the merge of two "sub–routing" processes: one for the delivery sequence and one for the restore sequence. The process `Sub-Routing` is parameterised with three stations, the product type and the batch size. The second and third station parameter indicate the first station and the final station of a sequence. The first station parameter indicates the station that is actually occupied by the batch. By comparing this station parameter with the second and third parameter the first station and the last station in a sequence of stations can be identified. These stations require a deviant approach: the first station has no incoming belt, the last station has no outgoing belt. Furthermore, the first station has to return a start permission to the process `Start-Permission`.

The functions `next` and `pmod`, discussed before, guarantee that for a certain product type the right station sequences are visited. The actions in this routing process are the same as the actions in the routing process in the previous section.

```
process module Routing
  .....
  imports
    Station-RequestParameters, Station-Sequences, Batch-sizes,
    Product-Modifier
  .....
  definitions
    Routing(pt,bs)  = Sub-Routing(IPSd,IPSd,WAR,pt,bs) ||
                      Sub-Routing(TPC,TPC,IPSr,empty(pt),bs)

    Sub-Routing(s1,s2,s3,pt,bs) =
             send-request-RS(s1,pt,bs,reqp(s1,pt),next(s1,pt)).
             read-reply-SR(s1).
             (   [eqs(s1,s2) = true] -> send-permit-RP.
                      Sub-Routing(next(s1,pt),s2,s3,pmod(pt,s1),bs)
               + [eqs(s1,s3) = true] -> skip
               + [or(eqs(s1,s2),eqs(s1,s3)) = false] ->
                      Sub-Routing(next(s1,pt),s2,s3,pmod(pt,s1),bs) )
end Routing
```

Remark: the difference between the two routing processes is bigger than it may seem, regarding the parts of the modules depicted so far. Most of the imported data modules of the data oriented module are much larger than those imported in the process oriented module (see Appendix A.4-5).

6 Conclusions

6.1 Centralised control versus decentralised control

It appears that specifying a centralised control is much simpler than the specification of a decentralised or *distributed* control architecture as was done in the previous papers [2, 4]. [2]

Though this is not very surprising, there is some more to be said. In this paper we take as point of departure (the description of) the production of batches. Production is split over various (batch) orders. Per order, separate routing processes are generated. Such a routing process defines (or gives access to) all algorithms necessary. So, orders can be seen as *generic*: they yield routing processes, each of which comprises the complete production process; there is no modular view on production. Interaction problems, such as the overtaking of batches by each other, are solved by global means. For instance, a station can only send a batch to its subsequent station whenever no other station is sending a batch to that particular station. In short, centralised control specification is not disturbed by (irrelevant) physical constraints.

[2]This can be immediately seen from the size of the PSF specifications involved.

In the previous two specifications [2, 4], orders (there called 'requests') "travel" by synchronisation through the model factory, yielding activity wherever and whenever appropriate. This kind of modelling is much more complicated than the one reported on in this paper. A possible advantage, however, is that it is based on (and therefore facilitates) analysis of the interaction between control and processing activity in *parts* of the model factory. In short, decentralised control specification supports a modular view on the production process.

6.2 Process orientation versus data orientation

We distinguished a process oriented and a data oriented specification of the control of the model factory. In the field of specification of real life situations, a working hypothesis supporting the distinction between *processes* and *data* is the following:

> In the case of *processes* one is as well interested in the properties of some particular process, as in mathematical properties such as (manipulations on) the representation of processes. A process is an object to be *specified*, as to investigate its possibly different executions.
>
> With *data* one often models a state of affairs that is thought to be clear. The rewriting of data itself is not considered especially interesting. For example, one can be interested in the derivability or truth of a relation between data objects, but not so much in how such a result is obtained.

Of course, in a philosophical sense, processes constitute nothing more than a special data type with some predefined (process) operators.

In the following we try to distinguish some characteristics of our process oriented and data oriented control specification of the model factory:

Process oriented control. This type of control specification allows relatively simple adaptations of the modelling of the physical part of the model factory: the different algorithms are defined in a single process specification.

Moreover it allows fast manipulation with the tools developed thus far for PSF [6].

Data oriented control. This type of control specification allows a modular and abstract modelling of the algorithms of the model factory. For instance, it is easy to specify the production of a new type of PCBs.

Manipulation with the PSF tools still is rather fast.

6.3 Future work

Finally, three directions along which further research can take place are the following:

1. Extending our specifications with features not yet considered: supply of boards and components, (manual) repair loop and shipping and receiving of PCBs.

2. Refinement of the current specifications in the following sense. As yet, stations behave as ideal stations: no errors or other problems are reported back to the control process. A refinement in the style of [1] would be to specify stations as having an internal state, reflecting their actual status. This implies that control should also be refined.

3. Refinement of control specification conform the ideas of [5], to be further specified and discussed with the author and other parties concerned.

References

[1] R. Hakkesteegt, R. van Riessen, L. Szakal and W.-J. van Zwienen. Functional description of the PLC program for the model factory. Technical Report Digital Cooperative Center, Amsterdam, November 2, 1992.

[2] J.A. Hillebrand and A. Ponse. An algebraic specification of a model factory, part II. Technical Report P9214, Programming Research Group, University of Amsterdam, 1992.

[3] S. Mauw and G.J. Veltink. A process specification formalism. *Fund. Inf.*, XII:85–139, 1990.

[4] A. Ponse and J.A. Verschuren. An algebraic specification of a model factory, part III. Technical Report P9303, Programming Research Group, University of Amsterdam, 1993.

[5] L. Szakal. Unpublished work.

[6] G.J. Veltink. The PSF toolkit. Technical Report P9107, Programming Research Group, University of Amsterdam, 1991.

[7] S.F.M. van Vlijmen and A. van Waveren. An algebraic specification of a model factory. Technical Report P9209, Programming Research Group, University of Amsterdam, 1992.

[8] J.J. van Wamel. A library for PSF. Technical Report P9301, Programming Research Group, University of Amsterdam, 1993.

A Complete listing of the PSF modules with comments

A.1 Introduction

In the previous sections only some parts of the data modules and the process modules were given. In this appendix the complete PSF specification is given. A few remarks in advance:

- All data functions are specified as *total* functions. This means that the number of equations is sometimes higher than strictly needed in the actual specification.
- The specifications in this appendix can be (and have been) simulated with the PSF simulation tool. For simulation purposes a process module `SimSets` has been specified, which contains the definition of the sets that are needed in order to make the specifications suitable for the simulator. The import section of several process modules may contain the module `SimSets` instead of other modules. In this way the import section of some modules in this appendix differs from the incomplete modules given before. The process module `SimSets` is given below.
- The structure of this appendix is the same as the structure in the paper: each Subsection A.x of the appendix corresponds with Section x in the paper.

Specification of the process module `SimSets`:

```
process module SimSets
begin
  exports
  begin
    sets of NATURAL
      NATURAL-set =
           {nat(^1),nat(^2),nat(^3),nat(^4)}
         of PRODUCTTYPE
      PRODUCTTYPE-set = {A,B,C}
         of PRODUCTTYPE
      PRODUCTTYPE-set-2 = {A,B,Ce,Ch,C}
         of BATCHSIZE
      BATCHSIZE-set = {one,two,three}
         of REQPARAM
      REQPARAM-set = {forward,backward,rp-nil,half}
         of STATION
      STATION-set = {TPC,SSB,SCP,MC1,MC2,CP1,CP2,BMR,RCL,IPSr,IPSd,BMW,IET,BTR,
                     WAR,BRE}
  end
  imports
    Product-types, Batch-sizes, Station-Ids, Station-RequestParameters
end SimSets
```

A.2 The modules of the model factory station level

The data modules `Product-Types`, `Batch-sizes` and `Station-Ids` need no further explanation.

```
data module Product-types
begin
  exports
    begin
      sorts
        PRODUCTTYPE
      functions
```

```
        A  :  ->  PRODUCTTYPE
        B  :  ->  PRODUCTTYPE
        C  :  ->  PRODUCTTYPE  -- full  C-board
        Ce :  ->  PRODUCTTYPE  -- empty C-board
        Ch :  ->  PRODUCTTYPE  -- half  C-board
        empty :   PRODUCTTYPE -> PRODUCTTYPE
    end
  equations
  [1] empty(A) = A
  [2] empty(B) = B
  [3] empty(C) = Ce
  [4] empty(Ch) = Ce
  [5] empty(Ce) = Ce
end Product-types


data module Batch-sizes
begin
  exports
    begin
      sorts
        BATCHSIZE
      functions
        one   :            ->   BATCHSIZE
        two   :            ->   BATCHSIZE
        three :            ->   BATCHSIZE
        n2b   :  NATURAL  ->   BATCHSIZE
    end
  imports
    Naturals
  variables
    n : -> NATURAL
  equations
    [BS1] n2b(nat(^1)) = one
    [BS2] n2b(nat(^2)) = two
    [BS3] n2b(nat(^3)) = three
    [BS4] n2b(n) = three when or(eq(n,nat(^0)),gt(n,nat(^3))) = true
end Batch-sizes


data module Station-Ids
begin
  exports
    begin
      sorts
        STATION
      functions
        TPC : -> STATION
        SSB : -> STATION  -- Second Side Buffer for C-half
        SCP : -> STATION
        MC1 : -> STATION
        MC2 : -> STATION
        CP1 : -> STATION
        CP2 : -> STATION
        BMR : -> STATION
```

```
        RCL : -> STATION
        IPSr: -> STATION  -- restore  facility of IPS
        IPSd: -> STATION  -- delivery facility of IPS
        BMW : -> STATION
        IET : -> STATION
        BTR : -> STATION
        WAR : -> STATION
        BRE : -> STATION
        Dum : -> STATION  -- dummy station
        eqs :    STATION # STATION -> BOOLEAN
    end
  imports
    Naturals
  functions
    s2n : STATION -> NATURAL  -- for equality function
  variables
    s1,s2 : -> STATION
  equations
  [1]  s2n(TPC) = nat(^0)
  [2]  s2n(SSB) = nat(^1)
  [3]  s2n(SCP) = nat(^2)
  [4]  s2n(MC1) = nat(^3)
  [5]  s2n(MC2) = nat(^4)
  [6]  s2n(CP1) = nat(^5)
  [7]  s2n(CP2) = nat(^6)
  [8]  s2n(BMR) = nat(^7)
  [9]  s2n(RCL) = nat(^8)
  [10] s2n(IPSr) = nat(^9)
  [11] s2n(IPSd) = nat(^1^0)
  [12] s2n(BMW) = nat(^1^1)
  [13] s2n(IET) = nat(^1^2)
  [14] s2n(BTR) = nat(^1^3)
  [15] s2n(WAR) = nat(^1^4)
  [16] s2n(BRE) = nat(^1^5)
  [17] s2n(Dum) = nat(^1^6)
  [18] eqs(s1,s2) = eq(s2n(s1),s2n(s2))
end Station-Ids
```

The function `reqp` in the module `Station-RequestParameters` is only used in the data oriented routing module from Section 5. This function is not needed in the process oriented routing module.

```
data module Station-RequestParameters
begin
  exports
    begin
      sorts
        REQPARAM
      functions
        half     : -> REQPARAM
        forward  : -> REQPARAM -- = up in layout schemes
        backward : -> REQPARAM -- = down in layout schemes
        rp-nil   : -> REQPARAM -- = no parameter
        reqp     :    STATION # PRODUCTTYPE -> REQPARAM
    end
  imports
    Product-types, Station-Ids
```

```
  variables
    p : -> PRODUCTTYPE
  equations
  [1]  reqp(TPC,p) = rp-nil
  [2]  reqp(SSB,p) = half
  [3]  reqp(SCP,p) = rp-nil
  [4]  reqp(MC1,A) = backward
  [5]  reqp(MC1,B) = backward
  [6]  reqp(MC1,Ce) = rp-nil
  [7]  reqp(MC1,Ch) = rp-nil
  [8]  reqp(MC1,C) = rp-nil
  [9]  reqp(MC2,A) = rp-nil
  [10] reqp(MC2,B) = forward
  [11] reqp(MC2,Ce) = backward
  [12] reqp(MC2,Ch) = backward
  [13] reqp(MC2,C) = rp-nil
  [14] reqp(CP1,p) = rp-nil
  [15] reqp(CP2,p) = rp-nil
  [16] reqp(BMR,A) = forward
  [17] reqp(BMR,B) = rp-nil
  [18] reqp(BMR,Ce) = forward
  [19] reqp(BMR,Ch) = forward
  [20] reqp(BMR,C) = rp-nil
  [21] reqp(RCL,p) = rp-nil
  [22] reqp(IPSr,p) = backward
  [23] reqp(IPSd,p) = forward
  [24] reqp(BMW,A) = rp-nil
  [25] reqp(BMW,B) = rp-nil
  [26] reqp(BMW,Ce) = forward
  [27] reqp(BMW,Ch) = forward
  [28] reqp(BMW,C) = rp-nil
  [29] reqp(IET,p) = rp-nil
  [30] reqp(BTR,p) = forward
  [31] reqp(BRE,p) = rp-nil
  [32] reqp(WAR,p) = forward
  [33] reqp(Dum,p) = rp-nil
end Station-RequestParameters
```

In the definition of a station process in the process module `Stations`, a distinction is made between first stations in a sequence (the stations `TPC` and `IPSd`), final stations in a sequence (the stations `IPSr` and `WAR`), and other stations. First stations have no incoming belt, final stations have no outgoing belt. As mentioned before, all stations behave as ideal stations: no errors or other problems are reported back to the control process. This refinement is left for future work.

```
process module Stations
begin
  exports
  begin
    atoms
      from-belt       : STATION
      to-belt         : STATION
      comm-belt       : STATION
      read-request-RS : STATION # PRODUCTTYPE # BATCHSIZE # REQPARAM  # STATION
      send-reply-SR   : STATION
    processes
      TheStations
```

```
    end
    imports
      SimSets
    processes
      Station-process : STATION
    communications
      to-belt(s) | from-belt(s) = comm-belt(s) for s in STATION
    variables
      i : -> STATION
      pt : -> PRODUCTTYPE
      bs : -> BATCHSIZE
      rp : -> REQPARAM
    definitions
      TheStations = merge(i  in STATION-set, Station-process(i))

      Station-process(i) = [or(eqs(i,IPSd),eqs(i,TPC)) = true] ->
          sum(pt in PRODUCTTYPE-set-2, sum(bs in BATCHSIZE-set,
          sum(rp in REQPARAM-set, sum(ns in STATION-set,
          read-request-RS(i,pt,bs,rp,ns) . to-belt(ns) .  send-reply-SR(i) .
          Station-process(i)))))
        + [or(eqs(i,WAR),eqs(i,IPSr)) = true] -> from-belt(i) .
          sum(pt in PRODUCTTYPE-set-2, sum(bs in BATCHSIZE-set,
          sum(rp in REQPARAM-set, read-request-RS(i,pt,bs,rp,Dum) .
          send-reply-SR(i) . Station-process(i))))
        + [not(or(or(eqs(i,IPSd),eqs(i,TPC)),or(eqs(i,WAR),eqs(i,IPSr)))) = true] ->
          from-belt(i) . sum(pt in PRODUCTTYPE-set-2,
          sum(bs in BATCHSIZE-set, sum(rp in REQPARAM-set, sum(ns in STATION-set,
          read-request-RS(i,pt,bs,rp,ns) . to-belt(ns) .
          send-reply-SR(i) . Station-process(i)))))
  end Stations
```

A.3 The modules of the centralised control process

The data module `Orders` needs no further explanation.

```
data module Orders
begin
  exports
    begin
      sorts
        ORDER
      functions
        order     : PRODUCTTYPE # NATURAL -> ORDER
        order-nil :                       -> ORDER
    end
  imports Naturals, Product-types
end Orders
```

The process `OrderQueue` from the process module with the same name enqueues orders from the outside world and delivers the first order in the queue to the Scheduler process. Of course, an order can only be delivered whenever the queue is not empty.

```
process module OrderQueue
begin
  exports
    begin
```

```
     atoms
       read-order-EO:  PRODUCTTYPE # NATURAL
       send-order-OSC: ORDER
     processes
       OrderQueue: QUEUE
    end
  imports
    SimSets,Queues {Queue-Parameters bound by [Q-ELEMENT -> ORDER,
                                      queue-nil -> order-nil] to Orders}
  variables
    oq: -> QUEUE
  definitions
    OrderQueue(oq) = sum(pt in PRODUCTTYPE-set, sum(os in NATURAL-set,
          read-order-EO(pt,os) .  OrderQueue(enqueue(order(pt,os),oq)) ))
     + [gt(length(oq),nat(^0)) = true] -> send-order-OSC(serve(oq)) .
          OrderQueue(dequeue(oq))
end OrderQueue
```

The process `Scheduler` from the process module with the same name reads an order from the order queue and invokes the process `SubScheduler`. This process first waits for a starting permit and, once this permit is received, starts a routing process for a batch of at most three PCBs. If the order size is greater than three, the SubScheduler process recursively invokes itself for the handling of the remaining batches. In parallel with the start of the last batch, the Scheduler process itself is invoked again, as to receive new orders. The process `Routing` is a sub–process of the Scheduler process.

```
process module Scheduler
begin
  exports
    begin
     atoms
       read-order-OSC: ORDER
       read-permit-PS
     processes
       Scheduler
       SubScheduler: PRODUCTTYPE # NATURAL
    end
  imports
    Orders, SimSets, Routing
  variables
    pt: -> PRODUCTTYPE
    os: -> NATURAL
  definitions
    Scheduler = sum(pt in PRODUCTTYPE-set, sum(os in NATURAL-set,
                       read-order-OSC(order(pt, os)) . SubScheduler(pt, os)))
    SubScheduler(pt, os) = read-permit-PS .
                               ([gt(os, nat(^3)) = true] ->
                          (Routing(pt, three) || SubScheduler(pt, os - nat(^3)))
                     +          [gt(os, nat(^3)) = false] ->
                          (Routing(pt, n2b(os)) || Scheduler) )
end Scheduler
```

The process `Start-Permission` is very simple: it sends a start permission to the Scheduler process, after which it expects the return of this permission from each routing sub–process (in our case: the delivery process and the restore process). Although small, the role of this process is important: it guarantees the in–order handling of the product orders as stored in the order queue.

```
process module Start-Permission
begin
  exports
  begin
    atoms
      send-permit-PS
      read-permit-RP
    processes
      Start-Permission
  end
  definitions
    Start-Permission = send-permit-PS.read-permit-RP.read-permit-RP.
                       Start-Permission
end Start-Permission
```

The process module `ModFac` contains the definition of the process `ModelFactory`: the encapsulated parallel composition of the processes involved in the model factory. Furthermore, this module contains the definitions of the encapsulation set and the communications.

```
process module ModFac
begin
  exports
    begin
     atoms
       comm-order-OSC:  ORDER
       comm-request-RS : STATION # PRODUCTTYPE # BATCHSIZE # REQPARAM # STATION
       comm-reply-SR : STATION
       comm-permit-PS
       comm-permit-RP
      processes
        ModelFactory
    end
  imports
    OrderQueue, Scheduler, Stations, Start-Permission
  sets of atoms
   H =   {send-order-OSC(o), read-order-OSC(o) | o in ORDER}
       + {send-request-RS(s,p,b,r,n), read-request-RS(s,p,b,r,n) | s in STATION,
             p in PRODUCTTYPE, b in BATCHSIZE, r in REQPARAM, n in STATION}
       + {send-reply-SR(s) , read-reply-SR(s) | s in STATION}
       + {to-belt(s) , from-belt(s) | s in STATION}
       + {send-permit-PS , read-permit-PS , send-permit-RP , read-permit-RP}
  communications
    send-order-OSC(o) | read-order-OSC(o) = comm-order-OSC(o) for o in ORDER
    send-request-RS(s,p,b,r,n) | read-request-RS(s,p,b,r,n) =
              comm-request-RS(s,p,b,r,n) for s in STATION, p in PRODUCTTYPE,
                 b in BATCHSIZE, r in REQPARAM, n in STATION
    send-reply-SR(s) | read-reply-SR(s) =
              comm-reply-SR(s) for s in STATION
    send-permit-PS | read-permit-PS = comm-permit-PS
    send-permit-RP | read-permit-RP = comm-permit-RP
  definitions
    ModelFactory = encaps(H, OrderQueue(empty-queue) || Scheduler ||
                               TheStations || Start-Permission)
end ModFac
```

A.4 The modules of the process oriented routing

As described in the accompanying Section 4, the process oriented routing process contains an explicit sequential composition of requests and replies for various sequences and product types. The routing process consists of a parallel composition of a delivery process and a restore process. The function `empty` guarantees that a restore process for a C batch starts with empty C–boards. A request is sent to a station with the action `send-request-RS`, a reply is received with the action `read-reply-SR`. In the beginning of each delivery process and restore process the start permission is returned to the process `Start-Permission` by the action `send-permit-RP`.

```
process module Routing
begin
  exports
    begin
     atoms
       send-request-RS : STATION # PRODUCTTYPE # BATCHSIZE # REQPARAM # STATION
       read-reply-SR :   STATION
       send-permit-RP
     processes
        Routing: PRODUCTTYPE # BATCHSIZE
    end
  imports
    Station-Ids, Station-RequestParameters, Batch-sizes, Product-types
  processes
    Delivery: PRODUCTTYPE # BATCHSIZE
    Restore:  PRODUCTTYPE # BATCHSIZE
  variables
    pt: -> PRODUCTTYPE
    bs: -> BATCHSIZE
  definitions
    Routing(pt, bs)  = Delivery(pt, bs) || Restore(empty(pt), bs)

    Delivery(pt, bs) =
    send-request-RS(IPSd,pt,bs,forward,BMW).send-permit-RP.read-reply-SR(IPSd).
    send-request-RS(BMW,pt,bs,rp-nil,IET).read-reply-SR(BMW).
    send-request-RS(IET,pt,bs,rp-nil,BTR).read-reply-SR(IET).
    send-request-RS(BTR,pt,bs,forward,BRE).read-reply-SR(BTR).
    send-request-RS(BRE,pt,bs,rp-nil,WAR).read-reply-SR(BRE).
    send-request-RS(WAR,pt,bs,forward,Dum).read-reply-SR(WAR)

    Restore(A,bs) =
    send-request-RS(TPC,A,bs,rp-nil,SCP).send-permit-RP.read-reply-SR(TPC).
    send-request-RS(SCP,A,bs,rp-nil,MC1).read-reply-SR(SCP).
    send-request-RS(MC1,A,bs,backward,CP1).read-reply-SR(MC1).
    send-request-RS(CP1,A,bs,rp-nil,MC2).read-reply-SR(CP1).
    send-request-RS(MC2,A,bs,rp-nil,CP2).read-reply-SR(MC2).
    send-request-RS(CP2,A,bs,rp-nil,BMR).read-reply-SR(CP2).
    send-request-RS(BMR,A,bs,forward,RCL).read-reply-SR(BMR).
    send-request-RS(RCL,A,bs,rp-nil,IPSr).read-reply-SR(RCL).
    send-request-RS(IPSr,A,bs,backward,Dum).read-reply-SR(IPSr)

    Restore(B,bs) =
    send-request-RS(TPC,B,bs,rp-nil,SCP).send-permit-RP.read-reply-SR(TPC).
    send-request-RS(SCP,B,bs,rp-nil,MC1).read-reply-SR(SCP).
    send-request-RS(MC1,B,bs,backward,CP1).read-reply-SR(MC1).
    send-request-RS(CP1,B,bs,rp-nil,MC2).read-reply-SR(CP1).
```

```
  send-request-RS(MC2,B,bs,forward,BMR).read-reply-SR(MC2).
  send-request-RS(BMR,B,bs,forward,RCL).read-reply-SR(BMR).
  send-request-RS(RCL,B,bs,rp-nil,IPSr).read-reply-SR(RCL).
  send-request-RS(IPSr,B,bs,backward,Dum).read-reply-SR(IPSr)

  Restore(Ce, bs) =
  send-request-RS(TPC,Ce,bs,rp-nil,SCP).send-permit-RP.read-reply-SR(TPC).
  send-request-RS(SCP,Ce,bs,rp-nil,MC1).read-reply-SR(SCP).
  send-request-RS(MC1,Ce,bs,rp-nil,MC2).read-reply-SR(MC1).
  send-request-RS(MC2,Ce,bs,backward,CP2).read-reply-SR(MC2).
  send-request-RS(CP2,Ce,bs,rp-nil,BMR).read-reply-SR(CP2).
  send-request-RS(BMR,Ce,bs,forward,RCL).read-reply-SR(BMR).
  send-request-RS(RCL,Ce,bs,rp-nil,BMW).read-reply-SR(RCL).
  send-request-RS(BMW,Ce,bs,forward,BRE).read-reply-SR(BMW).
  send-request-RS(BRE,Ce,bs,rp-nil,SSB).read-reply-SR(BRE).
  send-request-RS(SSB,Ch,bs,half,SCP).read-reply-SR(SSB).
  send-request-RS(SCP,Ch,bs,rp-nil,MC1).read-reply-SR(SCP).
  send-request-RS(MC1,Ch,bs,rp-nil,MC2).read-reply-SR(MC1).
  send-request-RS(MC2,Ch,bs,backward,CP2).read-reply-SR(MC2).
  send-request-RS(CP2,Ch,bs,rp-nil,BMR).read-reply-SR(CP2).
  send-request-RS(BMR,Ch,bs,forward,RCL).read-reply-SR(BMR).
  send-request-RS(RCL,Ch,bs,rp-nil,IPSr).read-reply-SR(RCL).
  send-request-RS(IPSr,Ch,bs,backward,Dum).read-reply-SR(IPSr)
end Routing
```

A.5 The modules of the data oriented routing

In the data module `Station-Sequences` the routing information for the delivery and restore sequences is coded with the `next` function. For each station and product type this function gives the next station in a sequence. If for a specific product type the next station in a sequence is not defined, the `next` function outputs the dummy station. Recall that the restore sequence for a C–batch contains a loop. A different selection of a next station (needed for the exit of the loop) has been made possible by using the different constants for C–products.

```
data module Station-Sequences
begin
  exports
    begin
      functions
        next : STATION # PRODUCTTYPE -> STATION
    end
  imports
    Station-Ids, Product-types
  variables
    s : -> STATION
    p : -> PRODUCTTYPE
  equations
  [1]  next(TPC,p) = SCP
  [2]  next(SSB,p) = SCP
  [3]  next(SCP,p) = MC1
  [4]  next(MC1,A) = CP1
  [5]  next(MC1,B) = CP1
  [6]  next(MC1,Ce) = MC2
  [7]  next(MC1,Ch) = MC2
  [8]  next(MC1,C) = Dum
```

```
    [9]  next(CP1,p) = MC2
    [10] next(MC2,A) = CP2
    [11] next(MC2,B) = BMR
    [12] next(MC2,Ce) = CP2
    [13] next(MC2,Ch) = CP2
    [14] next(MC2,C) = Dum
    [15] next(CP2,p) = BMR
    [16] next(BMR,p) = RCL
    [17] next(RCL,A) = IPSr
    [18] next(RCL,B) = IPSr
    [19] next(RCL,Ce) = BMW
    [20] next(RCL,Ch) = IPSr
    [21] next(RCL,C) = Dum
    [22] next(IPSr,p) = Dum
    [23] next(IPSd,p) = BMW
    [24] next(BMW,A) = IET
    [25] next(BMW,B) = IET
    [26] next(BMW,C) = IET
    [27] next(BMW,Ce) = BRE
    [28] next(BMW,Ch) = Dum
    [29] next(IET,p) = BTR
    [30] next(BTR,p) = BRE
    [31] next(BRE,A) = WAR
    [32] next(BRE,B) = WAR
    [33] next(BRE,Ce) = SSB
    [34] next(BRE,Ch) = Dum
    [35] next(BRE,C) = WAR
    [36] next(WAR,p) = Dum
    [37] next(Dum,p) = Dum
  end Station-Sequences
```

In the data module `Product-Modifier` the product type of a C–batch is changed from `Ce` to `Ch` at the station `BRE`.

```
data module Product-Modifier
begin
  exports
  begin
    functions
      pmod : PRODUCTTYPE # STATION -> PRODUCTTYPE
  end
  imports
    Product-types,Station-Ids
  variables
    s : -> STATION
  equations
  [1] pmod(Ce,s) = Ch when eqs(s,BRE) = true
  [2] pmod(Ce,s) = Ce when eqs(s,BRE) = false
  [3] pmod(A,s) = A
  [4] pmod(B,s) = B
  [5] pmod(Ch,s) = Ch
  [6] pmod(C,s) = C
end Product-Modifier
```

The process module `Routing` contains the definition of the routing process. This process is defined as the merge of two sub–routing processes: one for a delivery sequence and the other defining a restore sequence. A sub–routing process generates a request to a station and waits for the reply of the station. With all needed routing information encoded in the data modules given above, the sub–routing process itself is a general specification of a routing process for a sequence from one station to another. After the sending of a request to a station and the receipt of a reply from the station, three conditional alternatives are specified:

1. The actual station is the first station of the sequence (`s1 = s2`). A start permission is returned to the `Start-Permission` process before the next station is visited.
2. The actual station is the last station of the sequence (`s1 = s3`). The sequence is finished with a `skip` action.
3. The actual station is not the first or the last station. In that case simply the next station is visited.

```
process module Routing
begin
  exports
    begin
     atoms
       send-request-RS : STATION # PRODUCTTYPE # BATCHSIZE # REQPARAM # STATION
       read-reply-SR : STATION
       send-permit-RP
      processes
        Routing: PRODUCTTYPE # BATCHSIZE
    end
  imports
    Station-RequestParameters, Station-Sequences, Batch-sizes,
    Product-Modifier
  processes
    Sub-Routing: STATION # STATION # STATION # PRODUCTTYPE # BATCHSIZE
  variables
    s1,s2,s3 : -> STATION
    pt       : -> PRODUCTTYPE
    bs       : -> BATCHSIZE
  definitions
    Routing(pt,bs)  = Sub-Routing(IPSd,IPSd,WAR,pt,bs) ||
                      Sub-Routing(TPC,TPC,IPSr,empty(pt),bs)

    Sub-Routing(s1,s2,s3,pt,bs) =
             send-request-RS(s1,pt,bs,reqp(s1,pt),next(s1,pt)).
             read-reply-SR(s1).
             (   [eqs(s1,s2) = true] -> send-permit-RP.
                      Sub-Routing(next(s1,pt),s2,s3,pmod(pt,s1),bs)
               + [eqs(s1,s3) = true] -> skip
               + [or(eqs(s1,s2),eqs(s1,s3)) = false] ->
                      Sub-Routing(next(s1,pt),s2,s3,pmod(pt,s1),bs) )
end Routing
```