An Algebraic Specification of a Model Factory, Part II

Joris Hillebrand
Alban Ponse

Programming Research Group, University of Amsterdam

Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Email: `joris@fwi.uva.nl - alban@fwi.uva.nl`

Abstract

A case study about a model factory, investigated at DEC CEC, is reported on. This factory is described and simulated in the specification language PSF.

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 give an algebraic specification 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 two-fold. A first goal is to apply the particular specification language PSF [MV89, Mau90, MV90] in a modelling of relatively large scale. The model factory built for and programmed at CEC suits this goal perfectly. Secondly, it is intended to investigate whether desired, not yet implemented (or even existing) features can be easily specified and simulated on a symbolic level with PSF. This facilitates an analysis of such features. This second goal is approached here by modelling a desirable feature that is not implemented in the current proto type version at DEC's. 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 specification language PSF, a Process Specification Formalism, is developed at the University of Amsterdam. It is a combination of an algebraic specification formalism for data, and process algebra. PSF therefore has an explicit data-part and a process-part using the data. Also a simulation tool exists. However, this paper does not contain an introduction to PSF, references are [MV89], [Mau90], [MV90] and [Vel91].

Acknowledgements. We would like to thank Ronald van Riessen, manager at DEC's, for introducing us to the model factory and giving us the opportunity to study it. Carlo Koopmans and Ruud Hakkesteegt also played an important role in explaining the merits and shortcomings of the present version of the model factory. Bas van Vlijmen and Arjan van Waveren made a previous version of a PSF-specification of the model factory. Their paper has helped us a lot in making the current specification. Gert Veltink, Sjouke Mauw, Hans Mulder and Jos van Wamel were of great help with the use of PSF and its simulator. We thank them for spending the amount of time they did by helping us. Finally we thank Jan Bergstra for giving us the opportunity to work on this project and making the first contacts with DEC.

2 The model factory

In section 2.1 we describe the model factory as we modelled it. Some aspects of the simplification of the actual model factory are discussed in 2.2. In Section 2.3 we mention some characteristics of our modelling and that of our predecessors, who first made 'An Algebraic Specification of a Model Factory' ([VlijWa92]). This section contains no PSF syntax.

2.1 Overall behaviour

The model factory 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. There are two kinds of boards, called *board1* and *board2*, and three kinds of components, referred to as *red*, *yellow* and *green*. Because a PCB does not have to contain exactly six components, a fourth component, *emp*(ty) is introduced. Initially, a PCB is just an empty board.

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.

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

Figure 1 shows the track each batch will traverse. The figure is split up in eight *Control Zones* (CZs), separated by the dashed lines. A CZ is a conceptual unit that connects all manufacturing, storing and transporting units in the model factory to the appropriate local *Controller*. The overall control of the factory is specified as the concurrent execution of all CZs, and will be explained later in more detail. Now we describe the manufacturing of batches step by step, using the CZs merely for reference to locations.

CZ-1. The *Raw Material Stores* (RMSs) are stores containing empty boards; each RMS contains boards of one type. The empty boards are put from the RMSs onto the conveyor belt by board movers, and are further operated on as a batch.

CZ-2. Here the batch passes the *Screen Print Station* (SP). At this station solder past is put on the boards.

CZ-3 and CZ-4. Now the components can be placed. Some components are placed by *Placement Station 1* (CP1) and others by *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.

CZ-5. After the components have been placed each batch passes the *Reflow and Clean Station* (RC) and then goes into an *In Process Store* (IPS). There are three IPSs, one for each product-type. The PCBs stay here until a request comes in at the local Controller. Then the requested number of PCBs of the requested product type are taken from the IPS and transported further as a batch.

CZ-6 and CZ-7. Here the batch passes the *Test and Repair Station* (TR). This station performs a visual test (are all components well-placed?) and an electronic test. If anything is wrong the faulty PCB(s) is (are) repaired and the entire batch must go through the loop in zones CZ-6 and CZ-7 to be tested again. If finally everything is in order, the batch is further transported.

CZ-8. If the production of the batch is completed, it is sent to the *Warehouse*. 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.

2.2 Simplifications

The model factory as described above is a rather simple representation of the one at DEC CEC. We do not consider the supply of accessories (components and empty trays for components and boards) and the shipping of the final products from the Warehouse and therefore assume the RMSs and the Warehouse to be of a looping, respectively infinite capacity. Another simplification concerns the Test and Repair Station TR. We assume that a batch is always first judged faulty and repaired, and when visiting TR for the second time is judged `ok`. In reality however, batches are repaired manually in a special buffer and then put back in the production process. Here also accessories can be needed, which then have to be supplied.

2.3 Modelling of the control structure

In [VlijWa92], a PSF specification of the model factory is given. In that specification the product types, boards, components, PCBs and batches are defined, as well as a number of primary processes such as conveyor belts, movers and a number of workstations. This specification models the production in batches that can contain exactly three PCBs, reflecting the current functionality of the model factory at DEC CEC.

In the future, the model factory is meant to be able to deal with batches of size one or two as well. In the current PSF specification this feature has been added. However, the main characteristic of the

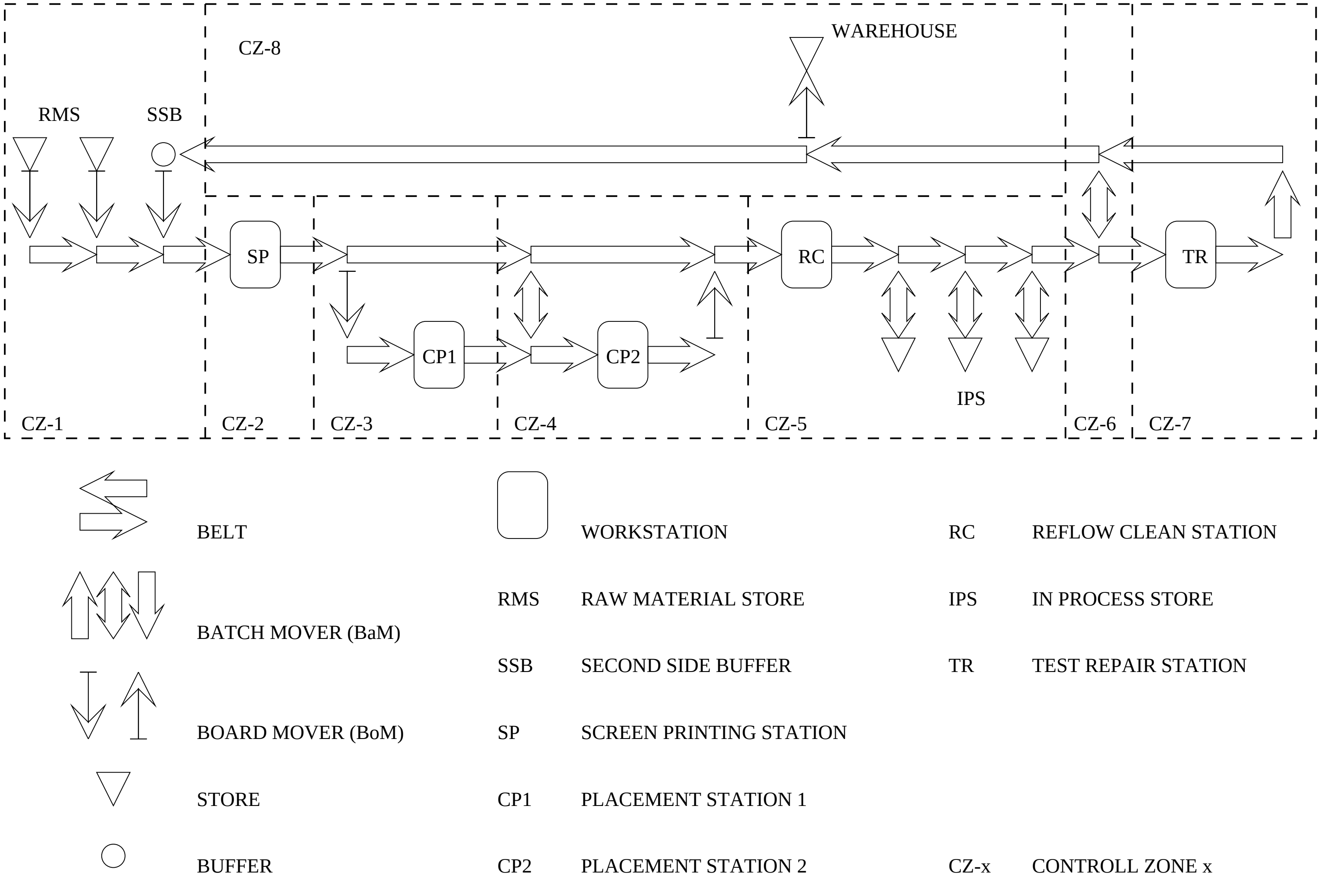
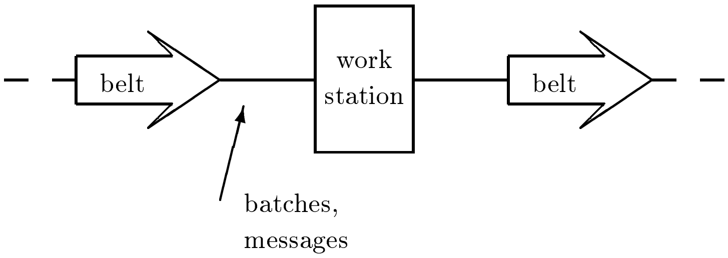
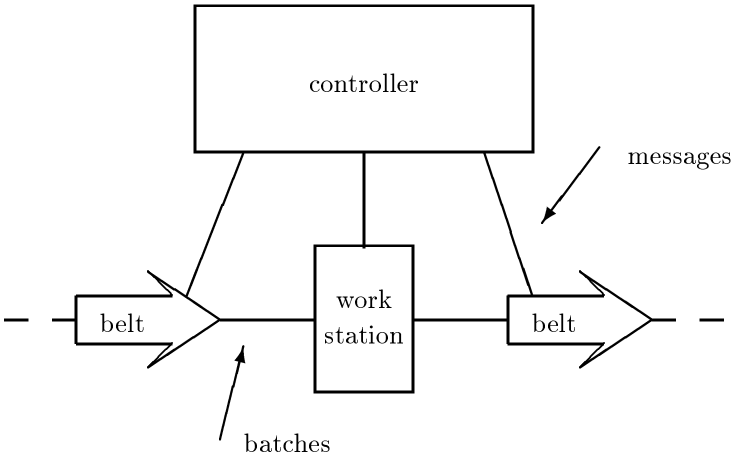


Figure 1: Flow of Products in the model factory

specification described here is its distributed control structure. In the previous version [VlijWa92] the situation was modelled as follows:



All parts of the model factory are connected by ports, on which batches as well as messages are sent. In the current version the ports that connect the manufacturing/transporting units of the model factory are only used for the communication of batches. These are the *batch-ports*. Messages are now all communications to a Controller on special *data-ports*. The Controller can then pass the messages through to another part of the model factory. Now the following situation has been created:



In the next two sections a PSF specification of the data and the processes involved will be studied in detail. Then, in section 5, there are some comments on our specification. Finally, in section 6, the specification as well as some simulation-traces are given.

3 Data specification

We will now give a short explanation of most of the data defined in the data modules of our PSF specification (see Section 6). The data modules are discussed in three separate sections, according to the nature of the data specified in them.

There are three data modules not discussed here:

```
Batch-operations,
Process-ids,
Loop-values.
```

These are explained in Section 4, as they immediately refer to the control structure of the model factory.

3.1 Primitive data types

The primitive data types form the building blocks of the data part of the model factory. Mostly only a sort and a couple of constants of that sort are defined. The primitive data types are defined in the following data modules:

`Naturals.` Defined as usual. The natural numbers are used by the environment (the "world outside") to give an order for a number of products. Also the stock in the RMSs and in the Warehouse are expressed as natural numbers.

`Batch-sizes.` Here constants `one`, `two`, `three` are defined, representing the size a batch can have. As to have addition and subtraction on these constants, two conversion functions 'bn' converting a batch-size to a natural number, and 'nb' are defined. For convenience (in the act of specifying), a constant `zero` has been defined as nb(0). Furthermore nb(x) for $x > s(s(s(0)))$ has been defined as `three`.

`Product types.` There are three product types: `A`, `B` and `C`.

`Boards.` The constant `board1` models the kind of board used for product A, `board2` that for B and C.

`Components.` The components are called `red`, `yellow` and `green`. There is also employ for a constant `emp`, modelling the placement of no component.

`Batch-status.` The status of a batch is used to express the stadium of the production process of that batch. Initially, all batches have status `empty`. After having part of its components placed, a batch has status `half`; it obtains status `fin`(ished) once all its components have been placed (so a B-batch can never have status `half`). Since PCBs do not physically differ any more once they have reached the status `fin`, there also have to exist status constants "marking" a batch. Once a batch is finished and stored in an IPS, it can get out of the IPS marked with status `del`(ivery). After the testing the batch has either status `rep`(ared), or it has status `ok`. The use of these latter three constants is explained below in Section 3.3.

`Ports.` All communications between Controllers and manufacturing/storing/transporting units is on Ports. There are two kinds of ports: *Batch-ports* are used to pass batches, control-messages and requests are being communicated on *data-ports.*

3.2 Dependent data types

Whenever a data type depends on one or more other data types, this data type is said to be *dependent.* This is the case with the data types specified in the following data modules:

`Printed-circuit-boards.` As mentioned in Section 2, a PCB consists of a board with six components (including `emp`) "placed" upon it.

`Pcb-prototypes.` In this module some functions are defined for easy reading of PCB terms much used in the specification. An example is `pcb-half(C)`, denoting a C-batch that is half-way its production.

`Batches.` A batch is specified by its size and by the product type it is going to contain. For instance `batch(pcb-half(C), one)` denotes a C-batch half-way in its production of size one.

`Batch-ids.` A batch-identifier must contain the information about a batch upon which a Controller can infer what must happen next with it. This information concerns `type-to-be` functions (see below), batch-sizes and status constants. Two typical examples are `bid(batch(...))` and `id(A, three, half)`.

`Requests.` There are two kinds of requests. The first is made by the environment and represents a number of products. Such a request is called an `ord`(er) and consists of a function application to a product type and a natural number. The second kind of request is called `req` and is used within the model factory, so always refers to just one batch. It therefore consists of a function application to a product type and a batch-size. The Warehouse-Controller, where orders come in from the environment, transforms an order into one or more requests.

`Messages.` Messages always go from a Controller to a physical part (i.e. a manufacturing/storing/transporting unit) of the model factory, or vice versa. A message from a physical part to a Controller usually reports that part to be available for use (`av`), or that it is holding a certain batch for processing on it: `ho(bid(-))`. In case the physical part is a store, the present stock is reported in the form `pres(natural number)`.

Messages from a Controller to a physical part are usually just `go`-messages, indicating that a belt can be transported, or a workstation can do its job. In the case of a store, the message expresses to take or to give a number of store-items. To make sure a store always knows how many to give (or get) of what, the take- and give-messages are sent in the form of a request: `take(req(-))` or `give(req(-))`.

3.3 Attribute functions

Attribute functions are functions from sort X to sort X and serve as a kind of 'stamp' to keep apart objects that are physically identical, but have functionally a different meaning. Attribute functions occur in the following modules:

`Printed-circuit-boards.` In this module four attribute functions are defined on PCBs. The first two are called `B-to-be` and `C-to-be`. Whenever no components are yet placed, no physical difference exists between a B-PCB and a C-PCB. Both are just an empty board2. Yet Controllers need to know the product type that such a PCB is going to be, in order to direct it to the proper Placement Stations. This information is provided by the attribute functions above.

By definition a PCB is determined by its board and components. Two more attribute functions are needed as to express that a PCB has passed the Screen Printing Station, respectively the Reflow and Clean Station. The attribute functions `sp` and `rc` are used for this purpose.

`Batches.` In this module three attribute functions are defined on batches: `del`, `rep` and `ok`. After a batch has reached status `fin`, it does not change physically anymore. Yet distinction has to be made between batches that are just filling the IPSs and ones that are a delivery for the Warehouse (status `del`). Similarly, when a batch is a delivery the result of the Test and Repair Station (`rep` or `ok`) must be expressed. The relation between these functions and the appropriate status constants is defined by equations of the form

```
bid(del(...))  = id(-, -, del).
```

4 Process specification

As explained in Section 2.1, the model factory has been modelled in eight Control Zones (CZs). One can see this in Figure 1, where each symbol represents a manufacturing/storing/transporting unit of the model factory and will be further called a *low-level process*. The low-level processes are all explained in Section 4.1. Every CZ also has one local *Controller*, a process that can communicate with all low-level processes in that zone (Controllers are not depicted in Figure 1). The low-level processes repeatedly communicate with their Controller, upon which they can undertake their appropriate action. Another task of a Controller is the (local) treatment of requests. Controllers are further described in Section 4.2. In Section 4.3 the specification of the CZs and the top-level process that specifies the behaviour of the model factory are explained.

4.1 Low-level processes

Initially a low-level process sends an 'available'-message to its Controller, after which a 'go'-command is received from the Controller. Then a batch is moved in from another low-level process and a processing action on that batch takes place. Next a 'holding'-message is sent to the Controller. This holding-message also has the `id`(entity) of the batch just processed in it, so that the Controller knows what must happen next. Finally the batch is sent out again and the low-level process returns to its initial state (sending 'available'). So a low-level process globally behaves like

```
send(av) .  read(go) .  read(batch) .  action .  send(ho(id(...)))  .  send(batch)
```

Now we will give an explanation of the low-level process modules in which the basic parts of the factory are specified.

`Belts.` A (conveyor-) belt is a part of the model factory with four batch-ports. When a batch is put on a belt by another belt or by a workstation, that batch is said to be put on the belt from the "main stream". When a batch is put on a belt by a batch- or board-mover, that batch is said to be put on the belt from the "side stream". The same holds for the exits of a belt: when a batch is delivered to another belt or a workstation, the batch is said to be delivered to the main stream; when a batch is delivered to a batch- or board-mover, that batch is said to be delivered to the side stream. Therefore, the process defining a belt has four batch-ports as parameters: 'batch-in-m', 'batch-in-s', batch-out-m' and 'batch-out-s', respectively.

Furthermore a belt has two data-ports, one for a control-message preceding the actual transportation, and another for a control-message following that action. In its defining process these two data-ports are parameters called 'contr-in' and 'contr-out', respectively. Mostly the data-ports are connected to the same Controller. Only when a belt crosses the border of a CZ, the data-port contr-out is connected to the next Controller. In our design only belts can cross CZ-borders. Giving other parts that possibility would make things a lot more complicated.

A belt repeatedly performs the following six steps:

1. Send an available message to the Controller on 'contr-in',
2. Read, on 'contr-in', one of the messages `go-mm`, `go-ms`, `go-sm`, `go-ss`, indicating from which port a batch can be received and to which port it has to be sent,
3. Read a batch on the proper batch-port,
4. Transport the belt,
5. Send a 'holding'-message on 'contr-out',
6. Send the batch out on the proper exit-batch-port.

`Batch-movers.` A batch-mover can move batches from a belt to an other belt or a store. Batch-movers move in a direction square to that of a belt and are said to be moving 'up' or moving 'down' (conform Figure 1). Therefore the process defining a batch-mover has four batch-ports as parameters: 'batch-in-top', 'batch-in-bot', 'batch-out-top' and 'batch-out-bot'. Furthermore, it is sufficient to have only one parameter 'contr' over which data are read/send to the Controller as a parameter.

A batch-mover repeatedly behaves as follows:

1. Send `av` to the Controller on 'contr',

2. Read on ‘contr’ one of the commands `go-up` or `go-dn`,

3. Read a batch on the proper batch-port,

4. Move the batch up or down,

5. Send a holding-signal on ‘contr’,

6. Send the batch out on the proper batch-port.

`Board-movers.` A board mover has the same purpose as a batch-mover, but it does not move a whole batch in one time. It reads in a batch, moves the PCBs in it one by one and sends them out as a batch again. A board-mover only moves PCBs in one direction. In our opinion the only reason why a board-mover is not a batch-mover is to save money and/or space. Because a board-mover only moves in one direction, its definition contains only two batch-port parameters and furthermore one data-port parameter: ‘batch-in’, ‘batch-out’ and ‘contr’, respectively. A board-mover repeatedly behaves as follows:

1. Send `av` on ‘contr’,

2. Read `go` from the Controller on ‘contr’,

3. Read a batch on ‘batch-in’,

4. Move the PCBs in it,

5. Send a holding-message on ‘contr’,

6. Send the PCBs out as a batch on ‘batch-out’.

`Workstations.` There are five workstations in the factory (depicted in Figure 1): the Screen Printing Station (SP), a Component Placement Station 1 (CP1), a Component Placement Station 2 (CP2), a Reflow and Clean Station (RC) and a Test and Repair Station (TR). All these stations perform different actions on batches and therefore have different results on them. However, their control structure can be modelled uniformly, which is done in this module. The process that models the control structure has as parameters two batch-ports, ‘batch-in’ and ‘batch-out’, and one data-port called ‘contr’. Its behaviour is defined by repeatedly performing the following six steps:

1. Send `av` on ‘contr’,

2. Read `go` from the Controller on ‘contr’,

3. Read a batch on ‘batch-in’,

4. Perform the station-action,

5. Send `ho(id(...))` on ‘contr’ with the identity of the **modified** batch,

6. Send the batch out on ‘batch-out’.

The *module* Workstations has three parameters: ‘station-id’, ‘station-action’ and ‘station-result’. The parameter ‘station-id’ serves to identify in any instantiation an independent process operating on the proper ports. The station-action and -result are bound to actual actions and functions in the process modules ‘Screen-printing-station’, ‘Placement-station-1’, ‘Placement-station-2’, ‘Reflow-clean-station’ and ‘Test-repair-station’.

For example, when the parameters are bound in Placement-station-1, ‘station-id’ is bound to `pid2` from the data module `Process-ids`, as this workstation is second in line. The parameter ‘action’ is bound to 'placement-action-1' from process module 'Station-actions' and models the physical placement of the components (moving a robot-arm etc.). The third parameter 'proc' is bound to the function 'comp-place-1' from the data module `Batch-operations` and describes the functional changing of the batch (`pcb-empty(A)` becomes `pcb-half(A)` etc.).

In the modules defining the other workstations, these three parameters are bound in a similar way.

`In-process-stores.` In these stores batches are stored when they have all their components being placed. They come out again when a request for a batch comes in at the local Controller. Any such a store can store at most three products (PCBs) and is a "batch-decoupling-point" where batches can be split: a batch with three PCBs can be stored in an empty store, but when a request comes in for a batch of size one, only one PCB leaves the store as a batch of size one. As there exists for each of the product types A, B and C a local store, the defining IPS process is parameterised over product types. The IPS process has further as parameters one batch-port, 'batch-io', and a data-port 'contr'. The behaviour of an IPS process is as follows:

1. Send the message `pres(n)` on 'contr', expressing that `n` PCBs are currently being stored,

2. Read a message from the Controller on 'contr', being either `take(...)` or `give(...)`, telling the IPS how many PCBs have to be taken or given,

3. *Only in the case of a* `give(...)`*-message:* send a holding message on 'contr' containing the identity of the batch that is going to be given,

4. Depending on the message read in 2, read a batch on 'batch-io' or send the one holding,

5. Update the store size and behave as described in 1.

The status-field of the batch-identifier in the holding-message must be set to `del`(ivery). Incoming batches have status `fin`. This is only a functional difference (not a physical) in order to keep incoming and outcoming batches apart (cf. Section 3.3).

`Raw-material-stores.` The two raw material stores keep empty boards (one store per type). When such a store gets a message to give a batch of a certain size, it takes that number of boards out of the store and sends them out as a batch. The defining process RMS has three parameters: a batch-ports 'batch-io', a data-port 'contr', and a natural number `n`, expressing the number of boards currently stored. The exact behaviour of a RMS is as follows:

1. Send the message `pres(n)` to the Controller on 'contr', expressing that `n` boards are currently being stored,

2. Read on 'contr' the message `give(...)` expressing how many boards have to be given and of which product type the batch is going to be,

3. Send a holding message `ho(id(...))` on 'contr' of the batch that is going to be given,

4. Send the boards out as a batch on 'batch-io', [2]

5. Update the store size and behave as described in 1. [3]

[2]Here is rough simplification being made: it is suggested that in case there is eg. one board in store, the store must be capable of giving one, *two* or *three* empty boards. We return to this point in Section 5.1.

[3]Here a less rough simplification is made: the so called *looping* capacity of the store is assumed (see Section 2.2); this is done by using data from a data module `Loop-values`.

The status-field in the batch-identifyer in the holding message must be set to `empty`. Since empty boards for product types B and C do not differ physically, the RMS must somehow stamp the empty batch, as to clarify of what product type the batch is going to be. The RMS has this information from the request that is sent together with the give-message. The stamp is modelled by the functions `B-to-be` and `C-to-be` (see also previous Section 3.3), to which `pcb-empty(B)` and `pcb-empty(C)` rewrite.

`Warehouse.` The Warehouse is the place where batches finally go to. It is connected to one batch-port and one data-port. Its defining process has one parameter expressing the current number of products stored, and the following behaviour:

1. Send a message `pres(nat)` on the data-port, expressing the number of products currently stored,
2. Read on the data-port the message `take(req(pt, bs)))`, expressing a request for some batch,
3. Read the present batch on the batch-port,
4. Update the store size and behave as described in 1.

`Second-side-buffer.` Batches of product type C pass the Warehouse if they are not yet finished, because they still have to pass Placement Station 2 for the second time (this behaviour is defined in the Warehouse's Controller). The SSB can hold one such batch until it is being put on the belt again by a board-mover. The defining process SSB has as parameters two batch-ports and one data-port. The SSB repeatedly behaves as follows:

1. Send `av` on the data-port,
2. Either

    Read message `reset` on the data-port (further explained in the next section under Second-side-controller),

   or

   (a) Read a message `go` on the data-port,
   (b) Read a batch from the first batch-port
   (c) Send a message `ho(id(...))` on the data-port, expressing what batch is being held,
   (d) Send that batch away on the second batch-port, where it is to be picked up by a board-mover.

4.2 Controllers

A Controller reacts on the `ho(id(...))`-signals from the low-level processes. For each low-level process in a CZ, the local Controller reads the holding-message, sees from `id(...)` what is being held, determines the next low-level process the batch must go to, reads an `av`-message from that next low-level process and gives a proper `go`-message. So a sequence of control-actions globally looks like

```
read(ho(id(...)))  .  read(av) .  send(go)
```

The interaction between a low-level process and its Controller is now given by communications. In Section 6.2 a trace (obtained by a simulation) is depicted that shows the interaction of the low-level processes of CZ-3 and their Controller.

Another task of a Controller is the treatment of requests. This will be explained in the next section where the interaction between Controllers is discussed.

Essentially, all Controllers work alike. For each low-level process in the CZ a sub-control-process is defined. All sub-control-processes are running in parallel. In this way flexibility is increased. Whenever one sub-control-process wants to read `av` from a low-level process that is busy, it can wait until that process is served by another sub-control-process. After that has happened, `av` can be read. Most of the Controllers just have as many of sub-control-processes running in parallel as there are low-level processes in the CZ, plus one sub-control-process that propagates incoming requests. The process modules in which the Controllers are specified are

`Second-side-controller`	(in CZ-1)	`Ips-controller`	(in CZ-5)
`Screen-print-controller`	(in CZ-2)	`Bottleneck-controller`	(in CZ-6)
`Placement-controller-1`	(in CZ-3)	`Test-repair-controller`	(in CZ-7)
`Placement-controller-2`	(in CZ-4)	`Warehouse-controller`	(in CZ-8)

and their defining processes have the same names. Three Controllers show a slightly different behaviour. We will now take a closer look at these.

`Second-side-controller` (CZ-1). The Second-side-controller does not propagate an incoming request, but turns it into an actual batch of the desired size and product-type-to-be. To do so it sends the proper `give(...)`-message to the appropriate RMS. But before that message is sent, the Controller must check whether the Second Side Buffer is empty. If this would be omitted, it is possible that the entire factory gets choked up with C-batches that cannot start their second round, because the first one does not leave the store. The result would be a deadlock (see also [VlijWa92]). If the Second Side Buffer is not empty, it has to be emptied first. So when a request comes in, the Controller first reads an `av`-message from SSB. If it is not empty, `av` cannot be read and the sub-control-process blocks until the SSB has been emptied (by another sub-control-process). After this the `av`-message can be read. Next the request can be handled and finally a reset-message is sent to the SSB. Between the messages `av` and `reset` the SSB is blocked: it can do nothing before `reset` has been received. So reading `av` from SSB can have two purposes: checking availability to be able to handle a request (the next message to the SSB will be `reset`), or checking that the SSB is free to be filled (the next message to the SSB will be `go`).

`Ips-controller` (CZ-5). Because the IPSs s are assumed to be filled, the initial behaviour of the Ips-controller is sending three requests to fill the stores. Whenever products are taken out of a store later, the Ips-controller will immediately send a request to refill that store. The Ips-controller does not directly propagate an incoming request. Instead, it takes the number of products being requested out of the proper store and sends them as a batch to the Test Repair Station. Then the Ips-controller sends a request to refill the store. If the requested number of products is not immediately present, it will be only a matter of time before it is present, as refill-requests are always immediately sent after something has been taken out of a store. To see whether a store has enough products to handle a request, the Controller tries to read a `pres(n)`-message denoting the requested amount or one or two more (a request for one product can also be handled if the store contains two or three products).

`Warehouse-controller` (CZ-8). This is the place where orders from the environment come in. Each order is converted to one or more requests. For example, when an order comes in for 11 products A, the Warehouse-controller sends three requests for an A-batch of size three and one request for an A-batch of size two. In the next section an example of the control-flow of the request-handling is given.

4.3 Control zones

The read and send actions referred to above, are modelled in the process module `Communication`. Globally, there are two kinds of communication:

- batch-communication, and

- control-communication (messages, requests).

In a CZ, all low-level processes, as well as the control-process are put in parallel. The idea is that each workstation or (group of) store(s) form a CZ together with the transportation units moving from or towards it (them). Next, each CZ has been given a Controller. In the proto type implementation at DEC CEC the RC and the IPSs form separate CZs and the present CZ-6 does not exist. Yet the people at DEC's suggested to divide the control as has been done in the current paper, because the batch-mover that is now in CZ-6 gave a lot of problems, being a huge bottleneck.

The CZs are defined in process modules `Control-zone-`x for $x = 1, ..., 8$. In the process definition of a CZ the port-variables (parameters in the process definitions of most of the low-level processes) get bound. The results can be seen in Figures 2 to 9, which are close-ups of the CZs from Figure 1. Batch- and data-ports that cross the CZ-border are said to be *external ports* of that zone. The other ports are *internal ports*. The process defining a CZ is that of all the processes in that zone, running in parallel. There has been *encapsulated* over read- and send-actions on internal ports, to enforce communications. *(See next eight pages.)*

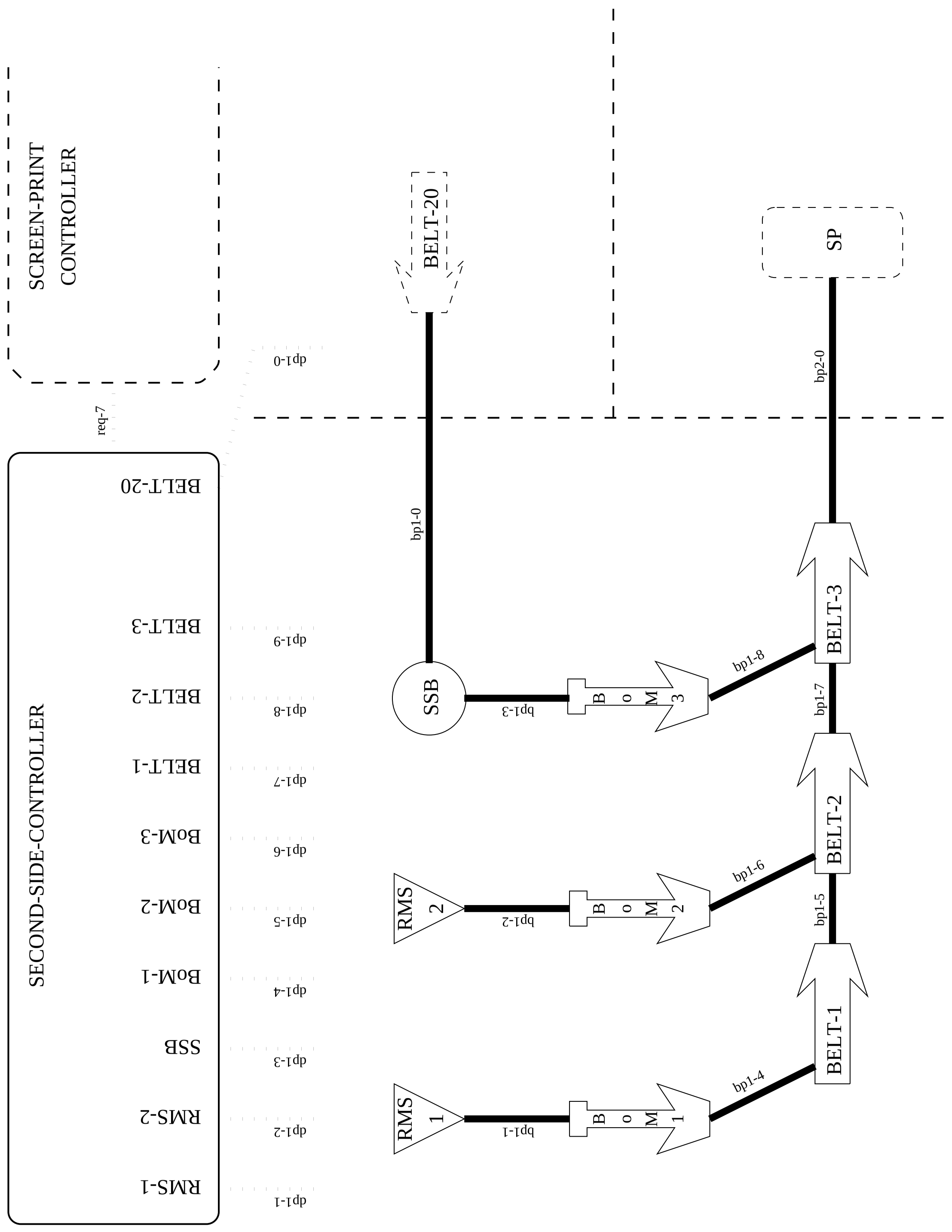


Figure 2: Control-zone-1

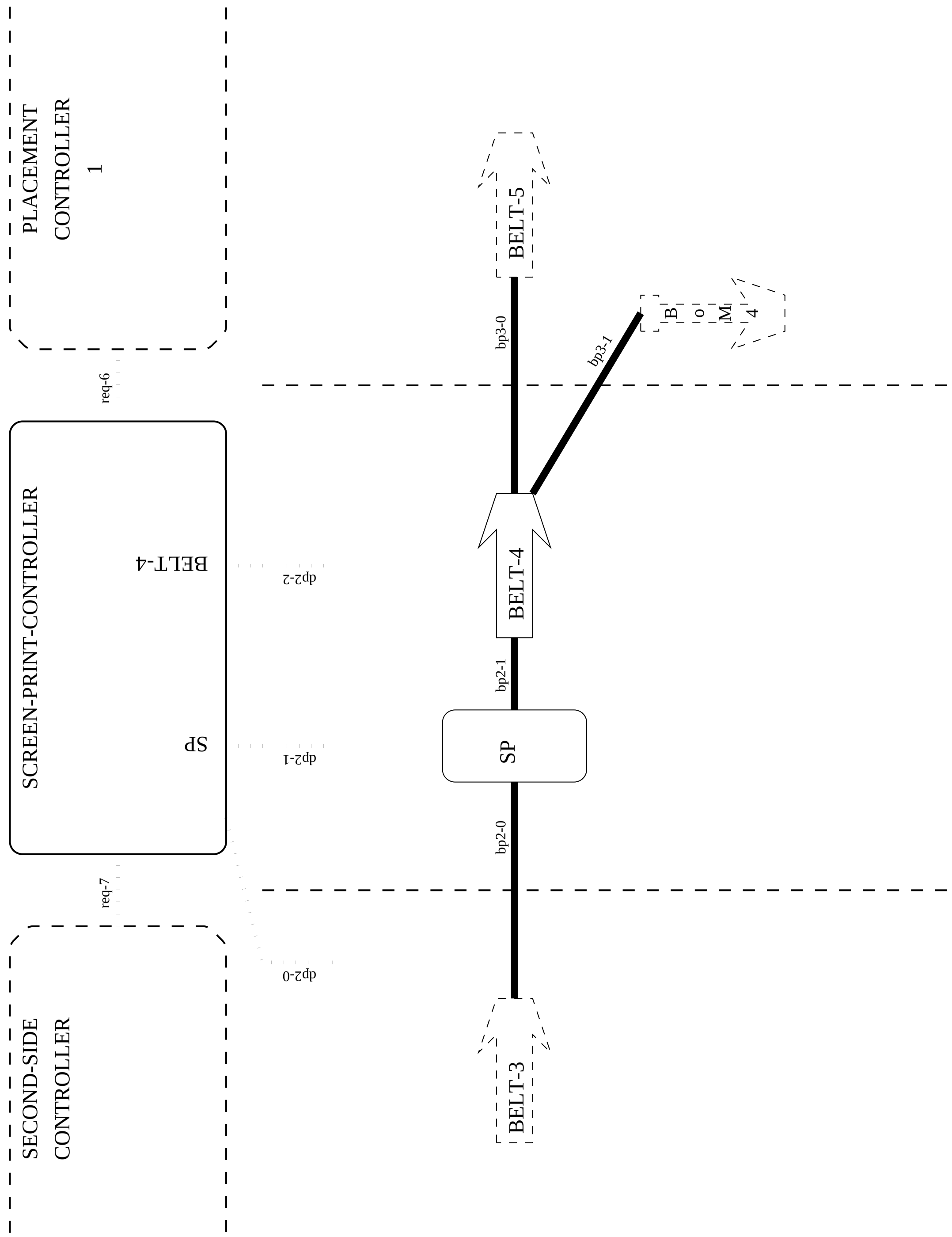


Figure 3: Control-zone-2

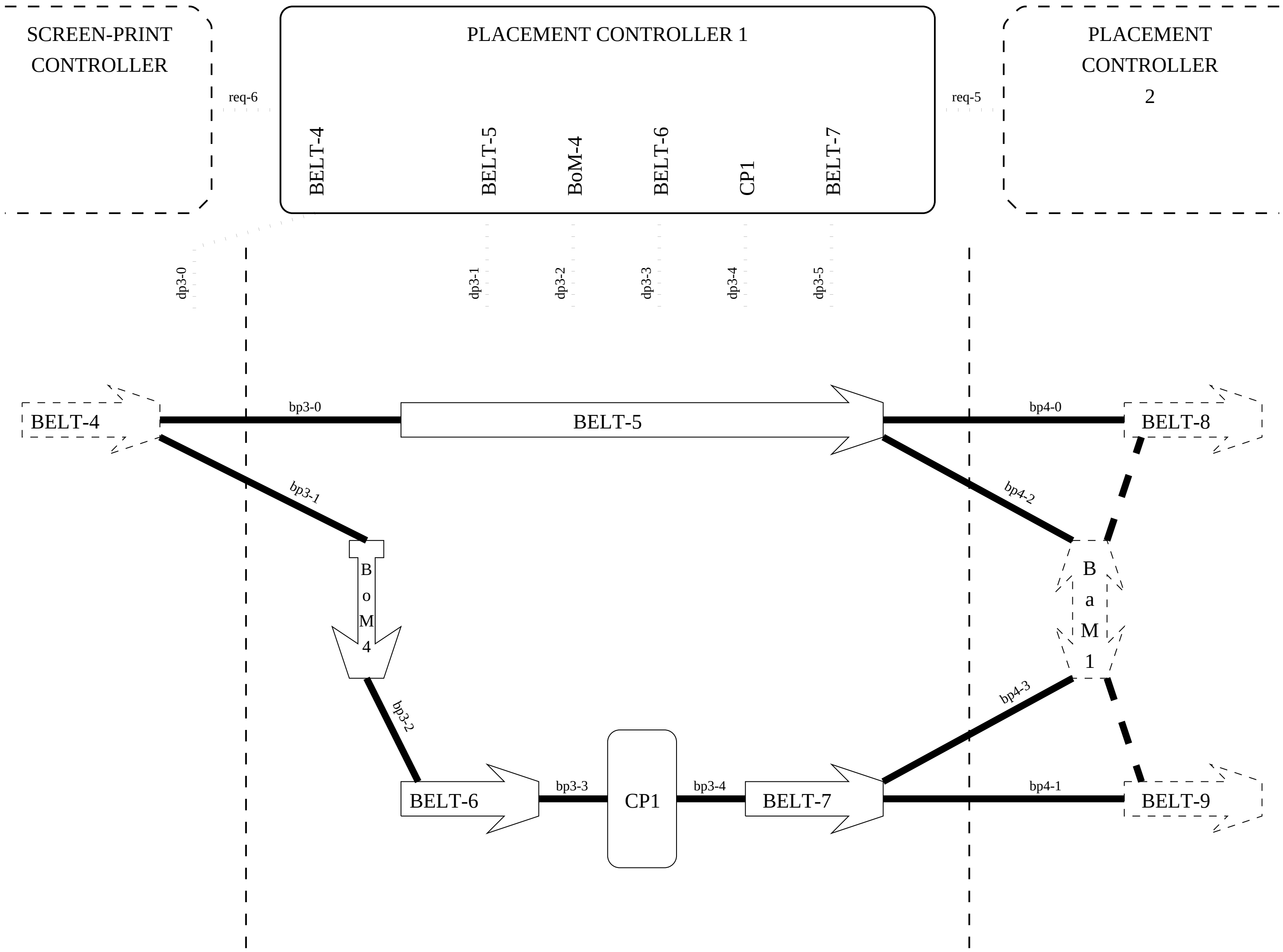


Figure 4: Control-zone-3

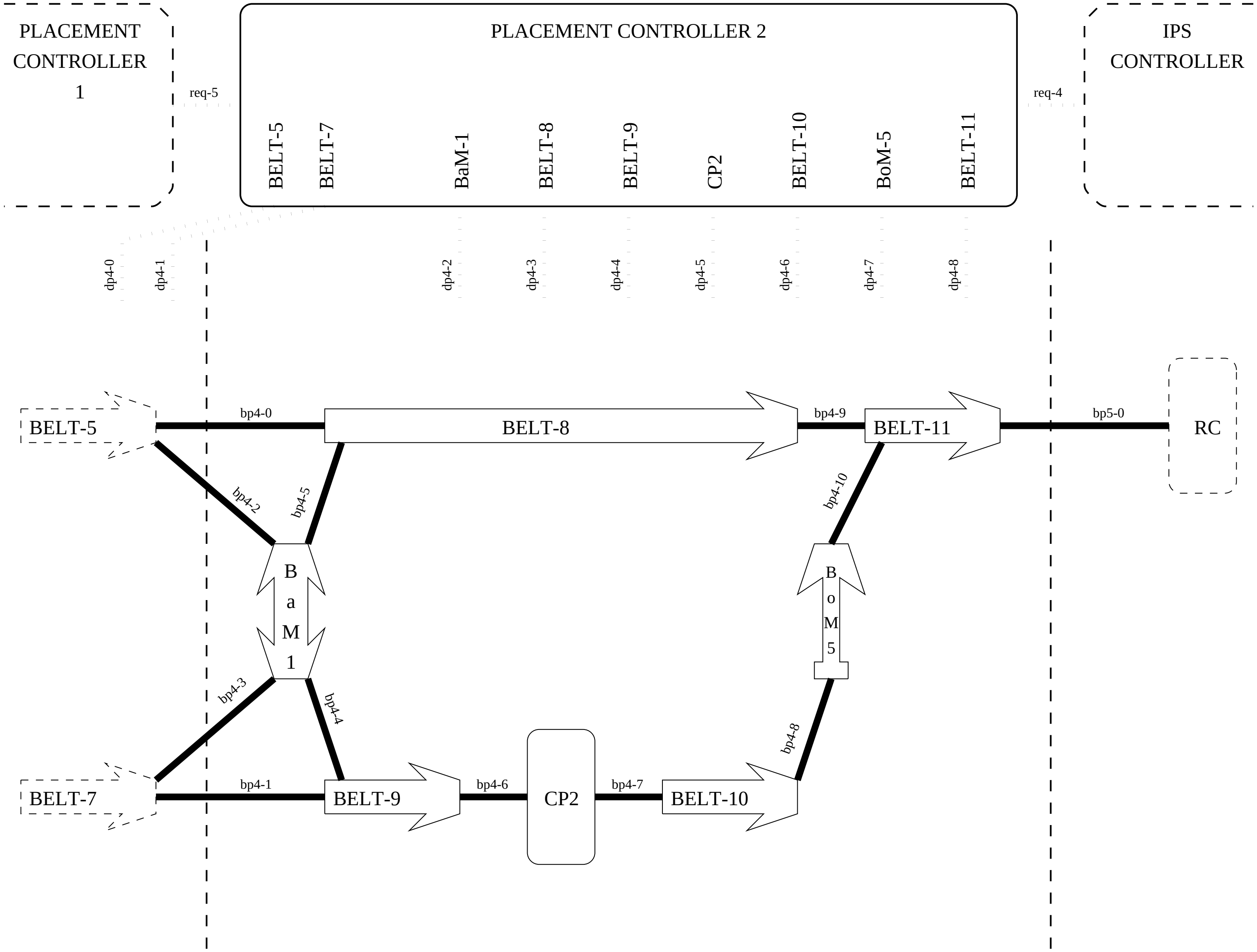


Figure 5: Control-zone-4

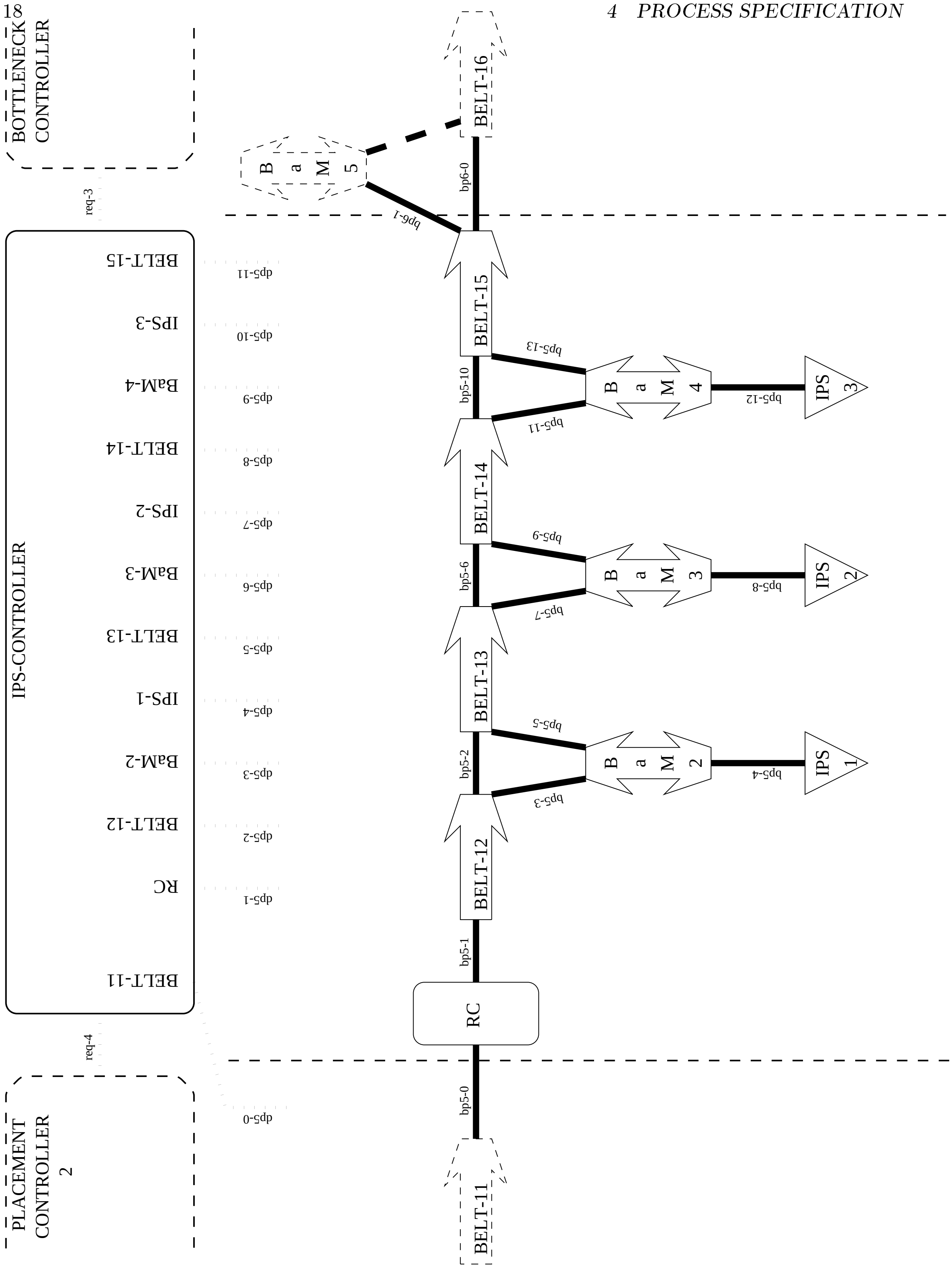


Figure 6: Control-zone-5

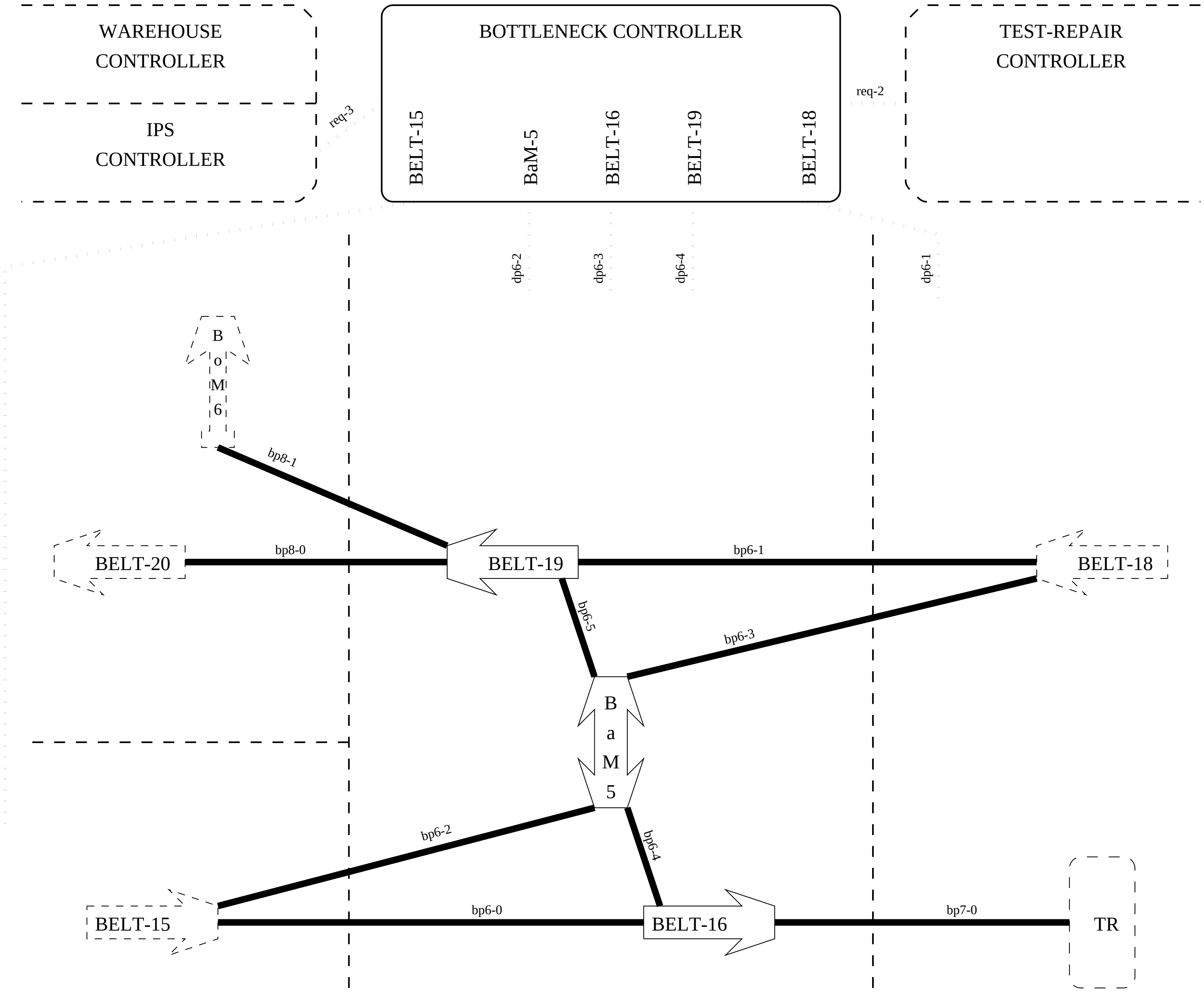


Figure 7: Control-zone-6

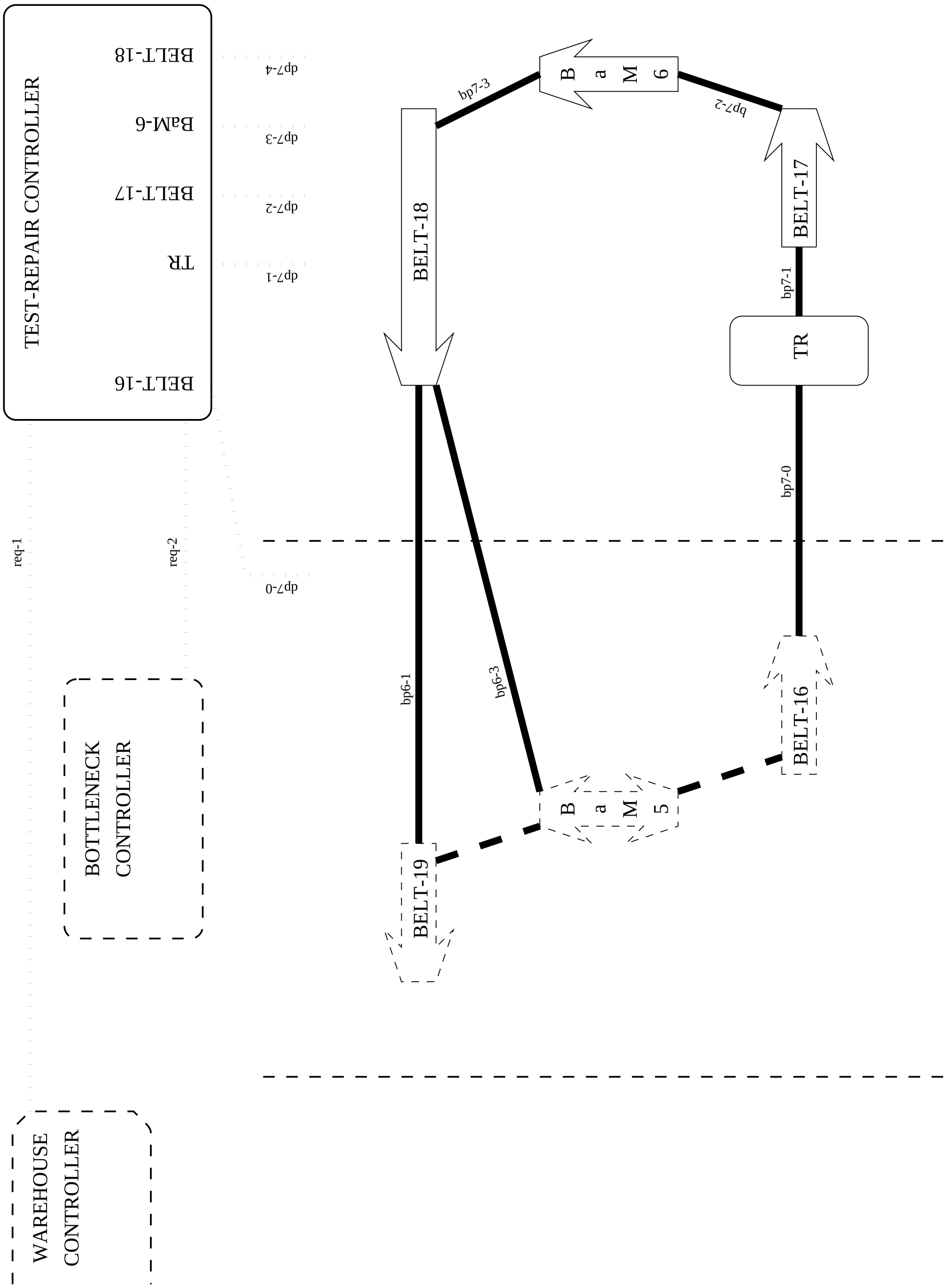


Figure 8: Control-zone-7

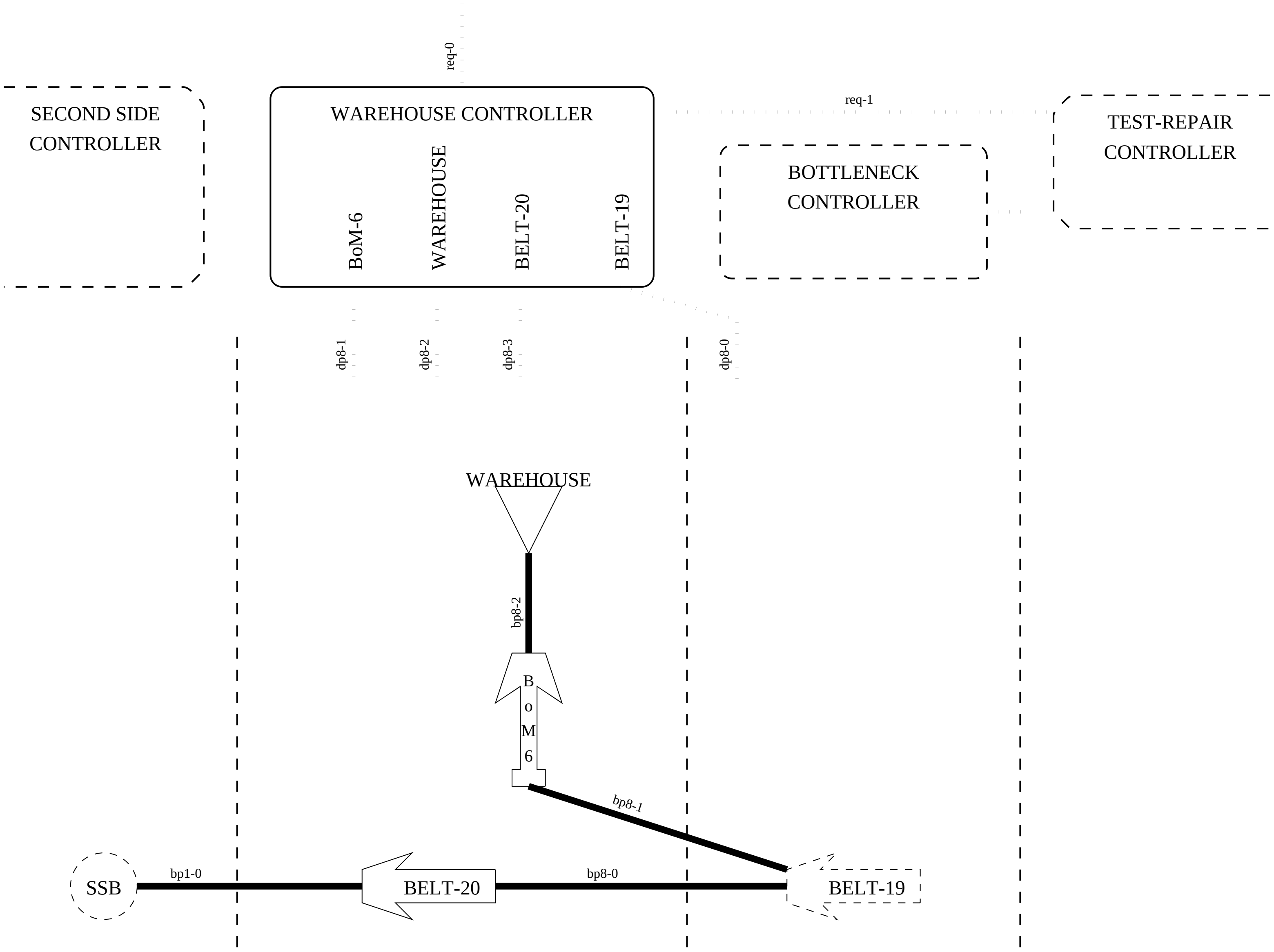


Figure 9: Control-zone-8

4.4 Specifying the model factory

The model-factory has been defined as the parallel working of the eight CZs, encapsulating over read- and send-actions on ports that cross the CZ-borders. There is now only one external port left. That is the one on which orders from the environment are received (req-0).

Finally we focus on the treatment of requests (recall that this is one of the tasks of a Controller). This is a pull-mechanism: an order comes in at the Warehouse-controller, then becomes transformed into a request, which is then communicated to the Test-repair-controller. The Test-repair-controller immediately communicates the request to the Bottleneck-controller, which on its turn communicates it to the Ips-controller. Here the request is taken out of the IPS and the Ips-controller communicates a request to the Placement-controller-2 to refill the IPS. Via the Placement-controller-1 and the Screen-print-controller, the request finally reaches the Second-side-controller, upon which a new batch is put on the belt. So in fact there are two "request-tracks": one from CZ-8 to CZ-5 and the other from CZ-5 to CZ-1.

In Figure 10 an example of the request flow is depicted.

1. Initially, the IPSs are empty. Three requests ($\triangle$) are sent by the Ips-controller to fill them. Meanwhile, a request ($\bullet$) for a batch with two products B has been sent from the Warehouse-controller
2. The IPSs have been filled, the request from the Warehouse-controller is now at the Ips-controller,
3. Two products B have been taken out of the IPS. The Ips-controller has sent a request ($\bigcirc$) to fill-up the IPS. Meanwhile, a new request ($\blacksquare$) for a batch with three products B has been sent by the Warehouse-controller,
4. The request from the Ips-controller is at the Second-side-controller. The request from the Warehouse-controller is at the Ips-controller. It has to wait there because there are no three products B in the IPS,
5. The IPS has been refilled. The request from the Warehouse-controller can now be dealt with
6. Three products B have been taken out of the IPS. Once more the Ips-controller has sent a request ($\square$) to fill-up the IPS.

5 Comments and directions for further research

5.1 Comments

Typical for our specification of the model factory is the attempt to obtain a flexible design of the basic parts of the factory. As for the modelling of the control structure, this is done in a distributed way based on synchronisations. As a consequence the heavy use of these synchronisations is not too restrictive.

About some features that are modelled in a way that would not match a final specification, we recall (in an arbitrarily order):

- The supply of components (see Section 2.2).
- The behaviour of the Raw Material Stores and the Warehouse (see Section 2.2, 4.1). Moreover, in the specification of the stores there is a mismatch between the argument representing the current stock and the ability to "give" the number of boards requested for. As the looping behaviour is anyhow a feature that should not be present in a final specification, we left this inconsistency in the present one for simplicity.
- The modelling of the Test Repair Station (see Section 2.2).

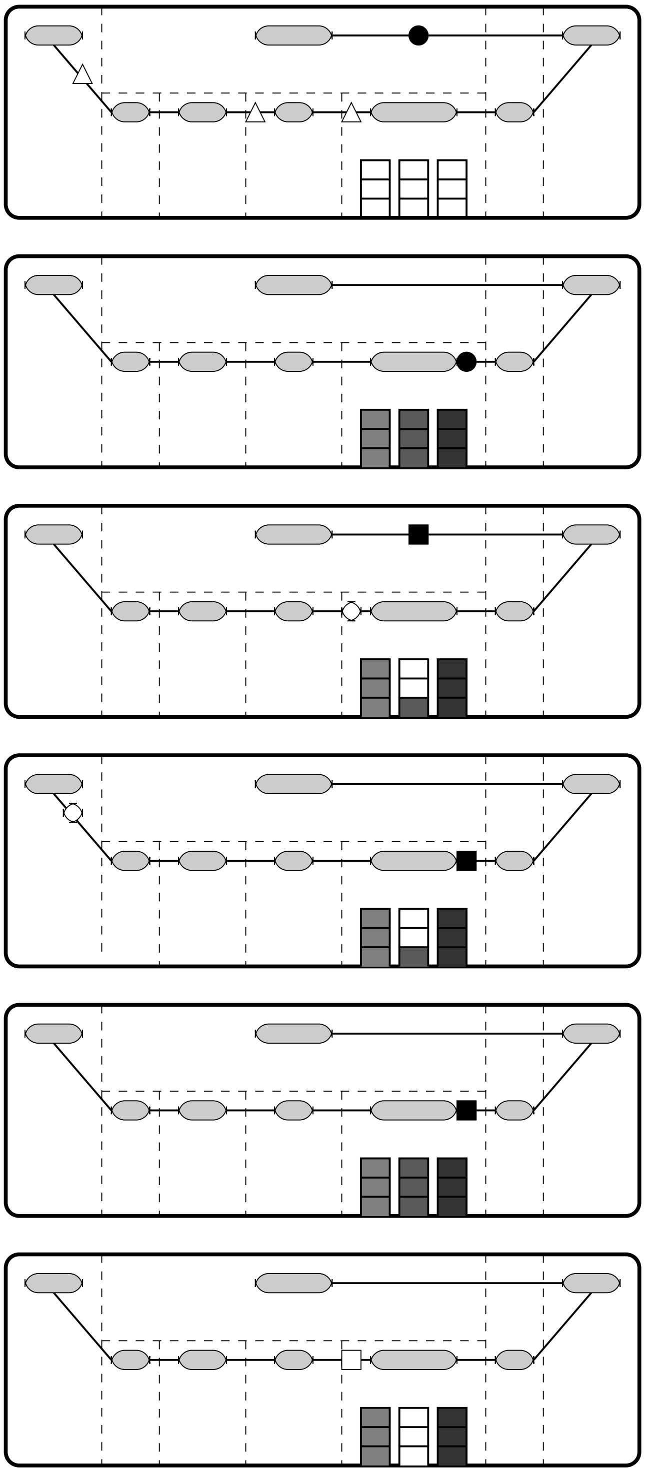


Figure 10: Flow of Requests in the model factory

Relating the present to modelling to the former one In [VlijWa92] each belt (and other transporting unit) was specified separately. In the present specification these are specified uniformly. Moreover, our predecessors only considered batches of size three, conform the current proto type implementation at DEC CEC. Last but not least, we added the definition of the control of the model factory to their specification.

5.2 Further research

Some directions for further research are the following:

- A better specification of the features not matching reality that were mentioned above.
- It may be a challenge (and a goal) to diminish synchronisations as much as possible while retaining the distributed character of the control structure.
- The request flow can be modelled along three request-lines (instead of one), facilitating a more fluent handling of requests at the In Process Stores.
- The activity performed by the workstations of the model factory are now modelled as "actions". It is very well possible to model these by processes instead, so that more complicated processing can be easily described.
- It may be desirable to have a "stop-and-restart" facility in the model factory, as to allow external maintenance. Therefore the specification of such a facility may be of interest.

6 Listings

6.1 The specification

(See next twelve pages.)

```
data module Naturals
begin
  exports
    begin
      sorts
        NAT
      functions
        0:                ->   NAT
        s:    NAT         ->   NAT
        add:  NAT # NAT   ->   NAT
        sub:  NAT # NAT   ->   NAT
    end
  variables
    x, y:   ->   NAT
  equations
    [N0]  add(x, 0) = x
    [N1]  add(x, s(y)) = s(add(x, y))
    [N2]  sub(x, 0) = x
    [N3]  sub(0, y) = 0
    [N4]  sub(s(x), s(y)) = sub(x, y)
end Naturals
```

```
data module Batch-sizes
begin
  exports
    begin
      sorts
        BATCH-SIZE
      functions
        zero:                ->   BATCH-SIZE
        one:                 ->   BATCH-SIZE
        two:                 ->   BATCH-SIZE
        three:               ->   BATCH-SIZE
        bn:    BATCH-SIZE    ->   NAT
        nb:    NAT           ->   BATCH-SIZE
    end
  imports
    Naturals
  variables
    x:   ->   NAT
  equations
    [BS0] bn(zero) = 0
    [BS1] bn(one) = s(0)
    [BS2] bn(two) = s(s(0))
    [BS3] bn(three) = s(s(s(0)))
    [BS4] nb(0) = zero
    [BS5] nb(s(0)) = one
    [BS6] nb(s(s(0))) = two
    [BS7] nb(s(s(s(0)))) = three
    [BS8] nb(s(s(s(s(x))))) = nb(s(s(s(x))))
end Batch-sizes
```

```
data module Loop-values
begin
  exports
    begin
      functions
        top:   ->   NAT
    end
  imports
    Naturals
  equations
    [I0] top = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))))))
end Loop-values
```

```
process module Loop-set
begin
  exports
    begin
      sets of NAT
        LOOP-SET = NAT \ {add(n, s(top)) | n in NAT}
    end
  imports Loop-values
end Loop-set
```

```
data module Product-types
begin
  exports
    begin
      sorts
        PRODUCT-TYPE
      functions
        A:   ->   PRODUCT-TYPE
        B:   ->   PRODUCT-TYPE
        C:   ->   PRODUCT-TYPE
    end
end Product-types
```

```
data module Boards
begin
  exports
    begin
      sorts
        BOARD-TYPE
      functions
        board1:   ->   BOARD-TYPE
        board2:   ->   BOARD-TYPE
    end
end Boards
```

```
data module Components
begin
  exports
    begin
      sorts
        COMPONENT
      functions
        red:      ->   COMPONENT
        yellow:   ->   COMPONENT
        green:    ->   COMPONENT
        emp:      ->   COMPONENT
    end
end Components



data module Printed-circuit-boards
begin
  exports
    begin
      sorts
        PCB
      functions
        pcb:     BOARD-TYPE #
                 COMPONENT # COMPONENT # COMPONENT #
                 COMPONENT # COMPONENT # COMPONENT   ->   PCB
        B-to-be: PCB                                 ->   PCB
        C-to-be: PCB                                 ->   PCB
        sp:      PCB                                 ->   PCB
        rc:      PCB                                 ->   PCB
      end
  imports
    Boards, Components
end Printed-circuit-boards



data module Pcb-prototypes
begin
  exports
    begin
      functions
        pcb-empty:   PRODUCT-TYPE   ->   PCB
        pcb-half:    PRODUCT-TYPE   ->   PCB
        pcb-fin:     PRODUCT-TYPE   ->   PCB
    end
  imports
    Product-types, Printed-circuit-boards
  equations
    [P0] pcb-empty(A) = pcb(board1, emp, emp, emp, emp, emp, emp)
    [P1] pcb-empty(B) = B-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))
    [P2] pcb-empty(C) = C-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))
    [P3] pcb-half(A) = pcb(board1, yellow, red, emp, red, emp, yellow)
    [P4] pcb-half(C) = pcb(board2, yellow, emp, yellow, emp, yellow, yellow)
    [P5] pcb-fin(A) = pcb(board1, yellow, red, green, red, green, yellow)
    [P6] pcb-fin(B) = pcb(board2, yellow, emp, yellow, red, red, red)
    [P7] pcb-fin(C) = pcb(board2, yellow, emp, yellow, green, yellow, yellow)
end Pcb-prototypes
```

```
data module Batches
begin
  exports
    begin
      sorts
        BATCH
      functions
        batch: PCB   #   BATCH-SIZE   ->   BATCH
        del:   BATCH                  ->   BATCH
        rep:   BATCH                  ->   BATCH
        ok:    BATCH                  ->   BATCH
        size:  BATCH                  ->   BATCH-SIZE
    end
  imports
    Printed-circuit-boards, Batch-sizes
  variables
    pcb:   ->   PCB
    bs:    ->   BATCH-SIZE
    b:     ->   BATCH
  equations
    [B0] size(batch(pcb, bs)) = bs
    [B1] size(del(b)) = size(b)
    [B2] size(rep(b)) = size(b)
    [B3] size(ok(b)) = size(b)
end Batches



data module Batch-status
begin
  exports
    begin
      sorts
        STATUS
      functions
        empty:   ->   STATUS
        half:    ->   STATUS
        fin:     ->   STATUS
        del:     ->   STATUS
        rep:     ->   STATUS
        ok:      ->   STATUS
    end
end Batch-status
```

```
data module Batch-ids
begin
  exports
    begin
      sorts
        BATCH-ID
      functions
        id:  PRODUCT-TYPE # BATCH-SIZE # STATUS   ->   BATCH-ID
        bid: BATCH                                ->   BATCH-ID
    end
  imports
    Pcb-prototypes, Batches, Batch-status
  functions
    prod: PCB   ->   PRODUCT-TYPE
    stat: PCB   ->   STATUS
  variables
    pcb:   ->   PCB
    bs:    ->   BATCH-SIZE
    pt:    ->   PRODUCT-TYPE
  equations
    [Bi0]  bid(batch(pcb, bs)) = id(prod(pcb), bs, stat(pcb))
    [Bi1]  bid(del(batch(pcb, bs))) = id(prod(pcb), bs, del)
    [Bi2]  bid(rep(batch(pcb, bs))) = id(prod(pcb), bs, rep)
    [Bi2]  bid(ok(batch(pcb, bs))) = id(prod(pcb), bs, ok)
    [Bi3]  prod(pcb(board1, emp, emp, emp, emp, emp, emp)) = A
    [Bi4]  prod(B-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))) = B
    [Bi5]  prod(C-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))) = C
    [Bi6]  prod(pcb(board1, yellow, red, emp, red, emp, yellow)) = A
    [Bi8]  prod(pcb(board2, yellow, emp, yellow, emp, yellow, yellow)) = C
    [Bi9]  prod(pcb(board1, yellow, red, green, red, green, yellow)) = A
    [Bi10] prod(pcb(board2, yellow, emp, yellow, red, red, red)) = B
    [Bi11] prod(pcb(board2, yellow, emp, yellow, green, yellow, yellow)) = C
    [Bi12] prod(sp(pcb)) = prod(pcb)
    [Bi13] prod(rc(pcb)) = prod(pcb)
    [Bi14] stat(pcb(board1, emp, emp, emp, emp, emp, emp)) = empty
    [Bi15] stat(B-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))) = empty
    [Bi16] stat(C-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))) = empty
    [Bi17] stat(pcb(board1, yellow, red, emp, red, emp, yellow)) = half
    [Bi18] stat(pcb(board2, yellow, emp, yellow, emp, yellow, yellow)) = half
    [Bi20] stat(pcb(board1, yellow, red, green, red, green, yellow)) = fin
    [Bi21] stat(pcb(board2, yellow, emp, yellow, red, red, red)) = fin
    [Bi22] stat(pcb(board2, yellow, emp, yellow, green, yellow, yellow)) = fin
    [Bi23] stat(sp(pcb)) = stat(pcb)
    [Bi24] stat(rc(pcb)) = stat(pcb)
end Batch-ids
```

```
data module Requests
begin
  exports
    begin
      sorts
        ORDER, REQUEST
      functions
        ord: PRODUCT-TYPE # NAT          ->   ORDER
        req: PRODUCT-TYPE # BATCH-SIZE   ->   REQUEST
    end
  imports
    Product-types, Batch-sizes
end Requests
```

```
data module Messages
begin
  exports
    begin
      sorts
        MESSAGE
      functions
        reset:              ->   MESSAGE
        av:                 ->   MESSAGE
        ho:    BATCH-ID     ->   MESSAGE
        go:                 ->   MESSAGE
        go-up:              ->   MESSAGE
        go-dn:              ->   MESSAGE
        go-mm:              ->   MESSAGE
        go-ms:              ->   MESSAGE
        go-sm:              ->   MESSAGE
        go-ss:              ->   MESSAGE
        pres:  NAT          ->   MESSAGE
        take:  REQUEST      ->   MESSAGE
        give:  REQUEST      ->   MESSAGE
    end
  imports
    Batch-ids, Requests
end Messages
```

```
data module Batch-operations
begin
  exports
    begin
      functions
        screen-print: BATCH      ->   BATCH
        comp-place-1: BATCH      ->   BATCH
        comp-place-2: BATCH      ->   BATCH
        reflow-clean: BATCH      ->   BATCH
        test-repair:  BATCH      ->   BATCH
        cp1:          PCB        ->   PCB
        cp2:          PCB        ->   PCB
    end
  imports
    Batches, Pcb-prototypes
  variables
    pcb:   ->   PCB
    bs:    ->   BATCH-SIZE
  equations
    [Bo0]  screen-print(batch(pcb, bs)) = batch(sp(pcb), bs)
    [Bo1]  comp-place-1(batch(pcb, bs)) = batch(cp1(pcb), bs)
    [Bo2]  comp-place-2(batch(pcb, bs)) = batch(cp2(pcb), bs)
    [Bo3]  reflow-clean(batch(pcb, bs)) = batch(rc(pcb), bs)
    [Bo4]  test-repair(del(batch(pcb, bs))) = rep(batch(pcb, bs))
    [Bo5]  test-repair(rep(batch(pcb, bs))) = ok(batch(pcb, bs))
    [Bo6]  cp1(sp(pcb(board1, emp, emp, emp, emp, emp, emp))) = pcb-half(A)
    [Bo7]  cp1(sp(B-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))))
           = pcb-fin(B)
    [Bo8]  cp2(pcb(board1, yellow, red, emp, red, emp, yellow)) = pcb-fin(A)
    [Bo9]  cp2(sp(C-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))))
           = pcb-half(C)
    [Bo10] cp2(sp(rc(pcb(board2, yellow, emp, yellow, emp, yellow, yellow))))
           = pcb-fin(C)
end Batch-operations
```

```
process module Station-actions
begin
  exports
    begin
      atoms
        print-screen: BATCH-ID
        placement-action-1: BATCH-ID
        placement-action-2: BATCH-ID
        reflow-and-clean: BATCH-ID
        test-and-repair: BATCH-ID
    end
  imports
    Batch-ids
end Station-actions
```

```
data module Ports
begin
  exports
    begin
      sorts
        DATA-PORT, BATCH-PORT
      functions
        dp1-0:    ->   DATA-PORT          bp1-0:    ->   BATCH-PORT
        dp1-1:    ->   DATA-PORT          bp1-1:    ->   BATCH-PORT
        dp1-2:    ->   DATA-PORT          bp1-2:    ->   BATCH-PORT
        dp1-3:    ->   DATA-PORT          bp1-3:    ->   BATCH-PORT
        dp1-4:    ->   DATA-PORT          bp1-4:    ->   BATCH-PORT
        dp1-5:    ->   DATA-PORT          bp1-5:    ->   BATCH-PORT
        dp1-6:    ->   DATA-PORT          bp1-6:    ->   BATCH-PORT
        dp1-7:    ->   DATA-PORT          bp1-7:    ->   BATCH-PORT
        dp1-8:    ->   DATA-PORT          bp1-8:    ->   BATCH-PORT
        dp1-9:    ->   DATA-PORT          bp2-0:    ->   BATCH-PORT
        dp2-0:    ->   DATA-PORT          bp2-1:    ->   BATCH-PORT
        dp2-1:    ->   DATA-PORT          bp3-0:    ->   BATCH-PORT
        dp2-2:    ->   DATA-PORT          bp3-1:    ->   BATCH-PORT
        dp3-0:    ->   DATA-PORT          bp3-2:    ->   BATCH-PORT
        dp3-1:    ->   DATA-PORT          bp3-3:    ->   BATCH-PORT
        dp3-2:    ->   DATA-PORT          bp3-4:    ->   BATCH-PORT
        dp3-3:    ->   DATA-PORT          bp4-0:    ->   BATCH-PORT
        dp3-4:    ->   DATA-PORT          bp4-1:    ->   BATCH-PORT
        dp3-5:    ->   DATA-PORT          bp4-2:    ->   BATCH-PORT
        dp4-0:    ->   DATA-PORT          bp4-3:    ->   BATCH-PORT
        dp4-1:    ->   DATA-PORT          bp4-4:    ->   BATCH-PORT
        dp4-2:    ->   DATA-PORT          bp4-5:    ->   BATCH-PORT
        dp4-3:    ->   DATA-PORT          bp4-6:    ->   BATCH-PORT
        dp4-4:    ->   DATA-PORT          bp4-7:    ->   BATCH-PORT
        dp4-5:    ->   DATA-PORT          bp4-8:    ->   BATCH-PORT
        dp4-6:    ->   DATA-PORT          bp4-9:    ->   BATCH-PORT
        dp4-7:    ->   DATA-PORT          bp4-10:   ->   BATCH-PORT
        dp4-8:    ->   DATA-PORT          bp5-0:    ->   BATCH-PORT
        dp5-0:    ->   DATA-PORT          bp5-1:    ->   BATCH-PORT
        dp5-1:    ->   DATA-PORT          bp5-2:    ->   BATCH-PORT
        dp5-2:    ->   DATA-PORT          bp5-3:    ->   BATCH-PORT
        dp5-3:    ->   DATA-PORT          bp5-4:    ->   BATCH-PORT
        dp5-4:    ->   DATA-PORT          bp5-5:    ->   BATCH-PORT
        dp5-5:    ->   DATA-PORT          bp5-6:    ->   BATCH-PORT
        dp5-6:    ->   DATA-PORT          bp5-7:    ->   BATCH-PORT
        dp5-7:    ->   DATA-PORT          bp5-8:    ->   BATCH-PORT
        dp5-8:    ->   DATA-PORT          bp5-9:    ->   BATCH-PORT
        dp5-9:    ->   DATA-PORT          bp5-10:   ->   BATCH-PORT
        dp5-10:   ->   DATA-PORT          bp5-11:   ->   BATCH-PORT
        dp5-11:   ->   DATA-PORT          bp5-12:   ->   BATCH-PORT
        dp6-0:    ->   DATA-PORT          bp5-13:   ->   BATCH-PORT
        dp6-1:    ->   DATA-PORT          bp6-0:    ->   BATCH-PORT
        dp6-2:    ->   DATA-PORT          bp6-1:    ->   BATCH-PORT
        dp6-3:    ->   DATA-PORT          bp6-2:    ->   BATCH-PORT
        dp6-4:    ->   DATA-PORT          bp6-3:    ->   BATCH-PORT
        dp7-0:    ->   DATA-PORT          bp6-4:    ->   BATCH-PORT
        dp7-1:    ->   DATA-PORT          bp6-5:    ->   BATCH-PORT
        dp7-2:    ->   DATA-PORT          bp7-0:    ->   BATCH-PORT
        dp7-3:    ->   DATA-PORT          bp7-1:    ->   BATCH-PORT
        dp7-4:    ->   DATA-PORT          bp7-2:    ->   BATCH-PORT
        dp8-0:    ->   DATA-PORT          bp7-3:    ->   BATCH-PORT
        dp8-1:    ->   DATA-PORT          bp8-0:    ->   BATCH-PORT
        dp8-2:    ->   DATA-PORT          bp8-1:    ->   BATCH-PORT
        dp8-3:    ->   DATA-PORT          bp8-2:    ->   BATCH-PORT
        req-0:    ->   DATA-PORT          disc:     ->   BATCH-PORT
        req-1:    ->   DATA-PORT
        req-2:    ->   DATA-PORT
        req-3:    ->   DATA-PORT
        req-4:    ->   DATA-PORT
        req-5:    ->   DATA-PORT
        req-6:    ->   DATA-PORT
        req-7:    ->   DATA-PORT
    end
end Ports
```

```
process module Communication
begin
  exports
    begin
      atoms
        rb: BATCH-PORT # BATCH
        sb: BATCH-PORT # BATCH
        cb: BATCH-PORT # BATCH
        rm: DATA-PORT  # MESSAGE
        sm: DATA-PORT  # MESSAGE
        cm: DATA-PORT  # MESSAGE
        rr: DATA-PORT  # REQUEST
        sr: DATA-PORT  # REQUEST
        cr: DATA-PORT  # REQUEST
        ro: DATA-PORT  # ORDER
    end
  imports
    Ports, Messages
  communications
    rb(bp, b) | sb(bp, b) = cb(bp, b)
      for bp in BATCH-PORT, b in BATCH
    rm(dp, m) | sm(dp, m) = cm(dp, m)
      for dp in DATA-PORT, m in MESSAGE
    rr(dp, r) | sr(dp, r) = cr(dp, r)
      for dp in DATA-PORT, r in REQUEST
end Communication
```

```
data module Process-ids
begin
  exports
    begin
      sorts
        PROCESS-ID
      functions
        pid1:   ->   PROCESS-ID
        pid2:   ->   PROCESS-ID
        pid3:   ->   PROCESS-ID
        pid4:   ->   PROCESS-ID
        pid5:   ->   PROCESS-ID
    end
end Process-ids
```

```
process module Belts
begin
  exports
    begin
      processes
        BELT: BATCH-PORT # BATCH-PORT # BATCH-PORT # BATCH-PORT #
              DATA-PORT # DATA-PORT
    end
  imports
    Communication
  atoms
    transport-belt: BATCH-PORT # BATCH-PORT
  variables
    batch-in-m, batch-in-s,
    batch-out-m, batch-out-s:   ->   BATCH-PORT
    contr-in, contr-out:        ->   DATA-PORT
  definitions
    BELT(batch-in-m, batch-in-s, batch-out-m, batch-out-s, contr-in, contr-out)
      = sm(contr-in, av) .
        ( rm(contr-in, go-mm) . sum(b in BATCH, rb(batch-in-m, b) .
          transport-belt(batch-in-m, batch-out-m) . sm(contr-out, ho(bid(b))) .
          sb(batch-out-m, b))
        + rm(contr-in, go-ms) . sum(b in BATCH, rb(batch-in-m, b) .
          transport-belt(batch-in-m, batch-out-s) . sm(contr-out, ho(bid(b))) .
          sb(batch-out-s, b))
        + rm(contr-in, go-sm) . sum(b in BATCH, rb(batch-in-s, b) .
          transport-belt(batch-in-s, batch-out-m) . sm(contr-out, ho(bid(b))) .
          sb(batch-out-m, b))
        + rm(contr-in, go-ss) . sum(b in BATCH, rb(batch-in-s, b) .
          transport-belt(batch-in-s, batch-out-s) . sm(contr-out, ho(bid(b))) .
          sb(batch-out-s, b))
        ) .
    BELT(batch-in-m, batch-in-s, batch-out-m, batch-out-s, contr-in, contr-out)
end Belts
```

```
process module Batch-movers
begin
  exports
    begin
      processes
        BATCH-MOVER: BATCH-PORT # BATCH-PORT # BATCH-PORT # BATCH-PORT #
                     DATA-PORT
    end
  imports
    Communication
  atoms
    move-up-batch: BATCH-PORT # BATCH-PORT
    move-dn-batch: BATCH-PORT # BATCH-PORT
  variables
    batch-in-top, batch-in-bot,
    batch-out-top, batch-out-bot:   ->   BATCH-PORT
    contr:                          ->   DATA-PORT
  definitions
    BATCH-MOVER(batch-in-top, batch-in-bot, batch-out-top, batch-out-bot, contr)
      = sm(contr, av) .
        ( rm(contr, go-up) . sum(b in BATCH, rb(batch-in-bot, b) .
          move-up-batch(batch-in-bot, batch-out-top) . sm(contr, ho(bid(b))) .
          sb(batch-out-top, b))
        + rm(contr, go-dn) . sum(b in BATCH, rb(batch-in-top, b) .
          move-dn-batch(batch-in-top, batch-out-bot) . sm(contr, ho(bid(b))) .
          sb(batch-out-bot, b))
        ) .
    BATCH-MOVER(batch-in-top, batch-in-bot, batch-out-top, batch-out-bot, contr)
end Batch-movers
```

```
process module Board-movers
begin
  exports
    begin
      processes
        BOARD-MOVER: BATCH-PORT # BATCH-PORT # DATA-PORT
    end
  imports
    Communication
  atoms
    move-pcb: BATCH-PORT # BATCH-PORT
  processes
    MOVE: BATCH-SIZE # BATCH-PORT # BATCH-PORT
  variables
    batch-in, batch-out, bp-in, bp-out:   ->   BATCH-PORT
    contr:                                ->   DATA-PORT
    pcb:                                  ->   PCB
    n:                                    ->   NAT
  definitions
    BOARD-MOVER(batch-in, batch-out, contr)
      = sm(contr, av) . rm(contr, go) . sum(b in BATCH, rb(batch-in, b).
        MOVE(size(b), batch-in, batch-out) .  sm(contr, ho(bid(b))) .
        sb(batch-out, b)) . BOARD-MOVER(batch-in, batch-out, contr)
    MOVE(one, bp-in, bp-out) = move-pcb(bp-in, bp-out)
    MOVE(two, bp-in, bp-out) = move-pcb(bp-in, bp-out) .
                               move-pcb(bp-in, bp-out)
    MOVE(three, bp-in, bp-out) = move-pcb(bp-in, bp-out) .
                                 move-pcb(bp-in, bp-out) .
                                 move-pcb(bp-in, bp-out)
end Board-movers
```

```
process module Workstations
begin
  parameters
    station-id
      begin
        functions
          k:   ->   PROCESS-ID
      end station-id,
    station-result
      begin
        functions
          proc: BATCH   ->   BATCH
      end station-result,
    station-action
      begin
        atoms
          action: BATCH-ID
      end station-action
  exports
    begin
      processes
        STATION: PROCESS-ID # BATCH-PORT # BATCH-PORT # DATA-PORT
    end
  imports
    Communication, Process-ids
  variables
    batch-in, batch-out:   ->   BATCH-PORT
    contr:                 ->   DATA-PORT
  definitions
    STATION(k, batch-in, batch-out, contr)
      = sm(contr, av) . rm(contr, go) . sum(b in BATCH, rb(batch-in, b) .
        action(bid(b)) . sm(contr, ho(bid(proc(b)))) . sb(batch-out, proc(b))) .
        STATION(k, batch-in, batch-out, contr)
end Workstations
```

```
process module Screen-printing-station
begin
  imports
    Workstations
     {station-id bound by
        [k   ->   pid1]
      to Process-ids
      station-result bound by
        [proc   ->   screen-print]
      to Batch-operations
      station-action bound by
        [action   ->   print-screen]
      to Station-actions}
end Screen-printing-station
```

```
process module Placement-station-1
begin
  imports
    Workstations
     {station-id bound by
        [k   ->   pid2]
      to Process-ids
      station-result bound by
        [proc   ->   comp-place-1]
      to Batch-operations
      station-action bound by
        [action   ->   placement-action-1]
      to Station-actions}
end Placement-station-1
```

```
process module Placement-station-2
begin
  imports
    Workstations
     {station-id bound by
        [k   ->   pid3]
      to Process-ids
      station-result bound by
        [proc   ->   comp-place-2]
      to Batch-operations
      station-action bound by
        [action   ->   placement-action-2]
      to Station-actions}
end Placement-station-2
```

```
process module Reflow-clean-station
begin
  imports
    Workstations
     {station-id bound by
        [k   ->   pid4]
      to Process-ids
      station-result bound by
        [proc   ->   reflow-clean]
      to Batch-operations
      station-action bound by
        [action   ->   reflow-and-clean]
      to Station-actions}
end Reflow-clean-station
```

```
process module Test-repair-station
begin
  imports
    Workstations
     {station-id bound by
        [k   ->   pid5]
      to Process-ids
      station-result bound by
        [proc   ->   test-repair]
      to Batch-operations
      station-action bound by
        [action   ->   test-and-repair]
      to Station-actions}
end Test-repair-station
```

```
process module In-process-stores
begin
  exports
    begin
      processes
        IPS: PRODUCT-TYPE # BATCH-PORT # DATA-PORT
    end
  imports
    Communication
  processes
    IPS: PRODUCT-TYPE # BATCH-PORT # DATA-PORT # BATCH-SIZE
  variables
    store-item:   ->   PRODUCT-TYPE
    batch-io:     ->   BATCH-PORT
    contr:        ->   DATA-PORT
    n:            ->   BATCH-SIZE
  definitions
    IPS(store-item, batch-io, contr) = IPS(store-item, batch-io, contr, zero)
    IPS(store-item, batch-io, contr, n)
      = sm(contr, pres(bn(n))) . sum(bs in BATCH-SIZE,
          rm(contr, take(req(store-item, bs))) .
          rb(batch-io, batch(rc(pcb-fin(store-item)), bs)) .
          IPS(store-item, batch-io, contr, nb(add(bn(bs), bn(n))))
        + rm(contr, give(req(store-item, bs))) .
          sm(contr, ho(id(store-item, bs, del))) .
          sb(batch-io, del(batch(rc(pcb-fin(store-item)), bs))) .
          IPS(store-item, batch-io, contr, nb(sub(bn(n), bn(bs)))) )
end In-process-stores
```

```
process module Raw-material-stores
begin
  exports
    begin
      processes
        RMS: BATCH-PORT # DATA-PORT
    end
  imports
    Communication, Loop-values
  processes
    RMS: BATCH-PORT # DATA-PORT # NAT
  variables
    batch-io:   ->   BATCH-PORT
    contr:      ->   DATA-PORT
    n:          ->   NAT
  definitions
    RMS(batch-io, contr) = RMS(batch-io, contr, top)
    RMS(batch-io, contr, 0) = RMS(batch-io, contr, top)
    RMS(batch-io, contr, s(n))
      = sm(contr, pres(s(n))) . sum(bs in BATCH-SIZE,
        sum(pt in PRODUCT-TYPE, rm(contr, give(req(pt, bs))) .
          sm(contr, ho(id(pt, bs, empty))) .
          sb(batch-io, batch(pcb-empty(pt), bs)) ) .
        RMS(batch-io, contr, sub(s(n), bn(bs))) )
end Raw-material-stores
```

```
process module Warehouse
begin
  exports
    begin
      processes
        WAREHOUSE
    end
  imports
    Communication
  processes
    WAREHOUSE: NAT
  variables
    n:   ->   NAT
  definitions
    WAREHOUSE = WAREHOUSE(0)
    WAREHOUSE(n)
      = sm(dp8-2, pres(n)) . sum(bs in BATCH-SIZE,
        sum(pt in PRODUCT-TYPE, rm(dp8-2, take(req(pt, bs))) .
        rb(bp8-2, ok(batch(rc(pcb-fin(pt)), bs)))) . WAREHOUSE(add(n, bn(bs))))
end Warehouse


-- The recursion on WAREHOUSE(n) should of course be
-- WAREHOUSE(add(n, bn(bs))), but for non-deadlocking
-- simulations this is replaced by WAREHOUSE(n).
```

```
process module Second-side-buffer
begin
  exports
    begin
      processes
        SSB
    end
  imports
    Communication
  definitions
    SSB = sm(dp1-3, av) .
          ( rm(dp1-3, reset)
          + rm(dp1-3, go) . sum(bs in BATCH-SIZE,
            rb(bp1-0, batch(rc(pcb-half(C)), bs)) .
            sm(dp1-3, ho(id(C, bs, half))) .
            sb(bp1-3, batch(rc(pcb-half(C)), bs)) )
          ) . SSB
end Second-side-buffer
```

```
process module Second-side-controller
begin
  exports
    begin
      processes
        SECOND-SIDE-CONTROLLER
    end
  imports
    Communication, Loop-set
  processes
    C1-0 C1-1 C1-2 C1-3 C1-4 C1-5 C1-6 C1-7 C1-8 C1-9
  definitions
    SECOND-SIDE-CONTROLLER = (C1-0 || C1-1 || C1-2 || C1-3 || C1-4 ||
                              C1-5 || C1-6 || C1-7 || C1-8 || C1-9)

    C1-0 = sum(bid in BATCH-ID, rm(dp1-0, ho(bid))) .
           rm(dp1-3, av) . sm(dp1-3, go) . C1-0
    C1-1 = sum(bid in BATCH-ID, rm(dp1-1, ho(bid))) .
           rm(dp1-4, av) . sm(dp1-4, go) . C1-1
    C1-2 = sum(bid in BATCH-ID, rm(dp1-2, ho(bid))) .
           rm(dp1-5, av) . sm(dp1-5, go) . C1-2
    C1-3 = sum(bid in BATCH-ID, rm(dp1-3, ho(bid))) .
           rm(dp1-6, av) . sm(dp1-6, go) . C1-3
    C1-4 = sum(bid in BATCH-ID, rm(dp1-4, ho(bid))) .
           rm(dp1-7, av) . sm(dp1-7, go-sm) . C1-4
    C1-5 = sum(bid in BATCH-ID, rm(dp1-5, ho(bid))) .
           rm(dp1-8, av) . sm(dp1-8, go-sm) . C1-5
    C1-6 = sum(bid in BATCH-ID, rm(dp1-6, ho(bid))) .
           rm(dp1-9, av) . sm(dp1-9, go-sm) . C1-6
    C1-7 = sum(bid in BATCH-ID, rm(dp1-7, ho(bid))) .
           rm(dp1-8, av) . sm(dp1-8, go-mm) . C1-7
    C1-8 = sum(bid in BATCH-ID, rm(dp1-8, ho(bid))) .
           rm(dp1-9, av) . sm(dp1-9, go-mm) . C1-8
    C1-9 = sum(bs in BATCH-SIZE,
             rr(req-7, req(A, bs)) . rm(dp1-3, av) . sum(n in LOOP-SET,
             rm(dp1-1, pres(n))) . sm(dp1-1, give(req(A, bs))) .
             sm(dp1-3, reset)
           + rr(req-7, req(B, bs)) . rm(dp1-3, av) . sum(n in LOOP-SET,
             rm(dp1-2, pres(n))) . sm(dp1-2, give(req(B, bs))) .
             sm(dp1-3, reset)
           + rr(req-7, req(C, bs)) . rm(dp1-3, av) . sum(n in LOOP-SET,
             rm(dp1-2, pres(n))) . sm(dp1-2, give(req(C, bs))) .
             sm(dp1-3, reset)
           ) . C1-9
end Second-side-controller
```

```
process module Screen-print-controller
begin
  exports
    begin
      processes
        SCREEN-PRINT-CONTROLLER
    end
  imports
    Communication
  processes
    C2-0 C2-1 C2-2
  definitions
    SCREEN-PRINT-CONTROLLER = (C2-0 || C2-1 || C2-2)
    C2-0 = sum(bid in BATCH-ID, rm(dp2-0, ho(bid))) .
           rm(dp2-1, av) . sm(dp2-1, go) . C2-0
    C2-1 = sum(bs in BATCH-SIZE,
             (rm(dp2-1, ho(id(A, bs, empty))) +
             rm(dp2-1, ho(id(B, bs, empty)))) .
             rm(dp2-2, av) . sm(dp2-2, go-ms)
           + (rm(dp2-1, ho(id(C, bs, empty))) +
             rm(dp2-1, ho(id(C, bs, half)))) .
             rm(dp2-2, av) . sm(dp2-2, go-mm)
           ) . C2-1
    C2-2 = sum(req in REQUEST, rr(req-6, req) . sr(req-7, req)) . C2-2
end Screen-print-controller
```

```
process module Placement-controller-1
begin
  exports
    begin
      processes
        PLACEMENT-CONTROLLER-1
    end
  imports
    Communication
  processes
    C3-0 C3-1 C3-2 C3-3 C3-4
  definitions
    PLACEMENT-CONTROLLER-1 = (C3-0 || C3-1 || C3-2 || C3-3 || C3-4)
    C3-0 = sum(bs in BATCH-SIZE,
             (rm(dp3-0, ho(id(A, bs, empty))) +
             rm(dp3-0, ho(id(B, bs, empty)))) .
             rm(dp3-2, av) . sm(dp3-2, go)
           + (rm(dp3-0, ho(id(C, bs, empty))) +
             rm(dp3-0, ho(id(C, bs, half)))) .
             rm(dp3-1, av) . sm(dp3-1, go-ms)
           ) . C3-0
    C3-1 = sum(bid in BATCH-ID, rm(dp3-2, ho(bid))) .
           rm(dp3-3, av) . sm(dp3-3, go-sm) . C3-1
    C3-2 = sum(bid in BATCH-ID, rm(dp3-3, ho(bid))) .
           rm(dp3-4, av) . sm(dp3-4, go) . C3-2
    C3-3 = sum( bs in BATCH-SIZE,
             rm(dp3-4, ho(id(A, bs, half))) . rm(dp3-5, av) . sm(dp3-5, go-mm)
           + rm(dp3-4, ho(id(B, bs, fin))) . rm(dp3-5, av) . sm(dp3-5, go-ms)
           ) . C3-3
    C3-4 = sum(req in REQUEST, rr(req-5, req) . sr(req-6, req)) . C3-4
end Placement-controller-1
```

```
process module Placement-controller-2
begin
  exports
    begin
      processes
        PLACEMENT-CONTROLLER-2
    end
  imports
    Communication
  processes
    C4-0 C4-1 C4-2 C4-3 C4-4 C4-5 C4-6 C4-7 C4-8
  definitions
    PLACEMENT-CONTROLLER-2
      = (C4-0 || C4-1 || C4-2 || C4-3 || C4-4 || C4-5 || C4-6 || C4-7 || C4-8)
    C4-0 = sum(bid in BATCH-ID, rm(dp4-0, ho(bid))) .
           rm(dp4-2, av) . sm(dp4-2, go-dn) . C4-0
    C4-1 = sum(bs in BATCH-SIZE,
             rm(dp4-1, ho(id(A, bs, half))) . rm(dp4-4, av) . sm(dp4-4, go-mm)
           + rm(dp4-1, ho(id(B, bs, fin))) . rm(dp4-2, av) . sm(dp4-2, go-up)
           ) . C4-1
    C4-2 = sum(bs in BATCH-SIZE,
             rm(dp4-2, ho(id(B, bs, fin))) . rm(dp4-3, av) . sm(dp4-3, go-sm)
           + (rm(dp4-2, ho(id(C, bs, empty))) +
             rm(dp4-2, ho(id(C, bs, half)))) .
             rm(dp4-4, av) . sm(dp4-4, go-sm)
           ) . C4-2
    C4-3 = sum(bid in BATCH-ID, rm(dp4-3, ho(bid))) .
           rm(dp4-8, av) . sm(dp4-8, go-mm) . C4-3
    C4-4 = sum(bid in BATCH-ID, rm(dp4-4, ho(bid))) .
           rm(dp4-5, av) . sm(dp4-5, go) . C4-4
    C4-5 = sum(bid in BATCH-ID, rm(dp4-5, ho(bid))) .
           rm(dp4-6, av) . sm(dp4-6, go-ms) . C4-5
    C4-6 = sum(bid in BATCH-ID, rm(dp4-6, ho(bid))) .
           rm(dp4-7, av) . sm(dp4-7, go) . C4-6
    C4-7 = sum(bid in BATCH-ID, rm(dp4-7, ho(bid))) .
           rm(dp4-8, av) . sm(dp4-8, go-sm) . C4-7
    C4-8 = sum(req in REQUEST, rr(req-4, req) . sr(req-5, req)) . C4-8
end Placement-controller-2


process module Ips-controller
begin
  exports
    begin
      processes
        IPS-CONTROLLER
    end
  imports
    Communication, Loop-set
  processes
    C5-0 C5-1 C5-2 C5-3 C5-4 C5-5 C5-6 C5-7 C5-8 C5-9 C5-10 C5-11 C5-12 C5-13
  definitions
    IPS-CONTROLLER =  sr(req-4, req(A, three)) .
                      sr(req-4, req(B, three)) .
                      sr(req-4, req(C, three)) .
                     (C5-0 || C5-1 || C5-2 || C5-3  || C5-4  || C5-5  || C5-6 ||
                      C5-7 || C5-8 || C5-9 || C5-10 || C5-11 || C5-12 || C5-13)

    C5-0  = sum(bid in BATCH-ID, rm(dp5-0, ho(bid))) .
            rm(dp5-1, av) . sm(dp5-1, go) . C5-0
    C5-1  = sum(bs in BATCH-SIZE,
              rm(dp5-1, ho(id(A, bs, fin))) . rm(dp5-2, av) . sm(dp5-2, go-ms)
            + (rm(dp5-1, ho(id(B, bs, fin))) + rm(dp5-1, ho(id(C, bs, fin))) +
              rm(dp5-1, ho(id(C, bs, half)))) . rm(dp5-2, av) . sm(dp5-2, go-mm)
            ) . C5-1
    C5-2  = sum(bs in BATCH-SIZE,
              rm(dp5-2, ho(id(A, bs, fin))) . rm(dp5-3, av) . sm(dp5-3, go-dn)
            + rm(dp5-2, ho(id(B, bs, fin))) . rm(dp5-5, av) . sm(dp5-5, go-ms)
            + (rm(dp5-2, ho(id(C, bs, fin))) + rm(dp5-2, ho(id(C, bs, half)))) .
              rm(dp5-5, av) . sm(dp5-5, go-mm)
            ) . C5-2
    C5-3  = sum(bs in BATCH-SIZE,
              rm(dp5-3, ho(id(A, bs, fin))) . sum(n in LOOP-SET,
              rm(dp5-4, pres(n))) . sm(dp5-4, take(req(A, bs)))
            + rm(dp5-3, ho(id(A, bs, del))) . rm(dp5-5, av) . sm(dp5-5, go-sm)
            ) . C5-3
    C5-4  = sum(bs in BATCH-SIZE, rm(dp5-4, ho(id(A, bs, del)))) .
            rm(dp5-3, av) . sm(dp5-3, go-up) . C5-4
    C5-5  = sum(bs in BATCH-SIZE,
              rm(dp5-5, ho(id(B, bs, fin))) . rm(dp5-6, av) . sm(dp5-6, go-dn)
            + rm(dp5-5, ho(id(C, bs, fin))) . rm(dp5-8, av) . sm(dp5-8, go-ms)
            + (rm(dp5-5, ho(id(A, bs, del))) + rm(dp5-5, ho(id(C, bs, half)))) .
              rm(dp5-8, av) . sm(dp5-8, go-mm)
            ) . C5-5
    C5-6  = sum(bs in BATCH-SIZE,
              rm(dp5-6, ho(id(B, bs, fin))) . sum(n in LOOP-SET,
              rm(dp5-7, pres(n))) . sm(dp5-7, take(req(B, bs)))
            + rm(dp5-6, ho(id(B, bs, del))) . rm(dp5-8, av) . sm(dp5-8, go-sm)
            ) . C5-6
    C5-7  = sum(bs in BATCH-SIZE, rm(dp5-7, ho(id(B, bs, del)))) .
            rm(dp5-6, av) . sm(dp5-6, go-up) . C5-7
    C5-8  = sum(bs in BATCH-SIZE,
              rm(dp5-8, ho(id(C, bs, fin))) . rm(dp5-9, av) . sm(dp5-9, go-dn)
            + rm(dp5-8, ho(id(C, bs, half))) . rm(dp5-11, av) .
              sm(dp5-11, go-ms)
            + (rm(dp5-8, ho(id(A, bs, del))) + rm(dp5-8, ho(id(B, bs, del)))) .
              rm(dp5-11, av) . sm(dp5-11, go-mm)
            ) . C5-8
    C5-9  = sum(bs in BATCH-SIZE,
              rm(dp5-9, ho(id(C, bs, fin))) . sum(n in LOOP-SET,
              rm(dp5-10, pres(n))) . sm(dp5-10, take(req(C, bs)))
            + rm(dp5-9, ho(id(C, bs, del))) . rm(dp5-11, av) . sm(dp5-11, go-sm)
            ) . C5-9
    C5-10 = sum(bs in BATCH-SIZE, rm(dp5-10, ho(id(C, bs, del)))) .
            rm(dp5-9, av) . sm(dp5-9, go-up) . C5-10
    C5-11 = sum(bs in BATCH-SIZE, rr(req-3, req(A, bs)) .
            (rm(dp5-4, pres(bn(bs))) + rm(dp5-4, pres(s(bn(bs)))) +
            rm(dp5-4, pres(s(s(bn(bs)))))) . sm(dp5-4, give(req(A, bs))) .
            sr(req-4, req(A, bs)) ) . C5-11
    C5-12 = sum(bs in BATCH-SIZE, rr(req-3, req(B, bs)) .
            (rm(dp5-7, pres(bn(bs))) + rm(dp5-7, pres(s(bn(bs)))) +
            rm(dp5-7, pres(s(s(bn(bs)))))) . sm(dp5-7, give(req(B, bs))) .
            sr(req-4, req(B, bs)) ) . C5-12
    C5-13 = sum(bs in BATCH-SIZE, rr(req-3, req(C, bs)) .
            (rm(dp5-10, pres(bn(bs))) + rm(dp5-10, pres(s(bn(bs)))) +
            rm(dp5-10, pres(s(s(bn(bs)))))) . sm(dp5-10, give(req(C, bs))) .
            sr(req-4, req(C, bs)) ) . C5-13
end Ips-controller
```

```
process module Bottleneck-controller
begin
  exports
    begin
      processes
        BOTTLENECK-CONTROLLER
    end
  imports
    Communication
  processes
    C6-0 C6-1 C6-2 C6-3
  definitions
    BOTTLENECK-CONTROLLER = (C6-0 || C6-1 || C6-2 || C6-3)
    C6-0 = sum(bs in BATCH-SIZE,
             rm(dp6-0, ho(id(C, bs, half))) . rm(dp6-2, av) . sm(dp6-2, go-up)
           + sum(pt in PRODUCT-TYPE, rm(dp6-0, ho(id(pt, bs, del)))) .
             rm(dp6-3, av) . sm(dp6-3, go-mm)) . C6-0
    C6-1 = sum(bs in BATCH-SIZE, sum(pt in PRODUCT-TYPE,
             rm(dp6-1, ho(id(pt, bs, rep))) . rm(dp6-2, av) . sm(dp6-2, go-dn)
           + rm(dp6-1, ho(id(pt, bs, ok))) . rm(dp6-4, av) . sm(dp6-4, go-ms)
           )) . C6-1
    C6-2 = sum(bs in BATCH-SIZE,
             rm(dp6-2, ho(id(C, bs, half))) . rm(dp6-4, av) . sm(dp6-4, go-sm)
           + sum(pt in PRODUCT-TYPE, rm(dp6-2, ho(id(pt, bs, rep)))) .
             rm(dp6-3, av) . sm(dp6-3, go-sm)) . C6-2
    C6-3 = sum(req in REQUEST, rr(req-2, req) . sr(req-3, req)) . C6-3
end Bottleneck-controller


process module Test-repair-controller
begin
  exports
    begin
      processes
        TEST-REPAIR-CONTROLLER
    end
  imports
    Communication
  processes
    C7-0 C7-1 C7-2 C7-3 C7-4
  definitions
    TEST-REPAIR-CONTROLLER = (C7-0 || C7-1 || C7-2 || C7-3 || C7-4)
    C7-0 = sum(bid in BATCH-ID, rm(dp7-0, ho(bid))) .
           rm(dp7-1, av) . sm(dp7-1, go) . C7-0
    C7-1 = sum(bid in BATCH-ID, rm(dp7-1, ho(bid))) .
           rm(dp7-2, av) . sm(dp7-2, go-ms) . C7-1
    C7-2 = sum(bid in BATCH-ID, rm(dp7-2, ho(bid))) .
           rm(dp7-3, av) . sm(dp7-3, go-up) . C7-2
    C7-3 = sum(bs in BATCH-SIZE, sum(pt in PRODUCT-TYPE,
             rm(dp7-3, ho(id(pt, bs, rep))) . rm(dp7-4, av) . sm(dp7-4, go-ss)
           + rm(dp7-3, ho(id(pt, bs, ok))) . rm(dp7-4, av) . sm(dp7-4, go-sm)
           )) . C7-3
    C7-4 = sum(req in REQUEST, rr(req-1, req) . sr(req-2, req)) . C7-4
end Test-repair-controller


process module Warehouse-controller
begin
  exports
    begin
      processes
        WAREHOUSE-CONTROLLER
    end
  imports
    Communication, Loop-set
  processes
    C8-0 C8-1 C8-2
    HANDLE: PRODUCT-TYPE # NAT
  variables
    n:    ->   NAT
    pt:   ->   PRODUCT-TYPE
  definitions
    WAREHOUSE-CONTROLLER = (C8-0 || C8-1 || C8-2)
    C8-0 = sum(bs in BATCH-SIZE, sum(pt in PRODUCT-TYPE,
               rm(dp8-0, ho(id(pt, bs, ok))) ) . rm(dp8-1, av) . sm(dp8-1, go)
           + rm(dp8-0, ho(id(C, bs, half))) . rm(dp8-3, av) . sm(dp8-3, go-ms)
           ) . C8-0
    C8-1 = sum(bs in BATCH-SIZE, sum(pt in PRODUCT-TYPE,
           rm(dp8-1, ho(id(pt, bs, ok))) . sum(n in LOOP-SET,
           rm(dp8-2, pres(n))) . sm(dp8-2, take(req(pt, bs))))) . C8-1
    C8-2 = sum(n in LOOP-SET, sum(pt in PRODUCT-TYPE,
           ro(req-0, ord(pt, n)) . HANDLE(pt, n))) . C8-2
    HANDLE(pt, 0)          = skip
    HANDLE(pt, s(0))       = sr(req-1, req(pt, nb(s(0))))
    HANDLE(pt, s(s(0)))    = sr(req-1, req(pt, nb(s(s(0)))))
    HANDLE(pt, s(s(s(n)))) = sr(req-1, req(pt, nb(s(s(s(0)))))) . HANDLE(pt, n)
end Warehouse-controller


process module Controll-zone-1
begin
  exports
    begin
      processes
        CZ-1
    end
  imports
    Raw-material-stores, Second-side-buffer,
    Belts, Board-movers, Second-side-controller
  processes
    RMS-1 RMS-2 BELT-1 BELT-2 BELT-3 BoM-1 BoM-2 BoM-3
  sets
    of BATCH-PORT
      BP-1 = { bp1-1, bp1-2, bp1-3, bp1-4, bp1-5, bp1-6, bp1-7, bp1-8, disc }
    of DATA-PORT
      DP-1 = { dp1-1, dp1-2, dp1-3, dp1-4, dp1-5, dp1-6, dp1-7, dp1-8, dp1-9 }
    of atoms
      H-1 =   {rb(bp, b), sb(bp, b) | bp in BP-1, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-1, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-1, r in REQUEST}
  definitions
    CZ-1 = encaps(H-1, RMS-1 || RMS-2 || SSB || BELT-1 || BELT-2 || BELT-3 ||
                       BoM-1 || BoM-2 || BoM-3 || SECOND-SIDE-CONTROLLER)
    RMS-1 = RMS(bp1-1, dp1-1)
    RMS-2 = RMS(bp1-2, dp1-2)
    BELT-1 = BELT(disc, bp1-4, bp1-5, disc, dp1-7, dp1-7)
    BELT-2 = BELT(bp1-5, bp1-6, bp1-7, disc, dp1-8, dp1-8)
    BELT-3 = BELT(bp1-7, bp1-8, bp2-0, disc, dp1-9, dp2-0)
    BoM-1 = BOARD-MOVER(bp1-1, bp1-4, dp1-4)
    BoM-2 = BOARD-MOVER(bp1-2, bp1-6, dp1-5)
    BoM-3 = BOARD-MOVER(bp1-3, bp1-8, dp1-6)
end Controll-zone-1
```

```
process module Controll-zone-2
begin
  exports
    begin
      processes
        CZ-2
    end
  imports
    Screen-printing-station, Belts, Screen-print-controller
  processes
    SP BELT-4
  sets
    of BATCH-PORT
      BP-2 = { bp2-1, disc }
    of DATA-PORT
      DP-2 = { dp2-1, dp2-2 }
    of atoms
      H-2 =   {rb(bp, b), sb(bp, b) | bp in BP-2, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-2, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-2, r in REQUEST}
  definitions
    CZ-2 = encaps(H-2, SP || BELT-4 || SCREEN-PRINT-CONTROLLER)
    SP = STATION(pid1, bp2-0, bp2-1, dp2-1)
    BELT-4 = BELT(bp2-1, disc, bp3-0, bp3-1, dp2-2, dp3-0)
end Controll-zone-2

process module Controll-zone-3
begin
  exports
    begin
      processes
        CZ-3
    end
  imports
    Placement-station-1, Belts, Board-movers, Placement-controller-1
  processes
    CP1 BELT-5 BELT-6 BELT-7 BoM-4
  sets
    of BATCH-PORT
      BP-3 = { bp3-2, bp3-3, bp3-4, disc }
    of DATA-PORT
      DP-3 = { dp3-1, dp3-2, dp3-3, dp3-4, dp3-5 }
    of atoms
      H-3 =   {rb(bp, b), sb(bp, b) | bp in BP-3, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-3, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-3, r in REQUEST}
  definitions
    CZ-3 = encaps(H-3, CP1 || BELT-5 || BELT-6 || BELT-7 || BoM-4 ||
                  PLACEMENT-CONTROLLER-1)
    CP1 = STATION(pid2, bp3-3, bp3-4, dp3-4)
    BELT-5 = BELT(bp3-0, disc, bp4-0, bp4-2, dp3-1, dp4-0)
    BELT-6 = BELT(disc, bp3-2, bp3-3, disc, dp3-3, dp3-3)
    BELT-7 = BELT(bp3-4, disc, bp4-1, bp4-3, dp3-5, dp4-1)
    BoM-4 = BOARD-MOVER(bp3-1, bp3-2, dp3-2)
end Controll-zone-3

process module Controll-zone-4
begin
  exports
    begin
      processes
        CZ-4
    end
  imports
    Placement-station-2, Belts, Batch-movers, Board-movers, Placement-controller-2
  processes
    CP2 BELT-8 BELT-9 BELT-10 BELT-11 BaM-1 BoM-5
  sets
    of BATCH-PORT
      BP-4 = { bp4-4, bp4-5, bp4-6, bp4-7, bp4-8, bp4-9, bp4-10, disc }
    of DATA-PORT
      DP-4 = { dp4-2, dp4-3, dp4-4, dp4-5, dp4-6, dp4-7, dp4-8 }
    of atoms
      H-4 =   {rb(bp, b), sb(bp, b) | bp in BP-4, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-4, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-4, r in REQUEST}
  definitions
    CZ-4 = encaps(H-4, CP2 || BELT-8 || BELT-9 || BELT-10 || BELT-11 ||
                  BaM-1 || BoM-5 || PLACEMENT-CONTROLLER-2)
    CP2 = STATION(pid3, bp4-6, bp4-7, dp4-5)
    BELT-8 = BELT(bp4-0, bp4-5, bp4-9, disc, dp4-3, dp4-3)
    BELT-9 = BELT(bp4-1, bp4-4, bp4-6, disc, dp4-4, dp4-4)
    BELT-10 = BELT(bp4-7, disc, disc, bp4-8, dp4-6, dp4-6)
    BELT-11 = BELT(bp4-9, bp4-10, bp5-0, disc, dp4-8, dp5-0)
    BaM-1 = BATCH-MOVER(bp4-2, bp4-3, bp4-5, bp4-4, dp4-2)
    BoM-5 = BOARD-MOVER(bp4-8, bp4-10, dp4-7)
end Controll-zone-4

process module Controll-zone-5
begin
  exports
    begin
      processes
        CZ-5
    end
  imports
    Reflow-clean-station, In-process-stores, Belts, Batch-movers, Ips-controller
  processes
    RC IPS-1 IPS-2 IPS-3 BELT-12 BELT-13 BELT-14 BELT-15 BaM-2 BaM-3 BaM-4
  sets
    of BATCH-PORT
      BP-5 = { bp5-1, bp5-2, bp5-3, bp5-4, bp5-5, bp5-6, bp5-7, bp5-8,
               bp5-9, bp5-10, bp5-11, bp5-12, bp5-13, disc }
    of DATA-PORT
      DP-5 = { dp5-1, dp5-2, dp5-3, dp5-4, dp5-5, dp5-6,
               dp5-7, dp5-8, dp5-9, dp5-10, dp5-11 }
    of atoms
      H-5 =   {rb(bp, b), sb(bp, b) | bp in BP-5, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-5, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-5, r in REQUEST}
  definitions
    CZ-5 = encaps(H-5, RC || IPS-1 || IPS-2 || IPS-3 || BELT-12 || BELT-13 ||
                  BELT-14 || BELT-15 || BaM-2 || BaM-3 || BaM-4 || IPS-CONTROLLER)
    RC = STATION(pid4, bp5-0, bp5-1, dp5-1)
    IPS-1 = IPS(A, bp5-4, dp5-4)             IPS-2 = IPS(B, bp5-8, dp5-7)
    IPS-3 = IPS(C, bp5-12, dp5-10)
    BELT-12 = BELT(bp5-1, disc, bp5-2, bp5-3, dp5-2, dp5-2)
    BELT-13 = BELT(bp5-2, bp5-5, bp5-6, bp5-7, dp5-5, dp5-5)
    BELT-14 = BELT(bp5-6, bp5-9, bp5-10, bp5-11, dp5-8, dp5-8)
    BELT-15 = BELT(bp5-10, bp5-13, bp6-0, bp6-2, dp5-11, dp6-0)
    BaM-2 = BATCH-MOVER(bp5-3, bp5-4, bp5-5, bp5-4, dp5-3)
    BaM-3 = BATCH-MOVER(bp5-7, bp5-8, bp5-9, bp5-8, dp5-6)
    BaM-4 = BATCH-MOVER(bp5-11, bp5-12, bp5-13, bp5-12, dp5-9)
end Controll-zone-5
```

```
process module Controll-zone-6
begin
  exports
    begin
      processes
        CZ-6
    end
  imports
    Belts, Batch-movers, Bottleneck-controller
  processes
    BELT-16 BELT-19 BaM-5
  sets
    of BATCH-PORT
      BP-6 = { bp6-4, bp6-5, disc }
    of DATA-PORT
      DP-6 = { dp6-2, dp6-3, dp6-4 }
    of atoms
      H-6 =   {rb(bp, b), sb(bp, b) | bp in BP-6, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-6, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-6, r in REQUEST}
  definitions
    CZ-6 = encaps(H-6, BELT-16 || BELT-19 || BaM-5 || BOTTLENECK-CONTROLLER)

    BELT-16 = BELT(bp6-0, bp6-4, bp7-0, disc, dp6-3, dp7-0)
    BELT-19 = BELT(bp6-1, bp6-5, bp8-0, bp8-1, dp6-4, dp8-0)
    BaM-5 = BATCH-MOVER(bp6-3, bp6-2, bp6-5, bp6-4, dp6-2)
end Controll-zone-6


process module Controll-zone-7
begin
  exports
    begin
      processes
        CZ-7
    end
  imports
    Test-repair-station, Belts, Batch-movers, Test-repair-controller
  processes
    TR BELT-17 BELT-18 BaM-6
  sets
    of BATCH-PORT
      BP-7 = { bp7-1, bp7-2, bp7-3, disc }
    of DATA-PORT
      DP-7 = { dp7-1, dp7-2, dp7-3, dp7-4 }
    of atoms
      H-7 =   {rb(bp, b), sb(bp, b) | bp in BP-7, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-7, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-7, r in REQUEST}
  definitions
    CZ-7 = encaps(H-7, TR || BELT-17 || BELT-18 || BaM-6 ||
                       TEST-REPAIR-CONTROLLER)

    TR = STATION(pid5, bp7-0, bp7-1, dp7-1)
    BELT-17 = BELT(bp7-1, disc, disc, bp7-2, dp7-2, dp7-2)
    BELT-18 = BELT(disc, bp7-3, bp6-1, bp6-3, dp7-4, dp6-1)
    BaM-6 = BATCH-MOVER(disc, bp7-2, bp7-3, disc, dp7-3)
end Controll-zone-7


process module Controll-zone-8
begin
  exports
    begin
      processes
        CZ-8
    end
  imports
    Warehouse, Belts, Board-movers, Warehouse-controller
  processes
    BELT-20 BoM-6
  sets
    of BATCH-PORT
      BP-8 = { bp8-2, disc }
    of DATA-PORT
      DP-8 = { dp8-1, dp8-2, dp8-3 }
    of atoms
      H-8 =   {rb(bp, b), sb(bp, b) | bp in BP-8, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-8, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-8, r in REQUEST}
  definitions
    CZ-8 = encaps(H-8, WAREHOUSE || BELT-20 || BoM-6 ||
                       WAREHOUSE-CONTROLLER)

    BELT-20 = BELT(bp8-0, disc, disc, bp1-0, dp8-3, dp1-0)
    BoM-6 = BOARD-MOVER(bp8-1, bp8-2, dp8-1)
end Controll-zone-8


process module Model-factory
begin
  exports
    begin
      processes
        MOD-FACT
    end
  imports
    Controll-zone-1, Controll-zone-2, Controll-zone-3, Controll-zone-4,
    Controll-zone-5, Controll-zone-6, Controll-zone-7, Controll-zone-8
  sets
    of BATCH-PORT
      BP = {bp1-0, bp2-0, bp3-0, bp3-1, bp4-0, bp4-1, bp4-2, bp4-3,
            bp5-0, bp6-0, bp6-1, bp6-2, bp6-3, bp7-0, bp8-0, bp8-1}
    of DATA-PORT
      DP = {dp1-0, dp2-0, dp3-0, dp4-0, dp4-1, dp5-0, dp6-0, dp6-1, dp7-0,
            dp8-0, req-1, req-2, req-3, req-4, req-5, req-6, req-7}
    of atoms
      H =   {rb(bp, b), sb(bp, b) | bp in BP, b in BATCH}
          + {rm(dp, m), sm(dp, m) | dp in DP, m in MESSAGE}
          + {rr(dp, r), sr(dp, r) | dp in DP, r in REQUEST}
  definitions
    MOD-FACT
      = encaps(H, CZ-1 || CZ-2 || CZ-3 || CZ-4 ||
                  CZ-5 || CZ-6 || CZ-7 || CZ-8)
end Model-factory
```

6.2 A trace

```
com.  cm(dp3-0, ho(id(A, one, empty)))
com.  cm(dp3-2, av)
com.  cm(dp3-2, go)
com.  cb(bp3-1, batch(sp(pcb(board1, emp, emp, emp, emp, emp, emp)), one))
atom  move-pcb(bp3-1, bp3-2)
com.  cm(dp3-2, ho(id(A, one, empty)))
com.  cm(dp3-3, av)
com.  cm(dp3-3, go-sm)
com.  cb(bp3-2, batch(sp(pcb(board1, emp, emp, emp, emp, emp, emp)), one))
atom  transport-belt(bp3-2, bp3-3)
com.  cm(dp3-3, ho(id(A, one, empty)))
com.  cm(dp3-4, av)
com.  cm(dp3-4, go)
com.  cb(bp3-3, batch(sp(pcb(board1, emp, emp, emp, emp, emp, emp)), one))
atom  placement-action-1(id(A, one, empty))
com.  cm(dp3-4, ho(id(A, one, half)))
com.  cm(dp3-5, av)
com.  cm(dp3-5, go-mm)
com.  cb(bp3-4, batch(pcb(board1, yellow, red, emp, red, emp, yellow), one))
atom  transport-belt(bp3-4, bp4-1)
com.  cm(dp4-1, ho(id(A, one, half)))
```

References

[Mau90] S. Mauw. Process algebra as a tool for the specification and verification of CIM–architectures. In J.C.M. Baeten, editor, *Applications of process algebra*, Cambridge Tracts in Theoretical Computer Science 17, pages 53–80. Cambridge University Press, 1990.

[MV89] S. Mauw and G.J. Veltink. An introduction to PSF_d. In J. Díaz and F. Orejas, editors, *Proceedings TAPSOFT 89, Vol. 2*, volume 352 of *Lecture Notes in Computer Science*, pages 272–285. Springer-Verlag, 1989.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[Vel91] G.J. Veltink. The PSF toolkit. Report P9107, University of Amsterdam, 1991.

[VlijWa92] S.F.M. van Vlijmen and A. van Waveren. An algebraic specification of a Model Factory. Report P9209, University of Amsterdam, 1992.