An Algebraic Specification of a Manufacturing System with Hierarchical Control

Joris Hillebrand

Programming Research Group, University of Amsterdam

Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Email: `joris@fwi.uva.nl`

Abstract

This paper presents a PSF specification of a fictitious manufacturing system. We also discuss the control of manufacturing systems in general. Finally, an attempt is made to compare the expressive power of PSF and the specification language ProcessTalk.

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

Contents

1 Introduction

Over the years, computers have become increasingly involved in industrial processes. This phenomenon has reduced the processing time of products and the manpower needed. But, at the same time, the controllability of the production process threatened to decrease. Although most machines and computers operated well, one could never be quite sure that such devices would continue to operate well and under all circumstances. Among many other techniques, formal specification methods have been developed to obtain a deeper understanding of the behaviour of systems, given their implementation. One of these formal specification methods is PSF, a Process Specification Formalism, developed at the University of Amsterdam ([MV90], see also [MV93]). Having process algebra as a building block, PSF makes it possible to give an exact description of how processes behave. Consequently, one of the supporting tools makes it possible to simulate the process that has been specified. In this way one has a method to test potential implementations before they are implemented. A specification language comparable to PSF is LOTOS, developed at the University of Twente ([Int89]).
Whereas PSF and LOTOS have an algebraic basis, ProcessTalk is an object-oriented specification language. ProcessTalk, developed at the Eindhoven University of Technology, is based on Smalltalk-80. It comes with a simulation utility ProcessTool ([RW90], [Wor91]). We wanted to compare the expressive power of PSF and ProcessTalk by means of a case study. Therefore, we decided to make a specification of an IC manufacturing system in PSF, that was given in ProcessTalk earlier ([Smi92]). During the project, however, it became clear that ProcessTalk could not be translated directly into PSF. The fundamentally different principles, according to which the two languages were designed, made such translation at least very hard. Another problem was the fact that some features of ProcessTalk, such as realtime, are not supported in PSF. We decided to continue the comparison of expressiveness of the languages by writing a new PSF specification of the IC manufacturing system. In order to concentrate fully on the principles of manufacturing systems and their control in general, we abstracted from details. As a consequence, resources and material were given symbolic names. The structure of the manufacturing system, as well as its composition and the control techniques, have been specified according to the principles of [Smi92]. This resulted in a specification that is more flexible than some of the prior PSF specifications of manufacturing systems, such as the ones in [vVvW92], [HP92], [PV93], and [BP93].
In the following section we will discuss the principles from [Smi92]. Section 3 gives a short introduction to the principles of PSF. Section 4 presents the resulting specification of the manufacturing system. In section 5 we will comment on the given specification and the differences between ProcessTalk and PSF. We will also discuss some features that have not yet been specified, but could be added in the future. Finally, in appendix A, one can find the complete PSF specification.

Acknowledgements. The author would like to thank Professor Rooda from the Eindhoven University of Technology for his cooperation with this project and his suggestions for relevant literature. Further thanks go to Bas van Vlijmen for helping set up the project, to Professor Baeten for introducing us to Professor Rooda, and to Professor Bergstra for his helpful suggestions and for proofreading this paper. Finally, the author thanks Bob Diertens and Sjouke Mauw for their help with the simulation tool for PSF.

2 Manufacturing systems

This section presents a brief explanation of the terminology that is used in [Smi92] and throughout this paper. We will also discuss some fundamental principles for the design of a manufacturing system. These principles are the basis of the specification that will be discussed later.

2.1 Classification of manufacturing systems

A manufacturing system is the part of a factory that takes care of the transformation of raw material into desired products. Other parts of a factory buy raw material from suppliers, and sell finished products to consumers. The manufacturing system is provided with a store, a transporter, several resources, and a controller. The controller takes care of the tuning between the different components of the manufacturing system. For every product that the manufacturing system is able to produce, there exists a recipe. The controller of the manufacturing system uses the recipe during the manufacturing process of the product. The recipe, as well as status information from the resources, is used to lead material along the resources. The resources of a manufacturing system can be actual machines, *leaf resources*, like a cutting machine or a screw-driving unit. Resources can also have their own controller, store, transporter, and resources. A resource is then called *expanded* and forms itself a manufacturing system. This way a hierarchical manufacturing system is created.
Recipes usually have the same structure as the manufacturing system. A manufacturing system with a single resource that can perform only one operation, will have recipes consisting of a single operation. More complex manufacturing systems have recipes that contain the sequential or parallel composition of subrecipes. The subrecipes may then contain assembling or transformation operations. One can classify manufacturing systems by the kind of recipes they have, and by the resource layout.
The resource layout determines the route that material can make along the resources. Figure 1 shows the basic layouts. It has been taken from [Smi92]. The boxes in the figure denote the resources.

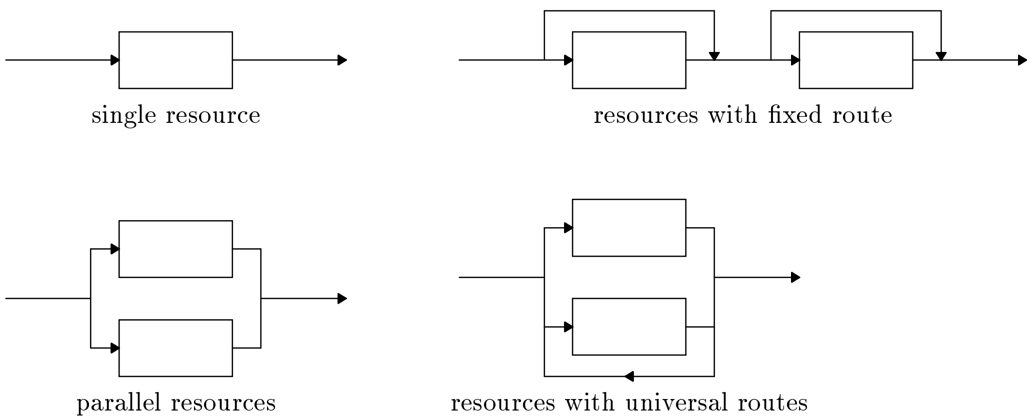


Figure 1: Basic resource layouts.

Manufacturing systems can now be placed in four categories:

1. A manufacturing system with a single resource and recipes with single operations, is called a **single shop**.
2. When a manufacturing system does not have a single resource, but its recipes contain only single operations, it is called a **parallel shop**. Each of the resources can perform the same operation.
3. When a manufacturing system has resources with a fixed route and recipes with multiple operations, it is a **flow shop**. The material has to pass more than one resource, but it must visit the resources in a fixed order.
4. A manufacturing system with resources with universal routes, and having recipes with multiple operations, is a **job shop**. The resources can be visited in arbitrary order.

The controller of a manufacturing system communicates with an expanded resource via the controller of the expanded resource. The controller of the manufacturing system is called the *supercontroller* of the controller of the expanded resource. So the latter receives messages from its supercontroller. The uppermost manufacturing controller has no supercontroller but communicates with consumers and producers. This controller is called the *factory controller.* It does not receive transformation jobs for the execution of operations, but it receives orders for the manufacturing of products. Because the factory controller has to negotiate with producers and consumers, it should not get overloaded by having to control too many resources directly. Therefore, a factory can best be a single shop (a manufacturing system with only one, possibly expanded, resource). The expanded resource then often is a flow shop or a job shop. The controller of the expanded resource splits the incoming assignments into smaller assignments by means of its own recipes. The smaller assignments can then be sent to its own resources.

2.2 Controlling manufacturing systems

Controlling a manufacturing system means taking care of the proper execution of incoming instructions by the resources of the manufacturing system. When the controller receives the instruction to produce a certain product, it makes its resources execute the process steps from the product's recipe. The controller therefore communicates with its resources and its transporter by means of certain types of messages. There are three types of messages: requests, jobs and reports. Each type of message can have the prefix 'transformation' or 'transport'. A request from a transporter is called a transport request; a job for a resource is called a transformation job. Let us now see what the different messages mean.

Requests are used to express availability of resources or the transporter. Transformation requests contain a list with the operations that can be performed by the resource that sent the request. A transport request indicates that a transporter is ready to convey material from the store to a resource, from one resource to another, or from a resource back to the store. A controller can receive and send requests. It can receive a transport request from its transporter or a transformation request from one of its resources. But sometimes the controller itself must express which operations can be performed by the manufacturing system it controls. In that case the controller sends a transformation request, listing these operations, to its supercontroller.

Jobs are sent by controllers to their transporter and resources. Transport jobs are sent to transporters; transformation jobs are sent to resources. If a resource is expanded, jobs are sent to the controller of the expanded resource. So the controller of a manufacturing system can also receive jobs from its supercontroller. Since transport jobs are only sent to transporters, the controller of an expanded resource can receive only transformation jobs. Jobs, received by a controller, are also called *superjobs*; the jobs that the controller sends to its resources are called *subjobs.*

When the execution of a job is completed, a report is sent to the controller that sent the job. Transport reports are sent by transporters when a transport job has been completed; transformation reports are sent by resources when a transformation job has finished. The controller of a manufacturing system can receive reports from its transporter and resources, but it can also send a transformation report to its supercontroller. This happens when all the necessary subjobs for a received superjob have been executed by the resources.

2.3 The communication protocol

In the previous section we have named all the messages that are sent between the different parts of a manufacturing system. We shall now concentrate on the sequence in which messages are sent between resources and the controller, and between the transporter and the controller. The transporter also communicates with resources and with the store, but that concerns the exchange of material and is not discussed here.

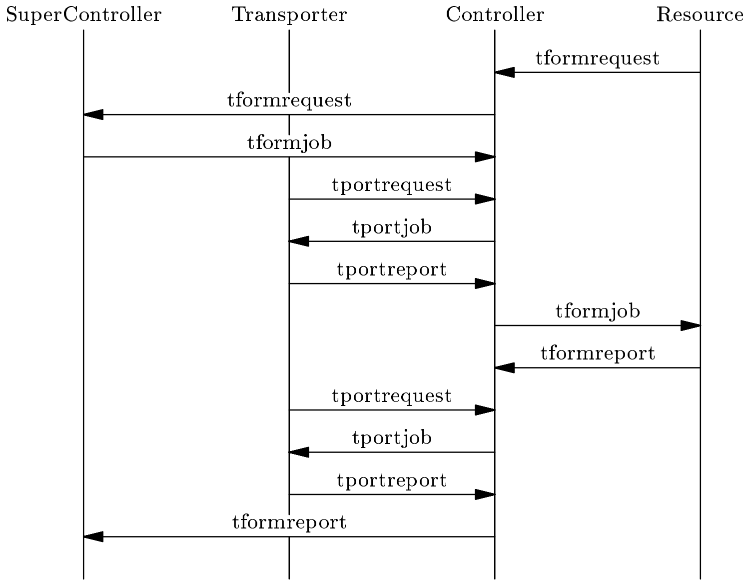


Figure 2: Message interface of controller, transporter, resource, and supercontroller

Figure 2 shows an interworking diagram, describing a possible sequence of communications. The diagram shows the messages, sent between a controller, its transporter, one of its resources, and its supercontroller. Each communication is denoted by an arrow, stating which message is sent to which component of the manufacturing system. The transporter and the resource send and receive their messages in a fixed order; the order in which the controller receives messages, depends on when the transporter and the resource send them. Therefore, there are several sequences possible for the controller; figure 2 shows one possible sequence of communications.
The idea is that when the controller receives a transformation job from its supercontroller, the material necessary to execute the job is already in the store. First, the controller sends a transport job to the transporter. The transport job instructs the transporter to bring the material from the store to the resource that must execute the first subjob from the recipe. If the resource is expanded, the material will be sent to the store of the expanded resource. Next, a transformation job is sent to that particular resource. The transformation job orders the resource to perform the proper operation on the material. When a resource has finished its operation, it sends a report to the controller. The material can then be sent to other resources to undergo the remaining process steps. If there are no more process steps, the material will be sent back to the store. Thus material always enters and leaves a manufacturing system via the store.

The resources and the transporter each start communicating by sending a request to the controller. In the request, they specify the work they can handle at that moment. Next, they wait for a job from the controller. Finally, when a job has arrived and has been executed, a report is sent back. The report informs the controller that the execution of the job has been completed.
The controller does not start sending messages, but reacts to the reception of messages from its resources or from the transporter. When the controller has received a transformation request from every resource that is needed to perform a certain operation, it sends a transformation request for that operation to its supercontroller. When a transformation job has been received from the supercontroller, the controller sends subjobs to its resources in order to execute the superjob. The controller also sends

transport jobs to the transporter, when material has to be moved. When the controller receives a report, telling that a subjob has been completed, it sends the next subjob from the recipe. If there are no more subjobs, the material is transported back to the store. So when the controller receives a transport report, telling that a piece of material has been transported to the store, it sends a report to its supercontroller. This informs the supercontroller about the completion of the job that was sent to the resource by the supercontroller.

Not only control messages are sent within the manufacturing system. Material is being transferred as well. We will now discuss two principles concerning the transportation of material: material driven manufacturing versus command driven manufacturing, and an early transport strategy versus a late transport strategy.

We have stated before that when a controller receives a job, the material needed for that job is already present in the store. So the resource can start operating as soon as it has received the job. We say that the reception of the job *triggers* the execution of the operation. This characterises *command driven manufacturing*. Another way of operating would be to send the material to a resource after the job has been sent. Then the resource starts operating as soon as it has received the material. This is how *material driven manufacturing* works. Material driven manufacturing has the disadvantage that the route flexibility of the system decreases, because the route of material is fixed at a very early moment in the manufacturing process. With command driven manufacturing, the decision to which resource to send material can be made by the controller as late as possible. Both approaches can be found in the report series on the specification of a model factory ([vVvW92], [HP92], [PV93], and [BP93]). In [HP92] and [PV93] material driven manufacturing has been used, whereas command driven manufacturing has been used in [BP93]. In this paper we will use command driven manufacturing, as is the case in [Smi92].

In this paper we will use a *late transport strategy*. This implies that material is transported as soon as the destination resource becomes available (i.e., when it sends a transformation request). Now the material always arrives too late, keeping the resource waiting during transportation. An *early transport strategy* means that the material is transported as soon as it arrives at a manufacturing system, or at the moment a resource has finished operating on it. The advantage of this strategy is that wherever the material is being transported to, it will be there as soon as possible. This keeps the time that a resource might have to wait minimal. It might even be the case that the material has to wait in the store of an expanded resource, because the resource is still busy. Now it becomes clear immediately that this strategy does not work in a manufacturing system with leaf resources: if material is sent to a leaf resource that happens to be still busy, a deadlock occurs. So in such a manufacturing system an alternative strategy must be used. Late transport, however, avoids deadlocks in manufacturing systems with leaf resources. Therefore, late transport must always be used at that level. For manufacturing systems that do not have leaf resources, one can choose between an early and a late transport strategy. In this paper we have chosen to use late transport for all manufacturing systems, just as has been done in [Smi92].

3 The specification language PSF

A slightly different version of the text in this section appeared earlierer in [FvW92].

The specification language PSF, a Process Specification Formalism, has been developed at the University of Amsterdam. It is a combination of an algebraic specification formalism for data, and process algebra. Therefore, PSF has an explicit data-part and a process-part using the data. Some general references to PSF are [MV89], [MV90], [MV93] and [Vel93].

3.1 Data in PSF

This section contains a brief introduction to the specification of data by algebraic specifications, as is prescribed in PSF. More detailed background information can be found in [BHK89b] and [BHK89c].

Data are time-independent objects with mathematical properties such as symmetries, invariants and representation results. In PSF, data are constructed from a given, finite number of *constants* and *functions*. Furthermore, each data term is typed by a sort name (such as Naturals or Booleans). Properties of data can be defined by *equations* of the form $d = d'$, where d and d' are data terms of the same sort that may contain (typed) variables.

For example, let Nat be the sort of the naturals, with a constant 0 of type Nat and a function $S : \mathrm{Nat} \to \mathrm{Nat}$, denoting the successor function. No equations are specified. Then the *initial algebra* (representation of all closed terms modulo provable equality) of Nat is the collection

$$\{\ 0,\ S(0),\ S(S(0)),\ \ldots\ \}.$$

One can add a function such as $+ : \mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Nat}$, denoting addition, with equations

$$n + 0 = n$$
$$n + S(m) = S(n + m)$$

where m, n are variables of the type Nat. In this case the initial algebra is

$$\begin{array}{ll} \{ & [\ 0,\ 0+0,\ (0+0)+0,\ \ldots\ ] \\ & [\ S(0),\ 0+S(0),\ S(0)+0,\ \ldots\ ] \\ & [\ S(S(0)),\ S(0)+S(0),\ 0+S(S(0)),\ \ldots\ ] \\ & \cdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \end{array}$$

where the collections between square brackets are the equivalence classes of the sort Nat, generated by provable equality (a congruence relation).

In general, it can be hard to decide whether or not two closed data terms are equal. A means for doing this is offered by the theory of *term rewriting systems*. One can try to set up the equations for the data in such a way that, when applied from left to right only, each reduction of a closed data term is finite. This means that every reduction ends with a term, called a *normal form*, to which none of the (left-hand sides of the) equations matches anymore. The aim is to have *unique* normal forms, i.e., if two closed data terms are equal, their normal forms are literally the same. Then it clearly is decidable for each pair of closed data terms whether or not they are equal, by simply reducing them to normal forms.
In the example of the natural numbers above, the normal forms are the expressions of the form $S(S(\ldots S(0))\ldots)$. Thus the closed data terms of the sort Nat have unique normal forms. However, if an equality $m + n = n + m$ is added to the equations for the $+$, one gets infinite reductions such as $0 + S(0) = S(0) + 0 = 0 + S(0) = \ldots$.
Modules enable the explicit declaration of algebraic data types as described above. A module consists of two parts: one in which the *algebraic signature* is given and a second which contains the *equational logic*. So a data module is a structure of the following form:

```
data module Naturals
begin

  < algebraic signature >

  < equational logic >

end
```

Signatures are sets of declarations of *sorts* and *functions*. A *constant* is simply a function with no arguments. For example, the algebraic signature of the module Naturals is as follows:

```
sorts
  NAT

functions
  0   :             -> NAT
  S   : NAT         -> NAT
  _+_ : NAT # NAT -> NAT
```

In the second part, the equations for the data terms are given. In the case of the Naturals this is:

```
variables
  m,n :      -> NAT

equations
  [N1]  n+0 = n
  [N2]  n+S(m) = S(n+m)
```

An important feature is *import*. A module can import other modules, whence an initial algebra that has been created in a first module can be used in a second one.

3.2 Processes in PSF

In this section a short description of process algebra is given. Process algebra forms, together with module algebra and term rewriting techniques, the algebraic basis of PSF. After a short description of process algebra, some characteristics of PSF are mentioned, and some of the tools that are available for the manipulation of specifications in PSF.
A process can be described informally as *the behaviour of a system* over time. It turns out, however, to be an intricate philosophical problem to describe the difference between processes and data in a formal way. This is especially true when processes are parameterised with data, and complex interactions between processes and data take place. In [Pon92] attention is paid to process algebras with data.
Process algebra is a formal approach to the specification and verification of concurrent systems. It may be considered the study of concurrent communicating processes in an algebraic framework. Several process algebras have been developed in the past decade, but here we discuss only ACP, the Algebra of Communicating Processes. There are different variants of ACP. For instance, abstract or untimed ACP [BW90] and Real Time Process Algebra [BB91]. We restrict ourselves to abstract ACP, which forms part of the core of PSF. Other approaches to process algebra can be found in [Hoa85], [Hen88] and [Mil89].

We briefly introduce ACP. The theory of ACP has two parameters: a set of *actions* A, and a partial communication function $\gamma : A \times A \to A$ that is commutative and associative. Actions are the most basic *processes* considered, and are not assumed to be subject to further division. Three important binary operators to compose processes into more complex ones are sequential composition ($\cdot$), alternative composition or choice ($+$), and parallel composition ($\|$). Particularly, $p \parallel q$ represents the process that either performs a first action of p, or of q, or a 'communication action' of p and q, i.e., the simultaneous occurrence of a first action of p and one of q. There are also special constants in process algebra, the most important of which are the *deadlock* (δ) and the *silent step* (τ). If a system can perform no action but a deadlock, it is blocked: it cannot continue or terminate successfully. The silent step denotes an *internal* action, i.e., an action that is hidden (not observable). Two system operators must also be mentioned here: *encapsulation* and *hiding*. Encapsulation forces the components of a parallel system to communicate and thus to synchronise. Hiding renames all internal actions of a system into a silent step. Once the signature of a process algebra is defined, an axiom system can be defined. An axiomatic *verification* of a system is a formal proof of the fact that the observable behaviour of the system (abstracting from internal actions) satisfies a given requirement. Algebraic verification can be performed simply by manipulating equations, using the axioms of a given algebra. Another option is verification in some specific *model* for processes (a process algebra). Well-known models for processes are graphs or labelled transition systems (after dividing out some behavioural congruence such as *bisimulation*). Practical examples of specifications and verifications of concurrent systems can be found in the literature. Examples of such systems are protocols in computer networks, systolic algorithms, and logistic protocols in factories. Some references are [Bae90] and [vW92].

The formal specification language PSF is based on abstract ACP. It was developed in order to provide ACP with the means to write tools for the verification and simulation of specifications. The formal definition of PSF also makes it possible to verify whether a specification is syntactically correct or not. The definition of PSF can be found in [MV90].
A PSF specification is organised in *data modules* and *process modules*. The different modules of a specification are connected via an import-export mechanism. The *exports* section of a module contains the visible part of the algebraic signature of a module when it is imported into another module via its *imports* section.
In process modules, the processes of a specification are specified. For example: if a system SYS consisting of a sender S and a receiver R is specified, S can be put in one process module and R in another. Then the process SYS can be specified in a *top-module*, which means that it forms the highest level in the specification in which all other modules are imported.
In data modules, the abstract data types used in a process specification are defined. No standard types (*sorts* in PSF) such as naturals or reals are available. This part of PSF is not based on process algebra but on module algebra and term rewriting techniques, as described in the previous section. In process modules, data may be imported and used in various manners to specify processes: communication can be defined in data parametric way; processes can be defined recursively in a data parametric way with tests on data values; and there is a 'sum' operator for defining the alternative composition of the closed data-instantiations of a parametric process.

The specification language PSF is equipped with several tools for the analysis of a specification. We mention some of these that are typical for analysis by simulation:

1. The *term rewriter*. By means of this tool, expressions are rewritten to normal forms, which is useful to verify whether or not functions have been defined correctly.

2. The *initial*-tool, which makes it possible to calculate (part of) the initial algebra of a specification. It calculates the different elements of a given sort (type), using the functions and the rewrite rules defined on these functions.

3. The *simulator*. This tool makes possible a stepwise analysis of a specification. The simulator performs successive actions that the user can choose from a menu. To put it more technically: with the simulator traces through the *state-space* of a system can be generated. The simulator can also make random runs. This option yields, in general, a fast method to detect deadlocks or other undesired system behaviour.

Tools 1 and 2 are suitable for the analysis of the data in specifications.
Some recent examples of PSF specifications can be found in [Bru93] and [HP92]. For more detailed information on the syntax of PSF the reader is referred to [MV90] and [MV93].

3.3 Specification and Verification of CIM-Architectures in PSF

Computer Integrated Manufacturing, shortly CIM, refers to industrial production in a setting in which computers are used. Both the product flow and the information flow can first be designed and analysed

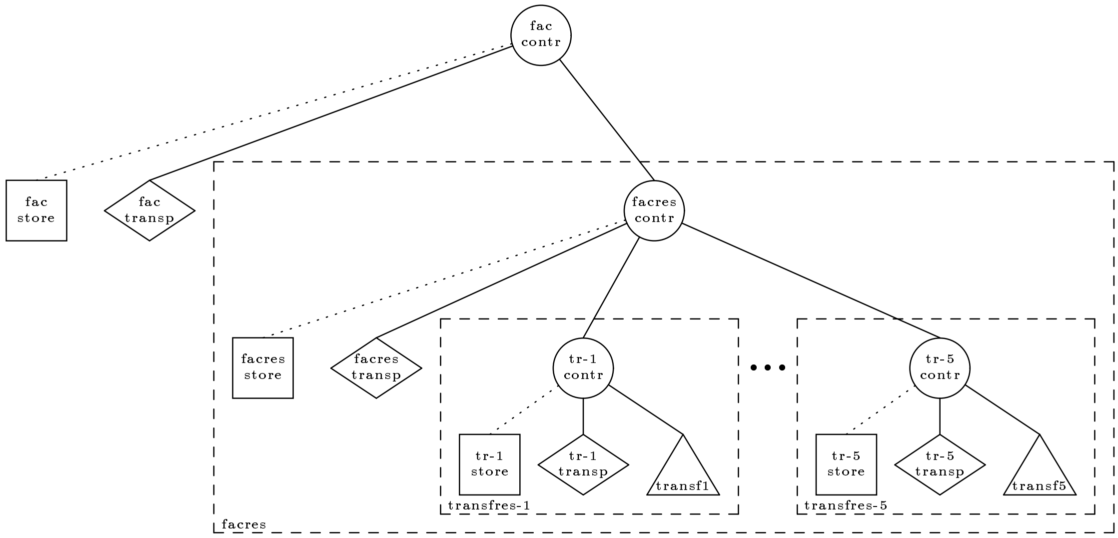


Figure 3: A fictitious factory with information flow.

by means of computers. Next, also the actual control of industrial production can be organised by means of computers. In such production systems, often many machines operate independently of each other. Providing a correct functioning of the total of all machines, computers and transport services is not a trivial exercise. Before such a CIM architecture can be built, one must design it. Such a design, when judged 'correct', describes a properly functioning system on a symbolic level. In particular, the possibilities to use process algebra as a means for the specification and verification of CIM architectures are manifold.
The specification of leaf resources and transporters can be given in PSF. Examples can be found in [Mau90], [vVvW92], [HP92], [PV93], and [BP93].
When abstracting from the internal actions, it is possible to determine its external, or *observable* behaviour. At the high level view on the flow of products, one can be only interested in how products pass resources. Also, at the high level view on the flow of information, one may only be interested in the sequence of messages (requests, jobs, reports) transferred between the various parts of a manufacturing system.

Finally, it must be remarked that no real time features are captured. Consequently, important features like *efficiency* (maximal productivity of machines) and *tuning* (synchronisation of the speed of the machines) cannot be modelled. Currently, the essential realtime features have been added to process algebra ([BB91]). Real space features have been added to process algebra as well ([BB93]).

4 Modelling a manufacturing system

In this section we will explain how we have modelled a manufacturing system in PSF. It concerns a fictitious manufacturing system with the same structure as the one used for a case study in [Smi92]. In our model, the names of products, machines and material are strictly symbolical.
Our specification describes a market, consisting of a consumer and a factory. Figure 3 contains a schematic picture of the factory. The picture clearly shows that the factory is a hierarchical manufacturing system. It has a controller (`fac contr`), a store (`fac store`), a transporter (`fac transp`), and

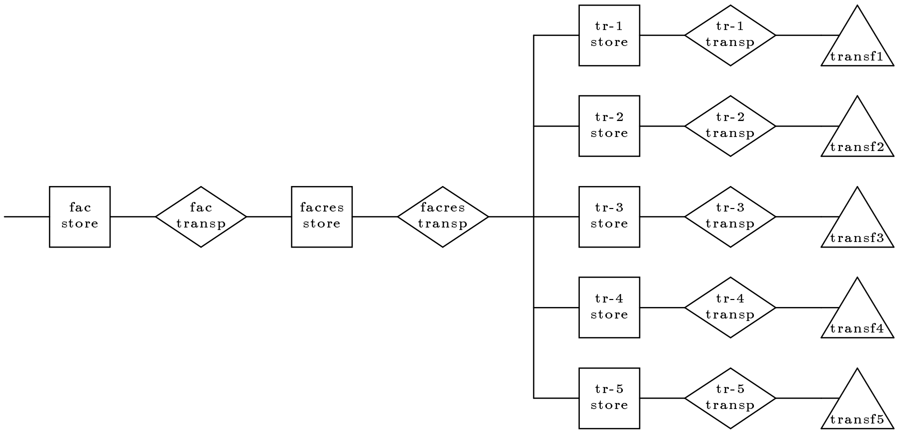


Figure 4: Product flow of the factory.

one expanded resource, the factory resource (`facres`). The factory resource is a manufacturing system itself. The factory resource has five expanded transformation resources `transfres-1` to `transfres-5`. The controller of each transformation resource (`tr-X contr`) controls one leaf resource, a transformer. In figure 3, the controller of each manufacturing system is connected to its store, transporter and resources by a line. The line from a controller to its store is dotted because there is no communication between them. A store only communicates with transporters. The specification of the market can be found in appendix A.
As we have mentioned before, it is the controllers task to direct material properly from its store along its resources and then back to the store. If a resource is expanded, the material will be put in the store of the expanded resource, where it will return after it has visited resources of the expanded resource. So what we try to make clear is that the actual location of material is always a store, a transporter or a leaf resource. We have tried to visualise this in figure 4, where the product flow of our fictitious factory is depicted.
Finally, it must be remarked that we assume that only one piece of material is needed for the manufacturing of a product. This piece of material is directed along various resources, which operate on it one after another.

4.1 Specifying the data

For the specification of the data we have used the PSF-library. The PSF-library ([vW93]) contains a number of standard specifications of frequently used data types such as the booleans and the natural numbers. Besides these two data types, we have used the specification of queues and tables from the PSF-library.
We will now mention some general features of the data specification.

- Equality between constants of a certain sort is determined by converting each constant to a unique natural number. If the corresponding naturals are equal, the constants are equal as well. We do this because it allows us to use the equality defined on the natural numbers in the PSF-library. By not doing so, we would have to state explicitly for every pair of constants of a certain sort, whether those constants are equal or not (see [BMW91]).

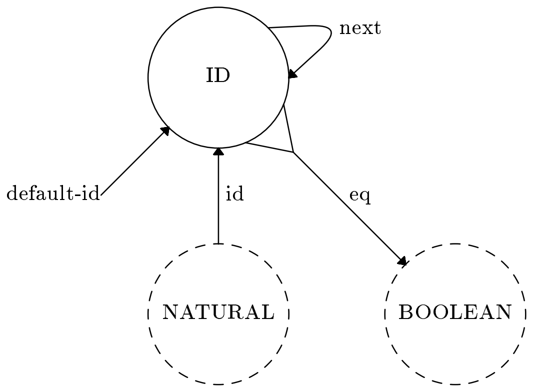
- Sometimes functions make only sense when they are applied to certain elements of a data type. For example, in the data module `Jobs`, the function `operation` can be applied to all jobs, but makes only sense when applied to a transformation job. Still we have to define that function

applied to a transport job. We have solved this problem by defining so-called *nil*-constants of certain sorts. These constants can be assigned to 'senseless' function applications. So now we can assign `operation-nil` to `operation(<transport-job>)`.
Note that this is only done for readability purposes. We always identify a nil-constant with some 'real' element of the concerned type, since we do not want the nil-constant to be added to the sort. The equations in which nil-constants are assigned to 'senseless' function applications can be found at the end of the data modules in which they appear, after a blank line.

We have added diagrams, visualising the signature concerning the sort that is currently discussed. Circles denote the collection of data terms of that sort; arrows model the functions. When a circle is dashed, the corresponding sort has been defined before, or is a basic sort from the PSF-library.

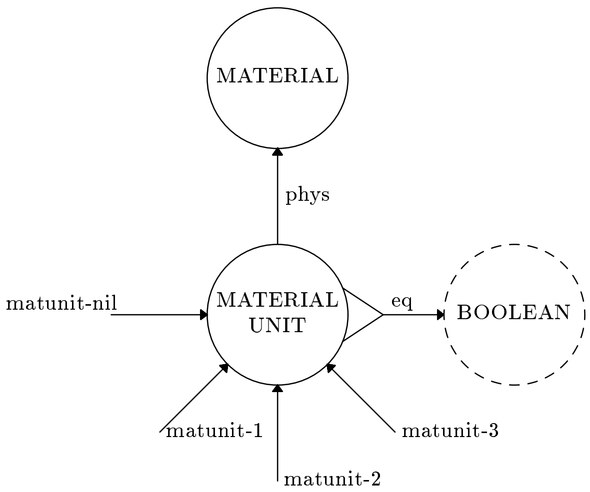
The basics. First we discuss the data types that form the building blocks of our specification. Other data types, such as the control messages, depend on these data types.

Identifiers. The identifiers are specified in the data module `Ids`.



The ids serve as the serial numbers on orders and jobs. Ids are in fact just natural numbers, converted to the sort `ID` by the function `id`. So ids look like `id(NAT)`. In order to keep the number of different ids finite, we state that `id(n) = id(nat(^0))` for every n greater than ten. The special constant `default-id` is needed in the process module `ManufacturingControllers`.

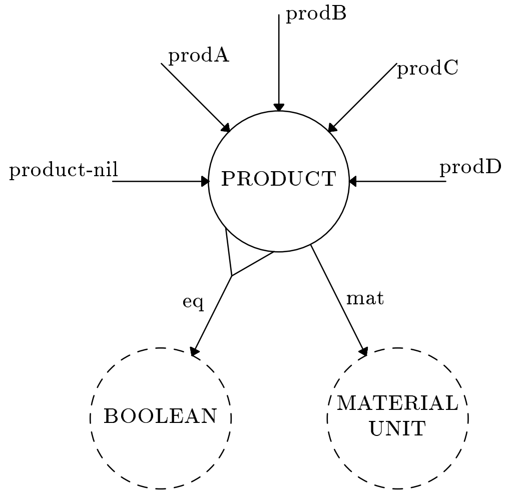
Material. In the data module `Material` the sorts `MATERIALUNIT` and `MATERIAL` are defined.



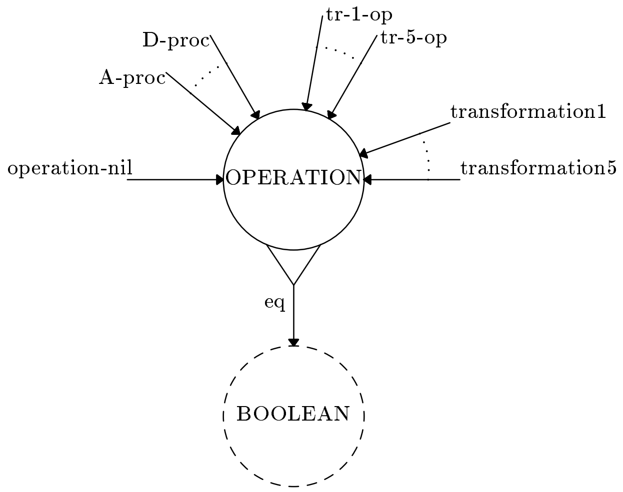
We will use materialunits most throughout the specification. A materialunit is the description of a piece of material by name. It is used in jobs or other administration structures and processes.

We have defined three materialunits. Physical pieces of material are terms of the sort `MATERIAL`. A materialunit can be converted to material by the function `phys`. So `phys(materialunit-1)` denotes the physical representation of `materialunit-1`.

Products. The products are the items that are finally produced by the factory and that are ordered by the consumer. Our factory can produce four different products, specified as constants in the data module `Products`. The function `mat` returns the materialunit that is needed to produce a given product.



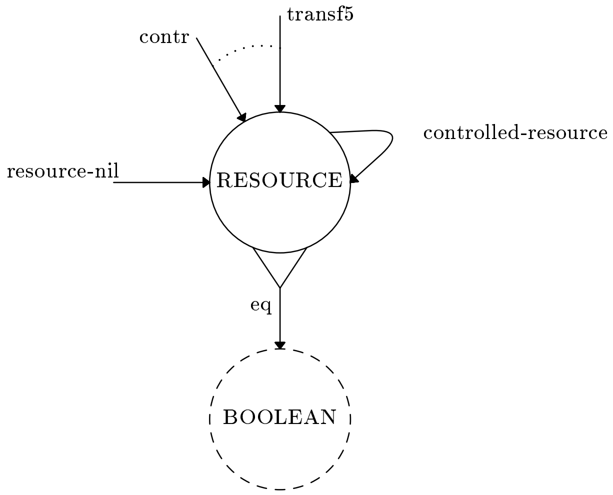
Operations. Operations, specified in the data module `Operations`, are the basic actions of a manufacturing system on a given level of abstraction (on a given level in the control hierarchy).



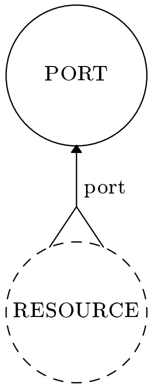
Recipes contain sequences of operations that have to be performed by the resources of a manufacturing system. Each operation in a recipe will therefore appear in a transformation job for one of the resources. If that resource is expanded, then the operation is not basic on the level of the expanded resource, but will have an entry there in the 'cookbook'. We will come back on this subject when we discuss recipes.

Resources. In the data module `Resources` we specify all the parts of our factory on every level of abstraction. This means that every store, transporter, controller and resource is named in this module. If some of these parts together form an expanded resource, this expanded resource as a whole is named as well. We will call the constants in the data module `Resources`, referring directly to a certain part of the factory, the *absolute resources*. We have also specified a

couple of *relative resources*: the constants `contr`, `trans`, `store`, `intern`, `extern` and `outside`. These relative resources make it possible to talk about, for example, the `contr`ol-port of a transporter. When we discuss ports we will see how the different parts of the factory are linked using combinations of absolute and relative resources.
Finally, for every controller, the function `controlled-resource` gives the expanded resource of which the given controller is the controller. This function behaves like the identity-function if a resource is given that is not a controller.



Ports. Ports are the connections between the different parts of the factory. In [HP92] the ports have been specified as a very long list of constants of sort `PORT`. Here, this is not the case. If we apply the function `port` in the module `Ports` to a pair of resources, we obtain a term of the sort `PORT`.

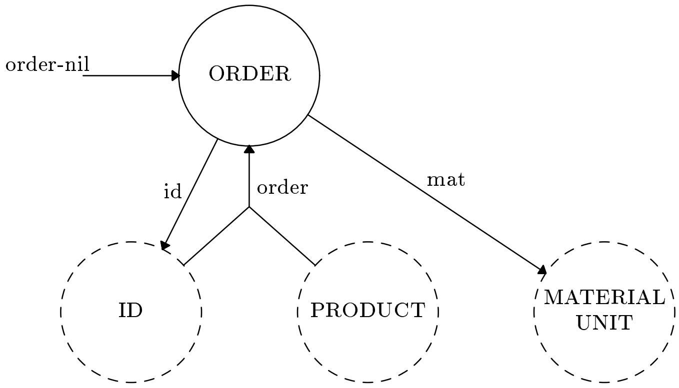


The first argument of the function `port` should always be an absolute resource. The last argument is often a relative resource. Now we can speak of ports in the following way:
`port(fac-contr, trans)` denotes the port of the factory controller, leading to the transporter. In other words: the transporter-port of the factory controller. The transporter of the factory controller is the factory transporter, as figure 3 shows. So the control-port of the factory transporter should lead to the factory controller. This connection can be made by adding the equation `port(fac-contr, trans) = port(fac-transp, contr)`, stating that the control port of the factory transporter and the transporter port of the factory controller are the same.
The second argument of the function `port` does not have to be a relative resource. If a manufacturing system has more than one resource, the term `port(facres-contr, res)`, with relative resource `res`, would not make any sense, since it does not say which resource is being meant. Because every manufacturing controller knows, as we will see later, the names of its resources,

terms like `port(facres-contr, transfres-2)` will be formed. Because `transfres-2` is an expanded resource, messages should be sent to the controller of `transfres-2`, named `tr-2-contr`. Therefore, the connection between the factory resource controller and the second transformation resource is made by the following equation:
`port(facres-contr, transfres-2) = port(tr-2-contr, contr)`.

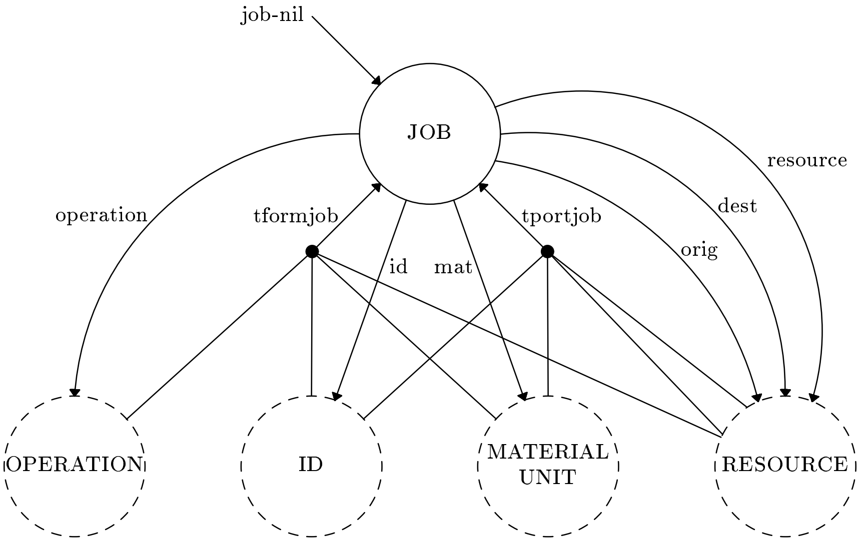
The messages. Now that we are familiar with the basic data types of the specification, we will talk about the modelling of the messages that are being transferred between the controller and the other parts of a manufacturing system.

Orders. Orders are not really sent *between* parts of a manufacturing system. The consumer sends them to the factory controller to ask for a certain product. Therefore, an order triggers the complete manufacturing process for that product.



Orders, specified in the data module `Orders`, are combinations of an `ID` and a `PRODUCT`. The id is assigned by the consumer and will be used for all necessary subjobs throughout the manufacturing process. The description of the material needed to manufacture the product asked for, a materialunit, is given by the function `mat`.

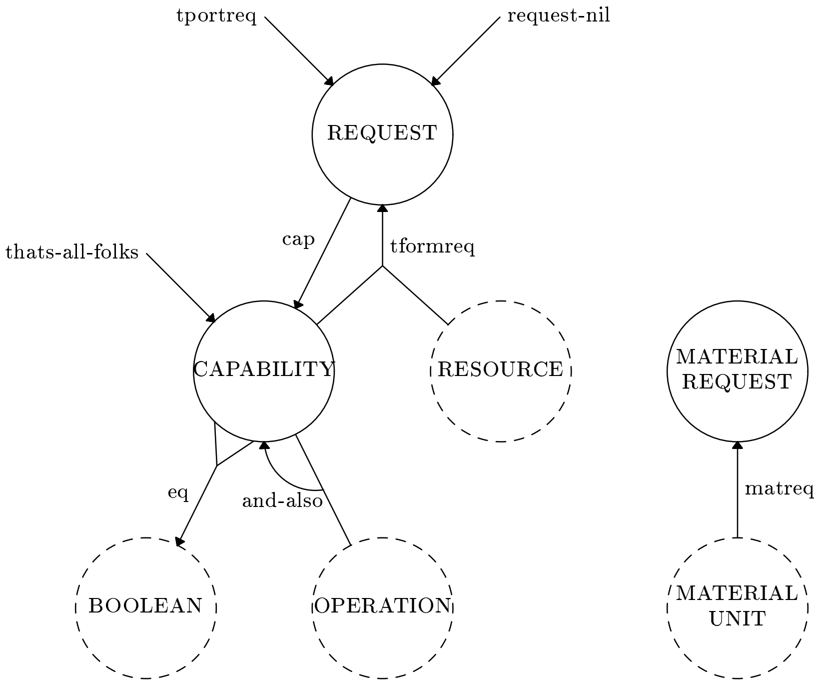
Jobs. When we specify jobs, we have to distinct transformation jobs and transport jobs.



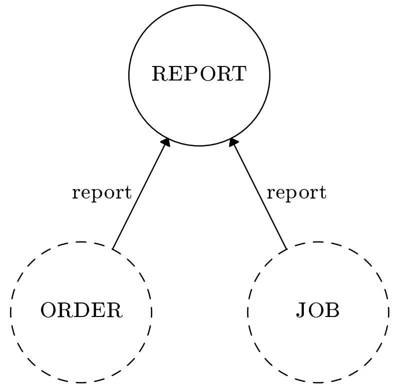
In the data module `Jobs` this is done by using two different constructor functions for jobs. First

of all, the function `tformjob(ID, OPERATION, RESOURCE, MATERIALUNIT)` constructs transformation jobs. A transformation job can now tell which operation has to be performed by which resource on which kind of material. The id makes clear for which order or super job the transformation job must be executed.
The function `tportjob(ID, RESOURCE, RESOURCE, MATERIALUNIT)` is used to construct transports jobs. Transport jobs tell which kind of material has to be transported from where to where. The first resource argument denotes the origin, the second one the destination. The id has the same meaning as in a transformation job.

Requests. Specifying requests, again we have to distinguish transformation requests and transport request. Just as we did with jobs, we handle this using different constructor functions. Transformation requests are used to inform controllers about which operations its resources can perform at a certain moment. In the data module `Requests`, the constructor function for transformation requests looks like `tformreq(RESOURCE, CAPABILITY)`. The resource argument denotes the resource that sends the requests. The capability argument is the list of operations that can be performed by that resource. Terms over the sort `CAPABILITY` are constructed as follows: the constant `thats-all-folks` models the empty list; the function `and-also` adds an operation to the capability (the operation-list).
Transport requests are constructed by the function `tportreq`. A resource argument is not necessary, because each controller has only one transporter. It will always be clear that a transport request comes from the transporter. Since a transporter can either convey material or not, there is also no need to specify all the possible combinations of origins and destinations that may appear in a transport job. Therefore, the function `tportreq` has no arguments.
Finally, when a transporter must fetch some material from a store or leaf resource, it first sends a *material request* to the store or leaf resource concerned. We will talk about this more extensively when we discuss stores, transporters and transformers. The function `matreq(MATERIALUNIT)` constructs materialrequests, the materialunit argument specifying the material asked for.

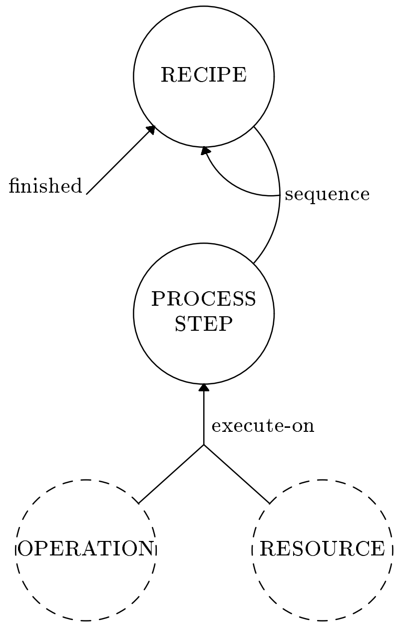


Reports. Reports are just messages, stating that a particular job or order has terminated successfully. So in a report we only need to specify which order or job that is. In the data module `Reports` we have modeled this using the constructor function `report`. This function has one argument that can either be an order or a job, specifying the order or job that is finished.



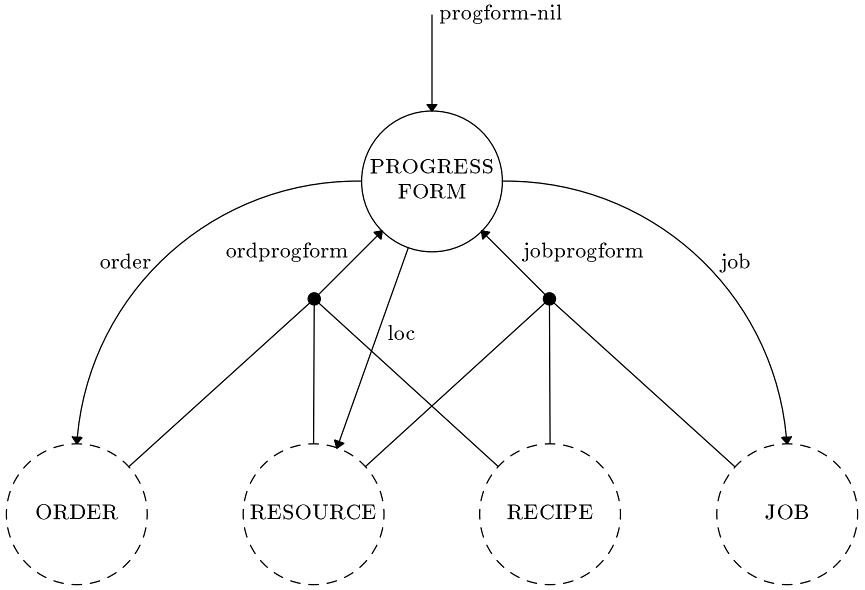
Further data specification. Next we will discuss the specification of the remaining data types. These data types are being used in the specification of the processes of a manufacturing system.

Recipes. We have already mentioned recipes a couple of times. A recipe contains the sequence of operations that have to be performed on a piece of material within a manufacturing system. It is used by the controller when a piece of material must be directed along the resources of the manufacturing system. In the data module `Recipes` we have modeled recipes as a list of process steps. The constant `finished` denotes the empty recipe; the function `sequence` adds a process step to a recipe. Process steps are combinations of an `OPERATION` and a `RESOURCE` and are constructed by the function `execute-on`. So `execute-on(OPERATION, RESOURCE)` is a process step, telling which operation has to be performed on which resource.



Progressforms. When a manufacturing controller receives a transformation job, it must record the progress of that job. This is done with a job-progressform, specified in the data module

ProgressForms. Job-progressforms contain the job of which the progress is being recorded, the present location of the material that comes with the job, and a copy of the recipe prescribing the execution of the job. The progress of the execution of a job can now be monitored as follows: whenever a process step from the recipe has been executed, it is removed from the recipe, until only `finished` remains. At that point the material is transported back to the store. `jobprogform` is a constructor function for the sort `PROGRESSFORM`, having a job, a resource, and a recipe as its arguments.
The factory controller is the only manufacturing controller that does not receive jobs from a supercontroller. It receives orders from a consumer. Therefore, we must specify separate progressforms for the factory controller, order-progressforms. Order-progressforms are constructed by the function `ordprogform`. They are used in exactly the same way by the factory controller, as are job-progressforms by manufacturing controllers on a lower level in the hierarchy.



Queues and tables. By now we have mentioned a lot of data types to be used in the specification of our factory. Yet, sometimes, we also want to administrate instances of these data types. For example, a manufacturing controller needs to have some kind of data structure to collect the recipes that its resources can execute. It also needs to archive the progressform of every job that has been received and that is being executed. Finally, the requests from transporter and resources must be stored, so that the controller knows the status of its resources. The PSF-library ([vW93]) provides two data structures for the ordering of instances of data types: a queue and a table. Both these data structures have the sort of its elements as a parameter. By importing the module specifying the queue or table, and binding the sort-parameter to a particular data type, one can create a queue or table for storing elements of that data type. We have done this for a number of data types in the data modules `Job-queues`, `Product-tables`, `Recipe-tables`, `Progform-tables` and `Request-tables`. An exception has been made in the data module `Material-tables`, where we have specified our own table. This table differs slightly from the one in the library. We will talk about each queue or table more extensively when we discuss the process modules where it is needed.

The cookbook. In the data module `Cookbook`, all the recipes that are used in the factory are listed. For each level of control, the recipes are ordered in a recipe table. When we specify the manufacturing controllers, recipe-parameters will be bound to these recipe tables.

Control functions. Controlling a manufacturing system implies making lots of calculations. Information, like the status of the different resources, the degree of progress of the received jobs, and the recipes, is used for these calculations. The calculations are being made with use of the functions, listed in the data module `Control-functions`.

4.2 Specifying the communication

Communication between the different parts of a manufacturing system can take place by means of `read` and `send` actions. These actions have two arguments: the first one specifies where to read from or where to send to; the second one specifies what is being read or sent. The first argument is always a term of the sort `PORT`. The second argument can be a term of any sort, depending on what is being transferred. Therefore we must specify a `read` and `send` action for each sort we want to transfer instances of. We have done this in the process module `Communication`. Using the communication and encapsulation mechanism of PSF we can force a simultaneous occurrence of a `read` and a `send` action in two different process expressions, to fuse into a new action. For this purpose we have defined an action `comm` for each sort we want to transfer instances of. These `comm` actions also have a port as its first argument and the sort of what is being transferred as its second argument. Now we can define communications as for example

```
read(p,j) | send(p,j) = comm(p,j)
```

for a port p and a job j. This definition demands that the same job is sent and read simultaneously over the same port. Otherwise there can be no communication. All communications that are needed are defined in the process module `Communication`.

Let us now look at an example. Suppose that the factory controller wants to send a job to the factory transporter. That job must be sent over the port connecting the factory controller and the factory transporter (see figure 3). In that case the following action will appear in the process expression of the factory controller:

```
send( port(self,trans) , transjob(_,_,_,_) )
```

The process expression of the factory transporter will at some point have the action

```
sum(j in JOB,  read( port(self,contr) , j )  )
```

The sum is a formal abbreviation: `sum(n in NATURAL, P(n))` denotes the process `P(1) + P(2) +` $\cdots$ (using an informal notation for the natural numbers, see the PSF-library). So what the above expression yields is that the factory transporter can read an arbitrary job on port `port(self,contr)`. The term `self` is a parameter, which the process expression of every part of the factory is equipped with. It stands for the name of the factory part that is being modeled. It can be bound to any name from the data module `Resources`. We will take a closer look on how parameters are defined and bound in process expressions later; the only fact we need to know now is that via the parameter `self`, every part of the factory knows its own name. We can now understand that the two `port` expressions above will be rewritten into `port(fac-contr,trans)` and `port(fac-transp,contr)`. Next, the first new `port` expression will be rewritten into `port(fac-transp,contr)`, using equation `[Po1]` from the data module `Ports`. All this term rewriting results in the following two terms:

```
send( port(fac-transp,contr) , transjob(_,_,_,_) )
```

and

```
sum(j in JOB,  read( port(fac-transp,contr) , j )  )
```

Because of the `sum` construction, any job that is read from `port(fac-transp,contr)` is covered in the second expression, in particular the one that is sent by the factory controller on the same port. So when the above actions occur simultaneously in two parallel processes, these processes can then communicate and produce the resulting action

```
comm( port(fac-transp,contr , transjob(_,_,_,_) )
```

In fact, using *encapsulation*, one can force this communication to happen by blocking the separate `read` and `send` actions. We will say more about the notation of encapsulation later.

4.3 The store process

Stores serve as the in- and output buffers for manufacturing systems. When material arrives at a manufacturing system, it is sent to the store of the manufacturing system. Later, as the corresponding transformation job arrives at the controller of the manufacturing system, the controller orders the transporter to convey the material out of the store to a certain resource. When all necessary operations have been performed on the material, it is sent back to the store. From there, it can be removed by the transporter of a higher control level.

When we want to model a store, we have to keep track of what is in it. In the process module `Stores` we have achieved this by giving the process `STORE` a material table as an argument. This material table is not an instance of the table from the PSF-library, because the standard table can not contain identical entries. That is not what we want, since stores must be able to store multiple pieces of the same material. Therefore we have specified a new table in the data module `Material-tables`. It has nearly the same interface as the one in the library, but it allows multiple occurrences of the same element. The material table is initially empty (`STORE(self) = STORE(self, empty-mat-table)`). The resource argument `self` represents the name of the store that is being specified. Remember that in the process module `Stores` we give a general specification of stores. This specification can be used later to model a specific store by calling, for example, `STORE(facres-store)`.

Considering what we have said above about the function of a store, we can deduce that a store must have two ports: an internal one and an external one. The port `intern` leads to the transporter of the same manufacturing system; The port `extern` leads to the transporter of a higher control level (see also figure 4). In figure 5, the process graph for the store process can be found. The store process can develop in the following ways:

- The store receives a request for material at its internal port. We assume that there are no requests that cannot be honoured. So next, the store delivers the material at its internal port. Finally, it returns to its initial state, deleting the description of the delivered piece of material from the material table.
  ```
  sum(mu in MATERIALUNIT, read(port(self, intern), matreq(mu)) .
  send(port(self, intern), phys(mu)) . STORE(self, delete(mat-tab, mu)))
  ```
- The store receives a piece of material at its internal port, which must be stored. The store accepts the material and returns to its initial state, adding the description of the material to the material table.
  ```
  sum(mu in MATERIALUNIT, read(port(self, intern), phys(mu)) .
  STORE(self, insert(mat-tab, mu)))
  ```
- Exactly the same as described in the first two points can happen at the external port of the store.

The factory store is the only store in the factory that behaves a little different than the other ones. The external port of the factory store does not lead to a transporter within the factory, but to producers and consumers. The producers supply the material that is needed to manufacture the desired products;

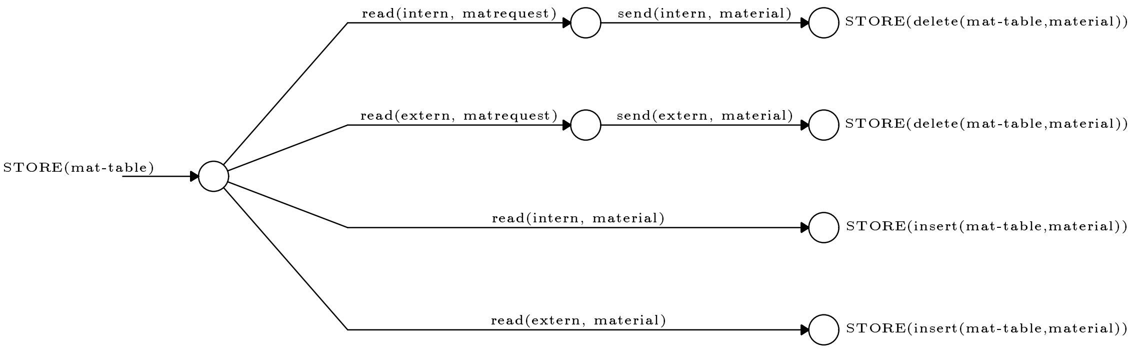


Figure 5: Process graph for the store process

consumers take these products when they are finished.
The point is that we do not consider producers in this report. Therefore we must assume that all necessary material is present in the factory store. We have modeled this in the process module `FactoryStore` as follows:

```
sum(mu in MATERIALUNIT, read(port(fac-store, intern), matreq(mu)) .
send(port(fac-store, intern), phys(mu)) . FACTSTORE(mat-tab))
```

The material desired is always given off at the internal port, but this has no effect on the material table. At the external port everything behaves normally.

4.4 The transporter process

As figure 4 shows, transporters convey material between two stores or between a store and a leaf resource. If a manufacturing system has multiple leaf resources, the transporter can move between two leaf resources as well. We have specified transporters in the process module `Transporters`. Figure 6 gives the process graph for the transporter process. The process describing transporters consists of the following recurring sequence of actions:
First of all, when a transporter is ready to move, it reports its availability by sending a transport request to the controller:

```
send(port(self, contr), tportreq).
```

Then there will come a transport job from the controller eventually:

```
sum(j in JOB, read(port(self, contr), j))
```

the sum construction makes sure that the transporter can read any job, especially the one that will be sent. After the job has been received, the transporter moves to the place, mentioned in the origin-field of the transport job:

```
move-to(self, orig(j))
```

We have added `self` as an argument of this action to be able to see which transporter is moving where to. Once the transporter has arrived at the origin of the transport job, it asks for the material mentioned, receives the material, and moves to the place, mentioned in the destination-field of the transport job:

```
send(port(self, orig(j)), matreq(mat(j))) .
read(port(self, orig(j)), phys(mat(j))) . move-to(self, dest(j))
```

Having arrived at its destination, the transporter delivers the material, which concludes the execution

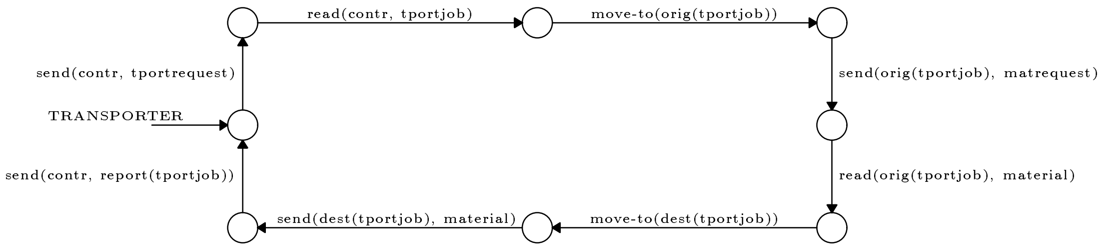


Figure 6: Process graph for the transporter proces.

of the transport job:
`send(port(self, dest(j)), phys(mat(j)))`
The transporter then reports that the job is finished by sending a report to the controller,
`send(port(self, contr), report(j))`
and returns to its initial state.

4.5 Leaf resources

Leaf resources are the machines that physically operate on material. There are many sorts of leaf resources, like machines that split a batch of material into smaller units, or machines that take two batches with different material and put them together. In this paper, we have chosen to specify machines that only perform a transformation on a piece of material, such as cutting off a piece or driving in a screw. So a leaf resource always needs exactly one piece of material in order to execute its operation. This piece of material is used throughout the entire manufacturing process. The general specification of these *transformers* can be found in the process module `Transformers`. This module has two parameters: the first, `self`, specifies the transformer that we are dealing with; the second, `operation`, models the operation that the transformer is able to perform. These parameters are bound to real transformers and operations in the process modules `Transformer-1` to `Transformer-5`. This is where the actual transformers are constructed. When we abstract from the operation that can be performed, the behaviour of the transformers (e.g., the communication with controller and transporter) is always the same. This makes it possible to specify the transformers only once (in the process module `Transformers`), using a few parameters. For example, if we import the process module `Transformers` into the process module `Transformer-1`, and call `TRANSFORMER(transf1)`, we obtain the exact process for the first transformer. The process graph for the transformer process can be found in figure 7. The process describing the transformers consists of the following recurring sequence of actions:
A transformer starts to send a transformation request with the operation it can perform to the controller. This indicates that it can receive a new job:
`send(port(self, contr), tformreq(self, and-also(operation, thats-all-folks)))`
The capability will always consist of a single operation, since transformers can perform only one operation. After a while, the transformer will receive a piece of material, followed by a transformation job. The transformation job prescribes what must happen with the material:
```
sum(mu in MATERIALUNIT, read(port(self, trans), phys(mu))) .
sum(j in JOB, read(port(self, contr), j))
```
Notice that in our case the contents of the transformation job is trivial, since the transformers can only perform one operation. Notice also that the transformer first receives material and then a job. This is a consequence of the command driven manufacturing approach we have chosen (see section 2.3). Next, the transformer performs its operation on the material. After that, it sends a report to the controller, announcing that the execution of the job has finished:

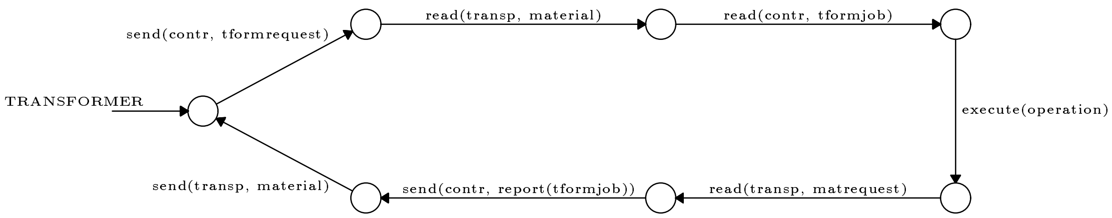


Figure 7: Process graph for the transformer process

```
execute(operation, phys(mu)) . send(port(self, contr), report(j))
```

Finally, the material will be taken away by the transporter. The transporter will first ask for the material by means of a material request, after which the material is given off to the transporter:

```
read(port(self, trans), matreq(mu)) . send(port(self, trans), phys(mu))
```

After this the transformer returns to its initial state.

4.6 The control process

By far the largest process module in the specification in appendix A, specifies the control process. We have to make a distinction between the top control process, the factory controller, and the other manufacturing controllers on a lower level in the hierarchy. The factory controller differs slightly from the other ones, due to the fact that it does not receive transformation jobs from a supercontroller, but orders from a consumer. The specification of the factory controller is in some sense simpler, because we had agreed that the topmost manufacturing system, the factory, should be a single shop. Consequently, it has only one expanded resource (see section 2.1). We shall now first describe the specification of the general manufacturing controller; after that we shall discuss the points on which the specification of the factory controller deviates. Unlike before, we will not show corresponding PSF-code, because the control process is too complicated for this. Instead, we give an elaborate explanation in English. We also added comments to the process module `ManufacturingControllers` in appendix A.

The course of the control process is determined by lots of variables, such as the availability of the resources and the transporter, and the status of the transformation jobs that have been received. For this reason, one will find *conditional* processes in the process module `ManufacturingControllers`. Conditional processes look like `[C] -> P`, and have the following behaviour: if the condition `C`, a boolean expression, rewrites to `true`, the process `P` is executed. If `C` can not be rewritten to `true`, `P` will not be executed.

The process module `ManufacturingControllers` has three parameters: the first one, `self`, specifies which controller is modeled; the second one, `RES-SET`, is the set of resources the controlled manufacturing system has; the third parameter, `recipes`, models the table with the recipes. The recipe-table is an instance of the table from the library. Its entries consist of an operation and a recipe. The operation serves as a key; the recipe is the actual item. When a controller receives a transformation job, it looks in the recipe-table for the operation mentioned in the job. The recipe found in the table is used to execute the job received. The specific controllers are constructed in the process modules `FactoryResourceController` and `Transformer-1-Controller` to `Transformer-5-Controller`. In these modules, the process module `ManufacturingControllers` is imported, the parameter `self` is bound to a controller from the data module `Resources`, the parameter `RES-SET` is bound to one of the resource-sets as listed in the process module `ResourceSets`, and the parameter `recipes` is bound to a recipe-table from the data module `Cookbook`.

The process modelling manufacturing controllers has, besides `self`, three other arguments: a job-queue, a request-table and a progressform-table. These are used to store information about the status of the transporter, the resources, and the transformation jobs received. These queue and tables are all initially empty:

```
MANFACCONTR(self) = MANFACCONTR(self,empty-queue,empty-req-table,empty-pf-table)
```

Unlike transporters and leaf resources, which all start by reporting their availability by means of requests, controllers do not start sending messages. A controller reacts to the messages it receives. We distinguish five kinds of messages:

1. transformation jobs from the supercontroller;
2. requests from the resources;
3. requests from the transporter;
4. reports from the resources;
5. reports from the transporter.

ad 1. When a manufacturing controller receives a transformation job from its supercontroller, it fetches the recipe, mentioned in the job. The controller looks for that recipe in the recipe-table. Next, a progressform is created. The progressform contains the job received, the present location of the material (which is the store), and the recipe. The progressform is added to the progressform-table.
Next, the controller checks whether the first subjob from the recipe can be executed. This is the case if the request-table contains a request for the operation concerned, from the resource on which the subjob must be executed. If that request is present, the resource is blocked to keep it available for the subjob. This is done by making the resource's request empty. Now the material, located at the store, must be brought to the resource. Notice that this is an implementation of the *late transport strategy* (see section 2.3). First look for the availability of the resource, then start transporting material. To bring the material to the resource, the controller looks for a request from the transporter in the request-table. If this request is present, it is removed from the request-table and a job is sent to the transporter. This job instructs the transporter to convey the material from the store to the resource. As soon as a report for this transport job comes in, the subjob will be sent to the resource and the resource will be unblocked. More about this when we discuss point 5: the reception of transport reports.
If the transport request is not present in the request-table, the transport job is put in the job-queue. As soon as the transporter becomes available (i.e., when a transport request comes in), the first transport job from the job-queue will be sent to the transporter. We will say more about how this works, when we discuss point 3: the reception of transport requests.
If the first subjob from the recipe cannot be executed, because the proper request from the resource that must do the job is not present in the request-table, nothing happens for the moment. When a transformation request comes in, the controller determines whether some job is currently waiting for an operation from the request. Therefore, the controller looks at the first subjob from the recipe in each progressform. More about this when we discuss point 2: the reception of transformation requests.
Finally, in all cases, the controller has to check whether it is necessary to send a request to its supercontroller. Later we will say more about how this works and why this is necessary at this point.

ad 2. Resources report their availability by means of transformation requests, in which the operations that can be performed are listed. When exactly these requests are sent and how this works will be discussed later. Newly received requests replace former requests from the same resource in the request-table.
When the controller receives a request from a certain resource, that resource may be blocked (see

above, ad 1). After all, the blocked resource does not know it is blocked. The only reason for blocking resources, is to prevent the controller from assigning multiple jobs to the same resource at the same time. Thus first, the controller searches in the request-table for an empty request from the resource concerned. This should indicate that that resource is blocked. If that is the case, the new request is discarded and nothing happens.
If the resource that sent the request is not blocked, the controller starts investigating whether that resource is needed. The controller does this by looking at the recipe in every progressform in the progressform-table. The topmost suboperation in the recipe is the operation that must be executed next, or that is currently being executed. When this suboperation is completed, it is removed from the recipe. More about this when we discuss point 4: the reception of transformation reports. So when the topmost operation in one of the recipes appears in the request received, this operation must be executed by the resource. It cannot be the case that the operation is currently being executed, because then the resource would have been busy and could not have sent the request.
Once it has been determined which subjob will be executed on the available resource, the controller looks at the present location of the material mentioned in the subjob. The information about the present location of material is kept in the progressforms. If the material is already at the resource that must execute the subjob, there is no need for transportation. The subjob concerned can now be sent to the resource. If there was still an old request in the request-table from this resource, that request is removed.
If transportation is needed, first the resource that must execute the subjob is blocked, so that no other jobs can be sent to it. Next, the controller checks whether the transporter is available. This is done by looking for a request from the transporter in the request-table. If this transport request is present, it is removed from the request-table, and a transport job is sent to the transporter. The job instructs the transporter to convey the material from its present location to the resource where the material has to undergo the next operation. If there is no transport request in the request-table, the transport job is put in the job-queue. The controller will send the first job from the job-queue to the transporter, as soon as the transporter becomes available. This is discussed at point 3: the reception of transport requests.
If none of the operations in the transformation request received appear as the topmost subjob in one of the recipes, the resource that sent the request is not needed currently. In that case, the request is inserted in the request-table. The resource that sent the request can become needed again, as soon as a new subjob appears at the top of the recipe in one of the progressforms. This happens when the controller receives a new transformation job from its supercontroller, or when the execution of a subjob has finished. In that case, the controller will look in the request-table for a request. More about this at points 1 (reception of transformation jobs) and 4 (reception of transformation reports). Because the manufacturing system now has a resource idle, the controller starts investigating whether it can send a request to its supercontroller. More about the sending of requests later.

ad 3. When a request from the transporter comes in, this indicates that the transporter is ready to receive a new job. First, the controller looks in the job-queue to see whether there are transport jobs waiting for the transporter. If there are transport jobs in the queue, the first one is removed from the queue, and sent to the transporter. If this is not the case, the transport request is stored in the request-table. The key of the items (the requests) in the request-table is the resource that sent the request. In case of a transport request, the relative resource `trans` is used as a key. We have seen above that, when the transporter becomes needed, the controller will first look in the request-table for a transport request.

ad 4. When a manufacturing controller receives a report from one of its resources, this means that the concerned resource has finished executing a subjob. That subjob must then be removed from the recipe in the progressform of the superjob. If this is the last subjob in the recipe, the execution of

the superjob is now finished, and the material must be transported back to the store. So next, the controller checks whether the transporter is available. If there is a request from the transporter in the request-table, that request is removed from the table and a transport job is sent. It instructs the transporter to convey the material from its present location to the store.
If there is no transport request in the request-table, the transport job is put in the job-queue. As we have seen at point 3 (the reception of transport requests), the first transport job in the job-queue is sent to the transporter, as soon as the transporter becomes available.
If the subjob, for which the report has been received, is not the last subjob from the superjob's recipe, the superjob is not finished yet. The next subjob must now be executed. The controller looks in the request-table to see if there is a request for the operation concerned, from the resource that must execute the next subjob. What happens if this is true, has already been described at point 2: the reception of a transformation request. This situation is comparable to the case, where the resource that sent the request is needed for the execution of a subjob.
If the next subjob can not be executed because the proper request is not in the request-table, nothing further happens. We have seen at point 2 (the reception of transformation requests), that as soon as a resource becomes available, the controller checks whether it is needed. If so, proper action is taken.

ad 5. When a transport report comes in, the controller first looks at the destination-field in the transport job, for which the report has come in. If this is `store`, it means that all the suboperations from the recipe have been executed. The execution of the superjob has finished. There are two things left to be done by the controller: sending a report for the superjob to the supercontroller, and deleting the corresponding progressform from the progressform-table.
If the destination-field in the transport job for which the report has come in is not `store`, this means that material has been transported to a resource where it has to undergo an operation. As we have seen at points 1, 2 and 4 above, resources get blocked, if the material first has to be transported to it. According to our *command driven manufacturing* (see section 2.3), the execution of the operation must be triggered by sending the proper subjob to the resource. This is what must happen now. Therefore, the controller unblocks the resource, and sends a transformation job for the uppermost operation from the superjob's recipe to it. The controller also updates the location-field in the superjob's progressform.

Request sending
From time to time, a manufacturing controller sends a request to its supercontroller. It lists all the operations that can be performed by the expanded resource that it controls. Every time an operation becomes executable, the controller sends a new request. If the supercontroller then puts that request in the request-table, any request from the same resource that is already present in the table is discarded. As we have seen above, the supercontroller removes the complete request from a resource from the request-table, if it sends a job to that resource. But a job contains only one operation, whereas there might have been numerous operations in the request. In that case, the controller of the resource sends a new request to its supercontroller, listing the operations it still can perform. Naturally, this only holds for expanded resources. A leaf resource can only perform one operation at the time.
So there are two occasions on which a manufacturing controller must send a request to its supercontroller. When it receives a job from the supercontroller, and when it stores a request in the request-table. The latter is the case when the controller receives a request that it cannot use immediately.
We will now take a look at how the sending of requests works. An expanded resource can perform an operation, if its resources can perform all the necessary suboperations. So for every operation in its recipe-table, the controller checks whether there are requests in the request-table for all the suboperations in the recipe. If that is the case, the operation is added to the list that will be sent with the request. If this results in at least one operation that can be performed, a request with the list of executable operations is sent to the supercontroller. The resource mentioned in that request is

not `self`, since that would give the name of the controller. Instead, the request is "signed" with the name of the expanded resource that is controlled.

The factory controller
At this point, we have given an elaborate description of the control process for all manufacturing controllers, except for the factory controller. Factory controllers differ in two ways from other manufacturing controllers. First of all, a factory controller does not have a supercontroller within the factory, but it receives messages from and sends messages to a consumer. Therefore, the factory controller receives orders instead of transformation jobs. The progression of the execution of these orders is recorded in order progressforms, stored in the progressform-table. This difference does not lead to real deviations in the control process. Some functions that are used had to be defined twice in the data module `Control-functions`, so that they can work with both jobs and orders.
A more important difference is the fact that the factory controller has only one expanded resource (section 2.1). Consequently, the process module `FactoryControllers` does not have a parameter `RES-SET` of type `set of RESOURCE`, but a parameter `res` of type `RESOURCE`. This difference also has some consequences for the process specification. Where in the process expression of other controllers one can find

```
sum(res in RES-SET, read(port(self, res), ...))
```

to denote that something is received from one of the resources, we can now omit the `sum`. Another consequence is that the factory transporter only has to move between the factory store and the factory resource. Material cannot be transported between different resources for the execution of various subjobs, because there is only one resource.
The final difference between the factory controller and other controllers is that the factory controller does not send requests. These requests would have to be converted to messages for the consumer. In order not to make things more complicated than they already are, we have chosen not to do this. As a consequence, the factory controller can receive an unlimited number of orders from the consumer.

4.7 Further process specification

Now that we have seen the specification of the basic processes of a manufacturing system, we will take a look at the higher level processes: the expanded resources. These are the five transformation resources, the factory resource and the factory itself. We will see how these parts of the manufacturing system have been assembled from parts that are lower in the hierarchy. Finally, a consumer and the resulting market are being looked at. First, however, we shall discuss the resource sets.

Resource sets. In the process module `ResourceSets` the sets of resources of all manufacturing systems are listed. The parameter `RES-SET` in the process module `ManufacturingControllers` can be bound to the sets listed in this module, when an actual controller is specified.

Transformation resources. The five transformation resources are formed in the process modules `TransformationResource-1` to `TransformationResource-5`. As an example, we will now discuss the first transformation resource. The process `TRANSFRES-1` is the overall process. It is the parallel composition of the basic processes in the first transformation resource. These are a controller, a store, a transporter, and a transformer. This is also the place where the `self` arguments from the various process expressions are bound to actual resources:

```
TRANSF-1-CONTR = MANFACCONTR(tr-1-contr)
TRANSF-1-STORE = STORE(tr-1-store)
TRANSF-1-TRANSP = TRANSPORTER(tr-1-transp)
TRANSF-1 = TRANSFORMER(transf1)
```

We need to enforce communication between these components. We do not want that one component can send a message to another component if that other component can not receive the message. Therefore, we forbid separate `read` and `send` actions to happen, by putting these

actions into the set `H` and putting `encaps(H, ...)`. `read` and `send` actions must now wait for the other party to communicate with, and can no longer take place separately.
Naturally, we must only do this for `read` and `send` actions between components within the first transformation resource. This means that we must only block these actions for the ports, mentioned in the set P, connecting the components of the first transformation resource:

```
port(tr-1-transp, contr)
port(transf1, contr)
port(tr-1-transp, store)
port(transf1, trans)
```

Factory resource. In the process module `FactoryResource`, the five transformation resources that we formed above, are put in parallel with a controller, a store, and a transporter. This results in the specification of the factory resource process. Again, we block separate `read` and `send` actions over the ports connecting the store, transporter, controller, and the five transformation resources. At this level, we see the transformation resources as atomic blocks.

Factory. In the process module `Factory`, we define the factory process as the parallel composition of the factory resource, the factory store, a transporter, and the factory controller. At this level, we see the factory resource as one atomic component. We block separate `read` and `send` actions over the ports connecting the components of the factory.

Consumer and market. The consumer can best be seen as the supercontroller of the factory controller. But the consumer is not a part of the factory. It can perform two actions: sending an order to the factory controller, and receiving a report from the factory controller. These order reports tell the consumer that a certain order has been completed.
In the process module `Consumer`, the process modelling the consumer has an identifier as an argument (`CONSUMER(oid)`). This is the identifier that will be sent with the next order. This identifier initially has value zero:

```
CONSUMER = CONSUMER(id(nat(^0)))
```

After the consumer has sent an order for a certain product, it returns to its initial state, and increments the id-number:

```
sum(p in PRODUCT, send(port(cons, fac), order(oid, p))) . CONSUMER(next(oid))
```

One might remark that it is strange that the consumer assigns an identifier to orders, and that this identifier is then being used in the entire factory. This is indeed not the way things work in a real market. There, every manufacturing system has organised its own administration. Here, we try to keep things simple.
After the consumer has received a report, it returns to its initial state as well, but it does not change its argument:

```
sum(o in ORDER, read(port(cons, fac), report(o))) . CONSUMER(oid)
```

One can see that the consumer process is nothing more than a dummy that gives input to and/or accepts output from the factory controller.

In the process module `Market`, the factory and the consumer are put in parallel. At this level we see the factory as one atomic component. We block separate `read` and `send` actions over the port connecting the consumer and the factory.

5 Conclusions and future work

We have presented a PSF-specification of a manufacturing system that is very flexible. The specification presented here has some major advantages in comparison with the one from [HP92]. In that specification there is no route flexibility at all. It has been built rigidly into the controllers that, when product X comes from resource Y, it should go to resource Z next. To change these routes, the controllers would have to be rebuilt. In the present specification, the manufacturing controllers have the recipes for the products that are to be manufactured, as a parameter. Using these recipes, the controllers can decide which route a piece of material must follow along the resources. Another advantage is that resources can be easily added to and/or removed from manufacturing systems. This is possible, because the set of resources that must be controlled, is a parameter of every controller. Finally, in [HP92] two resources became connected by assigning the same hard number to the ports of the resources. Here, we simply say that the ports of these resources are the same. This approach avoids the creation of a long list of port names, which does not contribute to the clarity of the specification. Different components are linked dynamically in this paper. The use of relative resources helps to make the specification of the different components as modular as possible.

It has proven to be very hard to translate a specification in ProcessTalk into a specification in PSF. This, we think, is due to the fact that ProcessTalk is an object oriented language, whereas PSF is strictly algebraic.
We think that it is impossible to say that one language is "better" than the other. Both languages provide a formalism, in which systems can be specified. Apart from the fundamental difference, one can say that PSF stays at a higher level of abstraction. ProcessTalk is equipped with realtime features, by means of which statistical information about throughput and lead time of products can be given. In ProcessTalk, a system can be built interactively using a graphical environment.
Because of the differences mentioned above, we imagine that up to the level of simulation, ProcessTalk has more advantages than PSF, when used for industrial purposes. However, the theory behind PSF offers the possibility to proof that the system specified is correct or deadlock-free. It is our belief that as soon as these features can be captured in an efficient computer-tool, the value of PSF for the industry will increase dramatically.
Both ProcessTalk and PSF are useful to gain insight into the behaviour of complex systems.

Future work
In this paper, some simplifications were made. A follow-up on this project could modify the present specification on the following points:

- In the present situation, exactly one piece of material is needed for the execution of each recipe. The different leaf resources operate on this material one after another. It would be better to specify *batches* of material for the manufacturing of *batches* of products. Consequently, resources that can handle recipes in which batches of material are being split or combined, should be specified.
- In the present specification, as well as in [Smi92], all manufacturing systems are equipped with a late transport strategy. One could try to model an early transport strategy for the manufacturing systems that have no leaf resources.
- It may be desirable to have a manufacturing system with multiple resources of the same kind. When a subjob must be executed on a resource of that kind, the recipe should not specify on which one that happens. The controller would have to make this decision on the basis of the status of these resources.
- Producers and consumers have been modelled very marginally. It must be possible to specify producers, from which the factory can order raw material.

References

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra.* Cambridge Tracts in Theoretical Computer Science 17. Cambridge University Press, 1990.

[BB91] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Journal of Formal Aspects of Computing Science*, 3(2):142–188, 1991.

[BB93] J.C.M. Baeten and J.A. Bergstra. Real space process algebra. *Journal of Formal Aspects of Computing Science*, 5(6), 1993.

[BHK89a] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BHK89b] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In *Algebraic specification* [BHK89a], pages 1–66.

[BHK89c] J.A. Bergstra, J. Heering, and P. Klint. A simple programming language and its implementation. In *Algebraic specification* [BHK89a], pages 67–132.

[BMW91] J.A. Bergstra, S. Mauw, and F. Wiedijk. Uniform algebraic specifications of finite sets with equality. *International Journal of Foundations of Computer Science*, 2(1):43–65, 1991.

[BP93] J.J. Brunekreef and A. Ponse. An algebraic specification of a model factory, part 4. Report P9316, University of Amsterdam, 1993.

[Bru93] J.J. Brunekreef. Two simple protocols for local area networks. In Mauw and Veltink [MV93], pages 135–160.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[FvW92] W.J. Fokkink and J.J. van Wamel. An algebraic specification of interest products. Confidential Report CR9201, Programming Research Group, University of Amsterdam, Amsterdam, 1992.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, Massachusetts, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[HP92] J.A. Hillebrand and A. Ponse. An algebraic specification of a model factory, part 2. Report P9214, University of Amsterdam, 1992.

[Int89] International Organisation for Standardisation. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/IS8807, 1989.

[Mau90] S. Mauw. Process algebra as a tool for the specification and verification of CIM–architectures. In Baeten [Bae90], pages 53–80.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[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.

[MV93] S. Mauw and G.J. Veltink, editors. *Algebraic Specification of Communication Protocols.* Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.

[Pon92] A. Ponse. *Process algebra with data.* PhD thesis, University of Amsterdam, 1992.

[PV93] A. Ponse and J.A. Verschuren. An algebraic specification of a model factory, part 3. Report P9303, University of Amsterdam, 1993.

[RW90] J.E. Rooda and A.M. Wortmann. The process-interaction environment user manual. Memorandum, Eindhoven University of Technology, 1990.

[Smi92] G.H. Smit. *A hierarchical control architecture for job-shop manufacturing systems.* PhD thesis, Eindhoven University of Technology, 1992.

[Vel93] G.J. Veltink. The PSF toolkit. *Computer Networks and ISDN Systems*, 25:875–898, 1993.

[vVvW92] S.F.M. van Vlijmen and A. van Waveren. An algebraic specification of a model factory. Report P9209, University of Amsterdam, 1992.

[vW92] J.J. van Wamel. A study of a one bit sliding window protocol. Report P9212, Programming Research Group, University of Amsterdam, 1992.

[vW93] J.J. van Wamel. A library for PSF. Report P9301, Programming Research Group, University of Amsterdam, 1993.

[Wor91] A.M. Wortmann. *Modelling and simulation of industrial systems.* PhD thesis, Eindhoven University of Technology, 1991.

A Listing

(See next 15 pages.)

```
data module Ids
begin
  exports
    begin
      sorts
        ID
      functions
        default-id:           ->   ID
        id:         NATURAL   ->   ID
        next:       ID        ->   ID
        eq:         ID # ID   ->   BOOLEAN
    end
  imports
    Naturals
  variables
    n, n1, n2: -> NATURAL
  equations
    [I1] id(n) = id(nat(^0)) when gt(n, nat(^1^0)) = true
    [I2] next(default-id) = default-id
    [I3] next(id(n)) = id(n + nat(^1))
    [I4] eq(default-id, default-id) = true
    [I5] eq(default-id, id(n)) = false
    [I6] eq(id(n), default-id) = false
    [I7] eq(id(n1), id(n2)) = eq(n1, n2)
end Ids
```

```
data module Material
begin
  exports
    begin
      sorts
        MATERIALUNIT,
        MATERIAL
      functions
        matunit-nil:                                ->   MATERIALUNIT
        matunit-1:                                  ->   MATERIALUNIT
        matunit-2:                                  ->   MATERIALUNIT
        matunit-3:                                  ->   MATERIALUNIT
        eq:        MATERIALUNIT # MATERIALUNIT   ->   BOOLEAN
        phys:      MATERIALUNIT                  ->   MATERIAL
    end
  imports
    Naturals
  functions
    to-nat: MATERIALUNIT   ->   NATURAL
  variables
    mu1, mu2: -> MATERIALUNIT
  equations
    [M1] matunit-nil = matunit-1
    [M2] to-nat(matunit-1) = nat(^0)
    [M3] to-nat(matunit-2) = nat(^1)
    [M4] to-nat(matunit-3) = nat(^2)
    [M5] eq(mu1, mu2) = eq(to-nat(mu1), to-nat(mu2))
end Material
```

```
data module Products
begin
  exports
    begin
      sorts
        PRODUCT
      functions
        product-nil:               ->   PRODUCT
        prodA:                     ->   PRODUCT
        prodB:                     ->   PRODUCT
        prodC:                     ->   PRODUCT
        prodD:                     ->   PRODUCT
        eq:    PRODUCT # PRODUCT   ->   BOOLEAN
        mat:   PRODUCT             ->   MATERIALUNIT
    end
  imports
    Material
  functions
    to-nat: PRODUCT   ->   NATURAL
  variables
    p1, p2: -> PRODUCT
  equations
    [Pr1]  product-nil = prodA
    [Pr2]  to-nat(prodA) = nat(^0)
    [Pr3]  to-nat(prodB) = nat(^1)
    [Pr4]  to-nat(prodC) = nat(^2)
    [Pr5]  to-nat(prodD) = nat(^3)
    [Pr6]  eq(p1, p2) = eq(to-nat(p1), to-nat(p2))
    [Pr7]  mat(prodA) = matunit-1
    [Pr8]  mat(prodB) = matunit-2
    [Pr9]  mat(prodC) = matunit-2
    [Pr10] mat(prodD) = matunit-3
end Products
```

```
data module Operations
begin
  exports
    begin
      sorts
        OPERATION
      functions
        operation-nil:                             ->   OPERATION
        A-proc:                                    ->   OPERATION
        B-proc:                                    ->   OPERATION
        C-proc:                                    ->   OPERATION
        D-proc:                                    ->   OPERATION
        tr-1-op:                                   ->   OPERATION
        tr-2-op:                                   ->   OPERATION
        tr-3-op:                                   ->   OPERATION
        tr-4-op:                                   ->   OPERATION
        tr-5-op:                                   ->   OPERATION
        transformation1:                           ->   OPERATION
        transformation2:                           ->   OPERATION
        transformation3:                           ->   OPERATION
        transformation4:                           ->   OPERATION
        transformation5:                           ->   OPERATION
        eq:              OPERATION # OPERATION   ->   BOOLEAN
    end
  imports
    Naturals
  functions
    to-nat: OPERATION   ->   NATURAL
  variables
    op1, op2: -> OPERATION
  equations
    [Op1]  operation-nil = A-proc
    [Op2]  to-nat(A-proc) = nat(^0)
    [Op3]  to-nat(B-proc) = nat(^1)
    [Op4]  to-nat(C-proc) = nat(^2)
```

```
    [Op5]  to-nat(D-proc) = nat(^3)
    [Op6]  to-nat(tr-1-op) = nat(^4)
    [Op7]  to-nat(tr-2-op) = nat(^5)
    [Op8]  to-nat(tr-3-op) = nat(^6)
    [Op9]  to-nat(tr-4-op) = nat(^7)
    [Op10] to-nat(tr-5-op) = nat(^8)
    [Op11] to-nat(transformation1) = nat(^9)
    [Op12] to-nat(transformation2) = nat(^1^0)
    [Op13] to-nat(transformation3) = nat(^1^1)
    [Op14] to-nat(transformation4) = nat(^1^2)
    [Op15] to-nat(transformation5) = nat(^1^3)
    [Op16] eq(op1, op2) = eq(to-nat(op1), to-nat(op2))
end Operations



data module Resources
begin
  exports
    begin
      sorts
        RESOURCE
      functions
        resource-nil:                                ->   RESOURCE
        contr:                                       ->   RESOURCE
        trans:                                       ->   RESOURCE
        store:                                       ->   RESOURCE
        intern:                                      ->   RESOURCE
        extern:                                      ->   RESOURCE
        outside:                                     ->   RESOURCE
        cons:                                        ->   RESOURCE
        fac:                                         ->   RESOURCE
        fac-contr:                                   ->   RESOURCE
        fac-transp:                                  ->   RESOURCE
        fac-store:                                   ->   RESOURCE
        facres:                                      ->   RESOURCE
        facres-contr:                                ->   RESOURCE
        facres-transp:                               ->   RESOURCE
        facres-store:                                ->   RESOURCE
        transfres-1:                                 ->   RESOURCE
        tr-1-contr:                                  ->   RESOURCE
        tr-1-transp:                                 ->   RESOURCE
        tr-1-store:                                  ->   RESOURCE
        transf1:                                     ->   RESOURCE
        transfres-2:                                 ->   RESOURCE
        tr-2-contr:                                  ->   RESOURCE
        tr-2-transp:                                 ->   RESOURCE
        tr-2-store:                                  ->   RESOURCE
        transf2:                                     ->   RESOURCE
        transfres-3:                                 ->   RESOURCE
        tr-3-contr:                                  ->   RESOURCE
        tr-3-transp:                                 ->   RESOURCE
        tr-3-store:                                  ->   RESOURCE
        transf3:                                     ->   RESOURCE
        transfres-4:                                 ->   RESOURCE
        tr-4-contr:                                  ->   RESOURCE
        tr-4-transp:                                 ->   RESOURCE
        tr-4-store:                                  ->   RESOURCE
        transf4:                                     ->   RESOURCE
        transfres-5:                                 ->   RESOURCE
        tr-5-contr:                                  ->   RESOURCE
        tr-5-transp:                                 ->   RESOURCE
        tr-5-store:                                  ->   RESOURCE
        transf5:                                     ->   RESOURCE
        controlled-resource: RESOURCE                ->   RESOURCE
        eq:                  RESOURCE # RESOURCE   ->   BOOLEAN
  end
imports
  Naturals
functions
  to-nat: RESOURCE   ->   NATURAL
variables
  r, r1, r2: -> RESOURCE
equations
  [Res1]  resource-nil = fac
  [Res2]  controlled-resource(fac-contr) = fac
  [Res3]  controlled-resource(facres-contr) = facres
  [Res4]  controlled-resource(tr-1-contr) = transfres-1
  [Res5]  controlled-resource(tr-2-contr) = transfres-2
  [Res6]  controlled-resource(tr-3-contr) = transfres-3
  [Res7]  controlled-resource(tr-4-contr) = transfres-4
  [Res8]  controlled-resource(tr-5-contr) = transfres-5
  [Res9]  controlled-resource(r) = r
            when or(eq(r, fac-contr),
                    or(eq(r, facres-contr),
                       or(eq(r, tr-1-contr),
                          or(eq(r, tr-2-contr),
                             or(eq(r, tr-3-contr),
                                or(eq(r, tr-4-contr),
                                   eq(r, tr-5-contr))))))) = false
  [Res10] to-nat(contr)          = nat(^0)
  [Res11] to-nat(trans)          = nat(^1)
  [Res12] to-nat(store)          = nat(^2)
  [Res13] to-nat(intern)         = nat(^3)
  [Res14] to-nat(extern)         = nat(^4)
  [Res15] to-nat(outside)        = nat(^5)
  [Res16] to-nat(cons)           = nat(^6)
  [Res17] to-nat(fac)            = nat(^7)
  [Res18] to-nat(fac-contr)      = nat(^8)
  [Res19] to-nat(fac-transp)     = nat(^9)
  [Res20] to-nat(fac-store)      = nat(^1^0)
  [Res21] to-nat(facres)         = nat(^1^1)
  [Res22] to-nat(facres-contr)   = nat(^1^2)
  [Res23] to-nat(facres-transp)  = nat(^1^3)
  [Res24] to-nat(facres-store)   = nat(^1^4)
  [Res25] to-nat(transfres-1)    = nat(^1^5)
  [Res26] to-nat(tr-1-contr)     = nat(^1^6)
  [Res27] to-nat(tr-1-transp)    = nat(^1^7)
  [Res28] to-nat(tr-1-store)     = nat(^1^8)
  [Res29] to-nat(transf1)        = nat(^1^9)
  [Res30] to-nat(transfres-2)    = nat(^2^0)
  [Res31] to-nat(tr-2-contr)     = nat(^2^1)
  [Res32] to-nat(tr-2-transp)    = nat(^2^2)
  [Res33] to-nat(tr-2-store)     = nat(^2^3)
  [Res34] to-nat(transf2)        = nat(^2^4)
  [Res35] to-nat(transfres-3)    = nat(^2^5)
  [Res36] to-nat(tr-3-contr)     = nat(^2^6)
  [Res37] to-nat(tr-3-transp)    = nat(^2^7)
  [Res38] to-nat(tr-3-store)     = nat(^2^8)
  [Res39] to-nat(transf3)        = nat(^2^9)
  [Res40] to-nat(transfres-4)    = nat(^3^0)
  [Res41] to-nat(tr-4-contr)     = nat(^3^1)
  [Res42] to-nat(tr-4-transp)    = nat(^3^2)
  [Res43] to-nat(tr-4-store)     = nat(^3^3)
  [Res44] to-nat(transf4)        = nat(^3^4)
  [Res45] to-nat(transfres-5)    = nat(^3^5)
  [Res46] to-nat(tr-5-contr)     = nat(^3^6)
  [Res47] to-nat(tr-5-transp)    = nat(^3^7)
  [Res48] to-nat(tr-5-store)     = nat(^3^8)
  [Res49] to-nat(transf5)        = nat(^3^9)
```

```
    [Res50] eq(r1, r2) = eq(to-nat(r1), to-nat(r2))
end Resources



data module Ports
begin
  exports
    begin
      sorts
        PORT
      functions
        port: RESOURCE # RESOURCE   ->   PORT
    end
  imports
    Resources
  equations
    [Po1]  port(fac-contr, trans) = port(fac-transp, contr)
    [Po2]  port(fac-contr, facres) = port(facres-contr, contr)
    [Po3]  port(facres-contr, trans) = port(facres-transp, contr)
    [Po4]  port(facres-contr, transfres-1) = port(tr-1-contr, contr)
    [Po5]  port(facres-contr, transfres-2) = port(tr-2-contr, contr)
    [Po6]  port(facres-contr, transfres-3) = port(tr-3-contr, contr)
    [Po7]  port(facres-contr, transfres-4) = port(tr-4-contr, contr)
    [Po8]  port(facres-contr, transfres-5) = port(tr-5-contr, contr)
    [Po9]  port(tr-1-contr, trans) = port(tr-1-transp, contr)
    [Po10] port(tr-1-contr, transf1) = port(transf1, contr)
    [Po11] port(tr-2-contr, trans) = port(tr-2-transp, contr)
    [Po12] port(tr-2-contr, transf2) = port(transf2, contr)
    [Po13] port(tr-3-contr, trans) = port(tr-3-transp, contr)
    [Po14] port(tr-3-contr, transf3) = port(transf3, contr)
    [Po15] port(tr-4-contr, trans) = port(tr-4-transp, contr)
    [Po16] port(tr-4-contr, transf4) = port(transf4, contr)
    [Po17] port(tr-5-contr, trans) = port(tr-5-transp, contr)
    [Po18] port(tr-5-contr, transf5) = port(transf5, contr)
    [Po19] port(cons, fac) = port(fac-contr, outside)
    [Po20] port(fac-store, intern) = port(fac-transp, store)
    [Po21] port(fac-transp, facres) = port(facres-store, extern)
    [Po22] port(facres-store, intern) = port(facres-transp, store)
    [Po23] port(facres-transp, transfres-1) = port(tr-1-store, extern)
    [Po24] port(tr-1-store, intern) = port(tr-1-transp, store)
    [Po25] port(tr-1-transp, transf1) = port(transf1, trans)
    [Po26] port(facres-transp, transfres-2) = port(tr-2-store, extern)
    [Po27] port(tr-2-store, intern) = port(tr-2-transp, store)
    [Po28] port(tr-2-transp, transf2) = port(transf2, trans)
    [Po29] port(facres-transp, transfres-3) = port(tr-3-store, extern)
    [Po30] port(tr-3-store, intern) = port(tr-3-transp, store)
    [Po31] port(tr-3-transp, transf3) = port(transf3, trans)
    [Po32] port(facres-transp, transfres-4) = port(tr-4-store, extern)
    [Po33] port(tr-4-store, intern) = port(tr-4-transp, store)
    [Po34] port(tr-4-transp, transf4) = port(transf4, trans)
    [Po35] port(facres-transp, transfres-5) = port(tr-5-store, extern)
    [Po36] port(tr-5-store, intern) = port(tr-5-transp, store)
    [Po37] port(tr-5-transp, transf5) = port(transf5, trans)
end Ports



data module Orders
begin
  exports
    begin
      sorts
        ORDER
      functions
        order-nil:                ->   ORDER
        order:     ID # PRODUCT   ->   ORDER
        id:        ORDER          ->   ID
        mat:       ORDER          ->   MATERIALUNIT
    end
  imports
    Ids, Products
  variables
    oid: -> ID
    p:   -> PRODUCT
  equations
    [O1] order-nil = order(default-id, product-nil)
    [O2] id(order(oid, p)) = oid
    [O3] mat(order(oid, p)) = mat(p)
end Orders



data module Jobs
begin
  exports
    begin
      sorts
        JOB
      functions
        job-nil:                                            ->   JOB
        tformjob:  ID # OPERATION # RESOURCE # MATERIALUNIT   ->   JOB
        tportjob:  ID # RESOURCE # RESOURCE # MATERIALUNIT    ->   JOB
        id:        JOB                                      ->   ID
        operation: JOB                                      ->   OPERATION
        resource:  JOB                                      ->   RESOURCE
        orig:      JOB                                      ->   RESOURCE
        dest:      JOB                                      ->   RESOURCE
        mat:       JOB                                      ->   MATERIALUNIT
    end
  imports
    Ids, Material, Operations, Resources
  variables
    jid:            -> ID
    op:             -> OPERATION
    res, res1, res2: -> RESOURCE
    mu:             -> MATERIALUNIT
  equations
    [J1]  job-nil =
            tformjob(default-id, operation-nil, resource-nil, matunit-nil)
    [J2]  id(tformjob(jid, op, res, mu)) = jid
    [J3]  id(tportjob(jid, res1, res2, mu)) = jid
    [J4]  operation(tformjob(jid, op, res, mu)) = op
    [J5]  resource(tformjob(jid, op, res, mu)) = res
    [J6]  resource(tportjob(jid, res1, res2, mu)) = trans
    [J7]  orig(tportjob(jid, res1, res2, mu)) = res1
    [J8]  dest(tportjob(jid, res1, res2, mu)) = res2
    [J9]  mat(tformjob(jid, op, res, mu)) = mu
    [J10] mat(tportjob(jid, res1, res2, mu)) = mu

-- "senseless" function applications
    [J11] operation(tportjob(jid, res1, res2, mu)) = operation-nil
    [J12] orig(tformjob(jid, op, res, mu)) = resource-nil
    [J13] dest(tformjob(jid, op, res, mu)) = resource-nil
end Jobs



data module Requests
begin
```

```
  exports
    begin
      sorts
        CAPABILITY,
        REQUEST,
        MATERIALREQUEST
      functions
        request-nil:                                ->   REQUEST
        tportreq:                                   ->   REQUEST
        tformreq:        RESOURCE # CAPABILITY      ->   REQUEST
        thats-all-folks:                            ->   CAPABILITY
        and-also:        OPERATION # CAPABILITY     ->   CAPABILITY
        cap:             REQUEST                    ->   CAPABILITY
        eq:              CAPABILITY # CAPABILITY    ->   BOOLEAN
        matreq:          MATERIALUNIT               ->   MATERIALREQUEST
    end
  imports
    Material, Operations, Resources
  variables
    op, op1, op2:    -> OPERATION
    res, res1, res2: -> RESOURCE
    cp, cp1, cp2:    -> CAPABILITY
  equations
    [Req1] request-nil = tformreq(resource-nil, thats-all-folks)
    [Req2] cap(tformreq(res, cp)) = cp
    [Req3] eq(thats-all-folks, thats-all-folks) = true
    [Req4] eq(thats-all-folks, and-also(op, cp)) = false
    [Req5] eq(and-also(op, cp), thats-all-folks) = false
    [Req6] eq(and-also(op1, cp1), and-also(op2, cp2)) =
             and(eq(op1, op2), eq(cp1, cp2))

-- "senseless" function applications
    [Req7] cap(tportreq) = thats-all-folks
end Requests



data module Reports
begin
  exports
    begin
      sorts
        REPORT
      functions
        report: ORDER   ->   REPORT
        report: JOB     ->   REPORT
    end
  imports
    Orders, Jobs
end Reports



data module Recipes
begin
  exports
    begin
      sorts
        PROCESSSTEP,
        RECIPE
      functions
        execute-on: OPERATION # RESOURCE   ->   PROCESSSTEP
        finished:                          ->   RECIPE
        sequence:   PROCESSSTEP # RECIPE   ->   RECIPE
    end
  imports
    Operations, Resources
end Recipes



data module ProgressForms
begin
  exports
    begin
      sorts
        PROGRESSFORM
      functions
        progform-nil:                                 ->   PROGRESSFORM
        ordprogform:  ORDER # RESOURCE # RECIPE       ->   PROGRESSFORM
        jobprogform:  JOB # RESOURCE # RECIPE         ->   PROGRESSFORM
        order:        PROGRESSFORM                    ->   ORDER
        job:          PROGRESSFORM                    ->   JOB
        loc:          PROGRESSFORM                    ->   RESOURCE
    end
  imports
    Recipes, Orders, Jobs
  variables
    o:   ->   ORDER
    j:   ->   JOB
    res: ->   RESOURCE
    rec: ->   RECIPE
  equations
    [Pf1] progform-nil = ordprogform(order-nil, resource-nil, finished)
    [Pf2] order(ordprogform(o, res, rec)) = o
    [Pf3] job(jobprogform(j, res, rec)) = j
    [Pf4] loc(ordprogform(o, res, rec)) = res
    [Pf5] loc(jobprogform(j, res, rec)) = res

-- "senseless" function applications
    [Pf6] order(jobprogform(j, res, rec)) = order-nil
    [Pf7] job(ordprogform(o, res, rec)) = job-nil
end ProgressForms



data module Job-queues
begin
  imports
    Queues{
      Queue-Parameters bound by
        [Q-ELEMENT -> JOB,
         queue-nil -> job-nil]
        to Jobs
      renamed by
        [QUEUE -> JOB-QUEUE]
    }
end Job-queues



data module Request-tables
begin
  imports
    Tables{
      Keys bound by
        [KEY -> RESOURCE,
         eq  -> eq]
        to Requests
      Items bound by
```

```
        [ITEM      -> REQUEST,
         table-nil -> request-nil]
        to Requests
      renamed by
        [TABLE       -> REQ-TABLE,
         ENTRY       -> REQ-ENTRY,
         empty-table -> empty-req-table]
    }
end Request-tables


data module Progform-tables
begin
  imports
    Tables{
      Keys bound by
        [KEY -> ID,
         eq  -> eq]
        to Ids
      Items bound by
        [ITEM      -> PROGRESSFORM,
         table-nil -> progform-nil]
        to ProgressForms
      renamed by
        [TABLE -> PF-TABLE,
         ENTRY -> PF-ENTRY,
         empty-table -> empty-pf-table]
    }
end Progform-tables


data module Recipe-tables
begin
  imports
    Tables{
      Keys bound by
        [KEY -> OPERATION,
         eq  -> eq]
        to Operations
      Items bound by
        [ITEM      -> RECIPE,
         table-nil -> finished]
        to Recipes
      renamed by
        [TABLE       -> RECIPE-TABLE,
         ENTRY       -> RECIPE-ENTRY,
         empty-table -> empty-rec-table]
    }
end Recipe-tables


data module Product-tables
begin
  imports
    Tables{
      Keys bound by
        [KEY -> PRODUCT,
         eq  -> eq]
        to Products
      Items bound by
        [ITEM      -> RECIPE,
         table-nil -> finished]
        to Recipes
      renamed by
        [TABLE       -> PRODUCT-TABLE,
         ENTRY       -> PRODUCT-ENTRY,
         empty-table -> empty-prod-table]
    }
end Product-tables


data module Material-tables
begin
  exports
    begin
      sorts
        MAT-TABLE
      functions
        empty-mat-table:                               ->   MAT-TABLE
        insert:          MAT-TABLE # MATERIALUNIT   ->   MAT-TABLE
        delete:          MAT-TABLE # MATERIALUNIT   ->   MAT-TABLE
        in-table:        MAT-TABLE # MATERIALUNIT   ->   BOOLEAN
        length:          MAT-TABLE                  ->   NATURAL
    end
  imports
    Material
  variables
    mu, mu1, mu2: ->   MATERIALUNIT
    mat-tab:      ->   MAT-TABLE
  equations
    [Mt1] delete(empty-mat-table, mu) = empty-mat-table
    [Mt2] delete(insert(mat-tab, mu), mu) = mat-tab
    [Mt3] delete(insert(mat-tab, mu1), mu2) = insert(delete(mat-tab, mu2), mu1)
            when eq(mu1, mu2) = false
    [Mt4] in-table(empty-mat-table, mu) = false
    [Mt5] in-table(insert(mat-tab, mu1), mu2) =
            or(eq(mu1, mu2), in-table(mat-tab, mu2))
    [Mt6] length(empty-mat-table) = nat(^0)
    [Mt7] length(insert(mat-tab, mu)) = length(mat-tab) + nat(^1)
end Material-tables


data module Cookbook
begin
  exports
    begin
      functions
        factory-recipes:   ->   PRODUCT-TABLE
        facres-recipes:    ->   RECIPE-TABLE
        tr-1-recipes:      ->   RECIPE-TABLE
        tr-2-recipes:      ->   RECIPE-TABLE
        tr-3-recipes:      ->   RECIPE-TABLE
        tr-4-recipes:      ->   RECIPE-TABLE
        tr-5-recipes:      ->   RECIPE-TABLE
    end
  imports
    Product-tables, Recipe-tables
  equations
    [C1] factory-recipes =
            entry(prodA, sequence(execute-on(A-proc, facres), finished)) ~
           (entry(prodB, sequence(execute-on(B-proc, facres), finished)) ~
           (entry(prodC, sequence(execute-on(C-proc, facres), finished)) ~
           (entry(prodD, sequence(execute-on(D-proc, facres), finished)) ~
           empty-prod-table)))
    [C2] facres-recipes =
```

```
            entry(A-proc, sequence(execute-on(tr-2-op, transfres-2),
                          finished)) ~
           (entry(B-proc, sequence(execute-on(tr-3-op, transfres-3),
                          sequence(execute-on(tr-5-op, transfres-5),
                          finished))) ~
           (entry(C-proc, sequence(execute-on(tr-1-op, transfres-1),
                          sequence(execute-on(tr-1-op, transfres-1),
                          sequence(execute-on(tr-4-op, transfres-4),
                          finished)))) ~
           (entry(D-proc, sequence(execute-on(tr-5-op, transfres-5),
                          sequence(execute-on(tr-4-op, transfres-4),
                          sequence(execute-on(tr-3-op, transfres-3),
                          sequence(execute-on(tr-2-op, transfres-2),
                          sequence(execute-on(tr-1-op, transfres-1),
                          finished)))))) ~
           empty-rec-table)))
    [C3] tr-1-recipes =
           entry(tr-1-op, sequence(execute-on(transformation1, transf1),
                                   finished)) ~ empty-rec-table
    [C4] tr-2-recipes =
           entry(tr-2-op, sequence(execute-on(transformation2, transf2),
                                   finished)) ~ empty-rec-table
    [C5] tr-3-recipes =
           entry(tr-3-op, sequence(execute-on(transformation3, transf3),
                                   finished)) ~ empty-rec-table
    [C6] tr-4-recipes =
           entry(tr-4-op, sequence(execute-on(transformation4, transf4),
                                   finished)) ~ empty-rec-table
    [C7] tr-5-recipes =
           entry(tr-5-op, sequence(execute-on(transformation5, transf5),
                                   finished)) ~ empty-rec-table
end Cookbook


data module Control-functions
begin
  exports
    begin
      functions
        block-resource:   REQ-TABLE # RESOURCE                ->   REQ-TABLE
        block-resource:   REQ-TABLE # ORDER # PRODUCT-TABLE   ->   REQ-TABLE
        block-resource:   REQ-TABLE # JOB # RECIPE-TABLE      ->   REQ-TABLE
        block-resource:   REQ-TABLE # PF-TABLE # JOB          ->   REQ-TABLE
        blocked:          RESOURCE # REQ-TABLE                ->   BOOLEAN
        capable:          JOB # REQ-TABLE                     ->   BOOLEAN
        executable:       RECIPE # REQ-TABLE                  ->   BOOLEAN
        finished:         PROGRESSFORM                        ->   BOOLEAN
        first-job:        ORDER # PRODUCT-TABLE               ->   JOB
        first-job:        JOB # RECIPE-TABLE                  ->   JOB
        get-pf:           PF-TABLE # RESOURCE # CAPABILITY    ->   PROGRESSFORM
        get-updated-pf:   PF-TABLE # JOB                      ->   PROGRESSFORM
        make-pf:          ORDER # PRODUCT-TABLE               ->   PROGRESSFORM
        make-pf:          JOB # RECIPE-TABLE                  ->   PROGRESSFORM
        needs-transport:  PROGRESSFORM                        ->   BOOLEAN
        new-loc:          PF-TABLE # JOB                      ->   RESOURCE
        next-job:         PROGRESSFORM                        ->   JOB
        trans2store:      JOB                                 ->   BOOLEAN
        unblock-resource: REQ-TABLE # PF-TABLE # JOB          ->   REQ-TABLE
        update-place:     PF-TABLE # JOB                      ->   PF-TABLE
        update-rec:       PF-TABLE # ID                       ->   PF-TABLE
        wanted-by:        REQUEST # PF-TABLE                  ->   ID
    end
  imports
    ProgressForms, Requests, Product-tables, Recipe-tables,
    Progform-tables, Request-tables
functions
  capable:      OPERATION # CAPABILITY              ->   BOOLEAN
  executable:   OPERATION # RESOURCE # REQ-TABLE    ->   BOOLEAN
  update-place: PROGRESSFORM # JOB                  ->   PROGRESSFORM
  update-rec:   PROGRESSFORM                        ->   PROGRESSFORM
  wanted-by:    REQUEST # PROGRESSFORM              ->   BOOLEAN
variables
  oid, jid, jid1, jid2: ->   ID
  mu:                   ->   MATERIALUNIT
  op, op1, op2:         ->   OPERATION
  p, p1, p2:            ->   PRODUCT
  ps:                   ->   PROCESSSTEP
  rec:                  ->   RECIPE
  res, res1, res2:      ->   RESOURCE
  o:                    ->   ORDER
  j:                    ->   JOB
  pf:                   ->   PROGRESSFORM
  req, req1, req2:      ->   REQUEST
  cp:                   ->   CAPABILITY
  prod-tab:             ->   PRODUCT-TABLE
  rec-tab:              ->   RECIPE-TABLE
  pf-tab:               ->   PF-TABLE
  req-tab:              ->   REQ-TABLE
equations
  [Cf1]  block-resource(req-tab, res) =
           insert(req-tab, tformreq(res, thats-all-folks), res)
  [Cf2]  block-resource(req-tab, o, prod-tab) =
           insert(req-tab,
                  tformreq(resource(first-job(o, prod-tab)), thats-all-folks),
                  resource(first-job(o, prod-tab)))
  [Cf3]  block-resource(req-tab, j, rec-tab) =
           insert(req-tab,
                  tformreq(resource(first-job(j, rec-tab)), thats-all-folks),
                  resource(first-job(j, rec-tab)))
  [Cf4]  block-resource(req-tab, pf-tab, j) =
           insert(req-tab, tformreq(new-loc(pf-tab, j), thats-all-folks),
                  new-loc(pf-tab, j))

  [Cf5]  blocked(res, req-tab) =
           eq(cap(retrieve(req-tab, res)), thats-all-folks)
           when in-table(req-tab, res) = true
  [Cf6]  blocked(res, req-tab) = false
           when in-table(req-tab, res) = false

  [Cf7]  capable(tformjob(jid, op, res, mu), req-tab) =
           capable(op, cap(retrieve(req-tab, res)))
           when in-table(req-tab, res) = true
  [Cf8]  capable(tformjob(jid, op, res, mu), req-tab) = false
           when in-table(req-tab, res) = false
  [Cf9]  capable(op, thats-all-folks) = false
  [Cf10] capable(op, and-also(op, cp)) = true
  [Cf11] capable(op1, and-also(op2, cp)) = capable(op1, cp)
           when eq(op1, op2) = false

  [Cf12] executable(finished, req-tab) = true
  [Cf13] executable(sequence(execute-on(op, res), rec), req-tab) =
           and(executable(op, res, req-tab), executable(rec, req-tab))
  [Cf14] executable(op, res, req-tab) =
           capable(op, cap(retrieve(req-tab, res)))
           when in-table(req-tab, res) = true
  [Cf15] executable(op, res, req-tab) = false
           when in-table(req-tab, res) = false

  [Cf16] finished(ordprogform(o, res, finished)) = true
```

```
[Cf17] finished(ordprogform(o, res, sequence(ps, rec))) = true
[Cf18] finished(jobprogform(j, res, finished)) = true
[Cf19] finished(jobprogform(j, res, sequence(ps, rec))) = true

[Cf20] first-job(o, prod-tab) = next-job(make-pf(o, prod-tab))
[Cf21] first-job(j, rec-tab) = next-job(make-pf(j, rec-tab))

[Cf22] get-pf(pf-tab, res, cp) =
         retrieve(pf-tab, wanted-by(tformreq(res, cp), pf-tab))
[Cf23] get-updated-pf(pf-tab, j) =
         retrieve(update-rec(pf-tab, id(j)), id(j))

[Cf24] make-pf(order(oid, p), entry(p, rec)~prod-tab) =
         ordprogform(order(oid, p), store, rec)
[Cf25] make-pf(order(oid, p1), entry(p2, rec)~prod-tab) =
         make-pf(order(oid, p1), prod-tab)
         when eq(p1, p2) = false
[Cf26] make-pf(tformjob(jid, op, res, mu), entry(op, rec)~rec-tab) =
         jobprogform(tformjob(jid, op, res, mu), store, rec)
[Cf27] make-pf(tformjob(jid, op1, res, mu), entry(op2, rec)~rec-tab) =
         make-pf(tformjob(jid, op1, res, mu), rec-tab)
         when eq(op1, op2) = false

[Cf28] needs-transport(ordprogform(o, res, sequence(execute-on(op, res),
                                                    rec))) = false
[Cf29] needs-transport(ordprogform(o, res1, sequence(execute-on(op, res2),
                                                    rec))) = true
         when eq(res1, res2) = false
[Cf30] needs-transport(jobprogform(j, res, sequence(execute-on(op, res),
                                                    rec))) = false
[Cf31] needs-transport(jobprogform(j, res1, sequence(execute-on(op, res2),
                                                    rec))) = true
         when eq(res1, res2) = false

[Cf32] new-loc(pf-tab, j) = resource(next-job(get-updated-pf(pf-tab, j)))

[Cf33] next-job(ordprogform(o, res1, sequence(execute-on(op, res2), rec))) =
         tformjob(id(o), op, res2, mat(o))
[Cf34] next-job(jobprogform(j, res1, sequence(execute-on(op, res2), rec))) =
         tformjob(id(j), op, res2, mat(j))

[Cf35] trans2store(tportjob(jid, res, store, mu)) = true
[Cf36] trans2store(tportjob(jid, res1, res2, mu)) = false
         when eq(res2, store) = false

[Cf37] unblock-resource(req-tab, pf-tab, j) =
         delete(req-tab, resource(next-job(retrieve(pf-tab, id(j)))))

[Cf38] update-place(empty-pf-table, tportjob(jid, res1, res2, mu)) =
         empty-pf-table
[Cf39] update-place(entry(jid, pf)~pf-tab,
                    tportjob(jid, res1, res2, mu)) =
         entry(jid, update-place(pf, tportjob(jid, res1, res2, mu)))~pf-tab
[Cf40] update-place(entry(jid1, pf)~pf-tab,
                    tportjob(jid2, res1, res2, mu)) =
         entry(jid1, pf)~update-place(pf-tab,
                                      tportjob(jid2, res1, res2, mu))
         when eq(jid1, jid2) = false
[Cf41] update-place(ordprogform(o, res, rec),
                    tportjob(jid, res1, res2, mu)) =
         ordprogform(o, res2, rec)
[Cf42] update-place(jobprogform(j, res, rec),
                    tportjob(jid, res1, res2, mu)) =
         jobprogform(j, res2, rec)
```

```
    [Cf43] update-rec(empty-pf-table, jid) = empty-pf-table
    [Cf44] update-rec(entry(jid, pf)~pf-tab, jid) =
             entry(jid, update-rec(pf))~pf-tab
    [Cf45] update-rec(entry(jid1, pf)~pf-tab, jid2) =
             entry(jid1, pf)~update-rec(pf-tab, jid2)
             when eq(jid1, jid2) = false
    [Cf46] update-rec(ordprogform(o, res, sequence(ps, rec))) =
             ordprogform(o, res, rec)
    [Cf47] update-rec(jobprogform(j, res, sequence(ps, rec))) =
             jobprogform(j, res, rec)

    [Cf48] wanted-by(tformreq(res, cp), empty-pf-table) = default-id
    [Cf49] wanted-by(tformreq(res, cp), entry(jid, pf)~pf-tab) = jid
             when wanted-by(tformreq(res, cp), pf) = true
    [Cf50] wanted-by(tformreq(res, cp), entry(jid, pf)~pf-tab) =
             wanted-by(tformreq(res, cp), pf-tab)
             when wanted-by(tformreq(res, cp), pf) = false
    [Cf51] wanted-by(tformreq(res1, cp), ordprogform(o, res2, finished)) = false
    [Cf52] wanted-by(tformreq(res1, cp),
                     ordprogform(o, res2,
                                 sequence(execute-on(op, res1), rec))) =
             capable(op, cp)
    [Cf53] wanted-by(tformreq(res1, cp),
                     ordprogform(o, res,
                                 sequence(execute-on(op, res2), rec))) =
             false
             when eq(res1, res2) = false
    [Cf54] wanted-by(tformreq(res1, cp), jobprogform(j, res2, finished)) = false
    [Cf55] wanted-by(tformreq(res1, cp),
                     jobprogform(j, res2,
                                 sequence(execute-on(op, res1), rec))) =
             capable(op, cp)
    [Cf56] wanted-by(tformreq(res1, cp),
                     jobprogform(j, res,
                                 sequence(execute-on(op, res2), rec))) =
             false
             when eq(res1, res2) = false

-- "senseless" function applications
    [Cf57] capable(tportjob(jid, res1, res2, mu), req-tab) = false
    [Cf58] make-pf(o, empty-prod-table) = progform-nil
    [Cf59] make-pf(j, empty-rec-table) = progform-nil
    [Cf60] make-pf(tportjob(jid, res1, res2, mu), rec-tab) = progform-nil
    [Cf61] needs-transport(ordprogform(o, res, finished)) =
             not(eq(res, store))
    [Cf62] needs-transport(jobprogform(j, res, finished)) =
             not(eq(res, store))
    [Cf63] next-job(ordprogform(o, res, finished)) = job-nil
    [Cf64] next-job(jobprogform(j, res, finished)) = job-nil
    [Cf65] trans2store(tformjob(jid, op, res, mu)) = false
    [Cf66] update-place(pf-tab, tformjob(jid, op, res, mu)) = pf-tab
    [Cf67] update-place(pf, tformjob(jid, op, res, mu)) = progform-nil
    [Cf68] update-rec(ordprogform(o, res, finished)) = progform-nil
    [Cf69] update-rec(jobprogform(j, res, finished)) = progform-nil
    [Cf70] wanted-by(tportreq, pf-tab) = default-id
    [Cf71] wanted-by(tportreq, pf) = false
end Control-functions


process module Communication
begin
  exports
    begin
      atoms
```

```
        read: PORT # ORDER
        read: PORT # JOB
        read: PORT # REQUEST
        read: PORT # REPORT
        read: PORT # MATERIALREQUEST
        read: PORT # MATERIAL
        send: PORT # ORDER
        send: PORT # JOB
        send: PORT # REQUEST
        send: PORT # REPORT
        send: PORT # MATERIALREQUEST
        send: PORT # MATERIAL
        comm: PORT # ORDER
        comm: PORT # JOB
        comm: PORT # REQUEST
        comm: PORT # REPORT
        comm: PORT # MATERIALREQUEST
        comm: PORT # MATERIAL
    end
  imports
    Ports, Requests, Reports
  communications
    read(p, o) | send(p, o) = comm(p, o)
      for p in PORT, o in ORDER
    read(p, j) | send(p, j) = comm(p, j)
      for p in PORT, j in JOB
    read(p, req) | send(p, req) = comm(p, req)
      for p in PORT, req in REQUEST
    read(p, rep) | send(p, rep) = comm(p, rep)
      for p in PORT, rep in REPORT
    read(p, mr) | send(p, mr) = comm(p, mr)
      for p in PORT, mr in MATERIALREQUEST
    read(p, m) | send(p, m) = comm(p, m)
      for p in PORT, m in MATERIAL
end Communication


process module Stores
begin
  exports
    begin
      processes
        STORE: RESOURCE
    end
  imports
    Communication, Material-tables
  processes
    STORE: RESOURCE # MAT-TABLE
  variables
    self:    ->   RESOURCE
    mat-tab: ->   MAT-TABLE
  definitions
    STORE(self) = STORE(self, empty-mat-table)
    STORE(self, mat-tab) = sum(mu in MATERIALUNIT,
        read(port(self, intern), matreq(mu)) .
        send(port(self, intern), phys(mu)) . STORE(self, delete(mat-tab, mu))
      + read(port(self, extern), matreq(mu)) .
        send(port(self, extern), phys(mu)) . STORE(self, delete(mat-tab, mu))
      + read(port(self, intern), phys(mu)) .
        STORE(self, insert(mat-tab, mu))
      + read(port(self, extern), phys(mu)) .
        STORE(self, insert(mat-tab, mu)))
end Stores
```

```
process module FactoryStore
begin
  exports
    begin
      processes
        FACTSTORE
    end
  imports
    Communication, Material-tables
  processes
    FACTSTORE: MAT-TABLE
  variables
    mat-tab: -> MAT-TABLE
  definitions
    FACTSTORE = FACTSTORE(empty-mat-table)
    FACTSTORE(mat-tab) = sum(mu in MATERIALUNIT,
        read(port(fac-store, intern), matreq(mu)) .
        send(port(fac-store, intern), phys(mu)) . FACTSTORE(mat-tab)
      + read(port(fac-store, extern), matreq(mu)) .
        send(port(fac-store, extern), phys(mu)) . FACTSTORE(delete(mat-tab, mu))
      + read(port(fac-store, intern), phys(mu)) .
        FACTSTORE(insert(mat-tab, mu))
      + read(port(fac-store, extern), phys(mu)) .
        FACTSTORE(insert(mat-tab, mu)))
end FactoryStore


process module Transporters
begin
  exports
    begin
      atoms
        move-to: RESOURCE # RESOURCE
      processes
        TRANSPORTER: RESOURCE
    end
  imports
    Communication
  variables
    self: ->   RESOURCE
  definitions
    TRANSPORTER(self) = send(port(self, contr), tportreq) .
      sum(j in JOB, read(port(self, contr), j) . move-to(self, orig(j)) .
      send(port(self, orig(j)), matreq(mat(j))) .
      read(port(self, orig(j)), phys(mat(j))) . move-to(self, dest(j)) .
      send(port(self, dest(j)), phys(mat(j))) .
      send(port(self, contr), report(j))) . TRANSPORTER(self)
end Transporters


process module Transformers
begin
  parameters
    Name
      begin
        functions
          self:   ->   RESOURCE
      end Name,
    Capability
      begin
        functions
```

```
          operation:   ->   OPERATION
      end Capability
  exports
    begin
      atoms
        execute: OPERATION # MATERIAL
      processes
        TRANSFORMER: RESOURCE
    end
  imports
    Communication
  definitions
    TRANSFORMER(self) =
      send(port(self, contr),
           tformreq(self, and-also(operation, thats-all-folks))) .
      sum(mu in MATERIALUNIT, read(port(self, trans), phys(mu)) .
      sum(j in JOB, read(port(self, contr), j) .
      execute(operation, phys(mu)) . send(port(self, contr), report(j))) .
      read(port(self, trans), matreq(mu)) .
      send(port(self, trans), phys(mu))) . TRANSFORMER(self)
end Transformers


process module Transformer-1
begin
  imports
    Transformers{
      Name bound by
        [self -> transf1]
        to Resources
      Capability bound by
        [operation -> transformation1]
        to Operations
    }
end Transformer-1


process module Transformer-2
begin
  imports
    Transformers{
      Name bound by
        [self -> transf2]
        to Resources
      Capability bound by
        [operation -> transformation2]
        to Operations
    }
end Transformer-2


process module Transformer-3
begin
  imports
    Transformers{
      Name bound by
        [self -> transf3]
        to Resources
      Capability bound by
        [operation -> transformation3]
        to Operations
    }
end Transformer-3


process module Transformer-4
begin
  imports
    Transformers{
      Name bound by
        [self -> transf4]
        to Resources
      Capability bound by
        [operation -> transformation4]
        to Operations
    }
end Transformer-4


process module Transformer-5
begin
  imports
    Transformers{
      Name bound by
        [self -> transf5]
        to Resources
      Capability bound by
        [operation -> transformation5]
        to Operations
    }
end Transformer-5


process module ManufacturingControllers
begin
  parameters
    Name
      begin
        functions
          self: -> RESOURCE
      end Name,
    ManfacResources
      begin
        sets
          of RESOURCE
            RES-SET
      end ManfacResources,
    ManfacRecipes
      begin
        functions
          recipes: -> RECIPE-TABLE
      end ManfacRecipes
  exports
    begin
      processes
        MANFACCONTR: RESOURCE
    end
  imports
    Control-functions, Communication, Job-queues
  processes
    MANFACCONTR:  RESOURCE # JOB-QUEUE # REQ-TABLE # PF-TABLE
    SEND-REQUEST: RESOURCE # JOB-QUEUE # REQ-TABLE # PF-TABLE # RECIPE-TABLE
    SR:           RESOURCE # REQ-TABLE # RECIPE-TABLE # CAPABILITY
  variables
```

```
    op:      ->   OPERATION
    cp:      ->   CAPABILITY
    rec:     ->   RECIPE
    job-q:   ->   JOB-QUEUE
    rec-tab: ->   RECIPE-TABLE
    pf-tab:  ->   PF-TABLE
    req-tab: ->   REQ-TABLE
  definitions
    MANFACCONTR(self) =
      MANFACCONTR(self, empty-queue, empty-req-table, empty-pf-table)
    MANFACCONTR(self, job-q, req-tab, pf-tab) =

-- receive a transformation job from the supercontroller
        sum(j in JOB, read(port(self, contr), j) .
        ( [capable(first-job(j, recipes), req-tab) = true] ->
-- first subjob can be executed
            ( [in-table(req-tab, trans) = true] ->
-- transporter is available; send the proper transport job
                send(port(self, trans),
                     tportjob(id(j), store, resource(first-job(j, recipes)),
                              mat(j))) .
-- block resource; remove transport request; add new progressform; send request
                SEND-REQUEST(self, job-q,
                             delete(block-resource(req-tab, j, recipes), trans),
                             insert(pf-tab, make-pf(j, recipes), id(j)),
                             recipes)
            + [in-table(req-tab, trans) = false] ->
-- transporter is not available; enqueue transport job; block resource;
-- add new progressform; send request
                SEND-REQUEST(self,
                             enqueue(tportjob(id(j), store,
                                              resource(first-job(j, recipes)),
                                              mat(j)),
                                     job-q),
                             block-resource(req-tab, j, recipes),
                             insert(pf-tab, make-pf(j, recipes), id(j)),
                             recipes)
            )
        + [capable(first-job(j, recipes), req-tab) = false] ->
-- first subjob an not be executed; add new progressform; send request
            SEND-REQUEST(self, job-q, req-tab,
                         insert(pf-tab, make-pf(j, recipes), id(j)),
                         recipes)
        ))

-- receive a request from a resource
      + sum(res0 in RES-SET, sum(cp in CAPABILITY,
        read(port(self, res0), tformreq(res0, cp)) .
        ( [blocked(res0, req-tab) = true] ->
-- resource is blocked; nothing happens
            MANFACCONTR(self, job-q, req-tab, pf-tab)
        + [blocked(res0, req-tab) = false] ->
-- resource is not blocked
            ( [eq(wanted-by(tformreq(res0, cp), pf-tab), default-id) = true] ->
-- resource is not needed; insert request in request-table; send own request
                SEND-REQUEST(self, job-q,
                             insert(req-tab, tformreq(res0, cp), res0),
                             pf-tab, recipes)
            + [eq(wanted-by(tformreq(res0, cp), pf-tab), default-id) = false] ->
-- resource is needed
                ( [needs-transport(get-pf(pf-tab, res0, cp)) = true] ->
-- material must be transported to the resource
                    ( [in-table(req-tab, trans) = true] ->
-- transporter is available; send the proper transport job
                        send(port(self, trans),
```

```
                             tportjob(wanted-by(tformreq(res0, cp), pf-tab),
                                      loc(get-pf(pf-tab, res0, cp)), res0,
                                      mat(next-job(get-pf(pf-tab, res0,
                                                          cp))))) .
-- remove transport request; block resource
                        MANFACCONTR(self, job-q,
                                    delete(block-resource(req-tab, res0),
                                           trans),
                                    pf-tab)
                    + [in-table(req-tab, trans) = false] ->
-- transporter is not available; enqueue transport job; block resource
                        MANFACCONTR(self,
                                    enqueue(tportjob(
                                       wanted-by(tformreq(res0, cp), pf-tab),
                                       loc(get-pf(pf-tab, res0, cp)),
                                       res0,
                                       mat(next-job(get-pf(pf-tab, res0, cp)))),
                                            job-q),
                                    block-resource(req-tab, res0),
                                    pf-tab)
                    )
                + [needs-transport(get-pf(pf-tab, res0, cp)) = false] ->
-- material does not need to be transported to the resource; send subjob
                    send(port(self, res0), next-job(get-pf(pf-tab, res0, cp))) .
-- remove old request from resource, if any
                    MANFACCONTR(self, job-q, delete(req-tab, res0), pf-tab)
                )
            )
        )))

-- receive a request from the transporter
      + read(port(self, trans), tportreq) .
        ( [eq(length(job-q), nat(^0)) = true] ->
-- transporter is not needed; insert request in request-table
            MANFACCONTR(self, job-q, insert(req-tab, tportreq, trans), pf-tab)
        + [eq(length(job-q), nat(^0)) = false] ->
-- transporter is needed; send the first transport job from the queue
            send(port(self, trans), serve(job-q)) .
-- remove transport job from the queue
            MANFACCONTR(self, dequeue(job-q), req-tab, pf-tab)
        )

-- receive a report from a resource
      + sum(res0 in RES-SET, sum(j in JOB, read(port(self, res0), report(j)) .
        ( [finished(get-updated-pf(pf-tab, j)) = true] ->
-- superjob is finished
            ( [in-table(req-tab, trans) = true] ->
-- transporter is available; send transport job to bring material to store
                send(port(self, trans), tportjob(id(j), res0, store, mat(j))) .
-- remove transport request; remove subjob from recipe
                MANFACCONTR(self, job-q, delete(req-tab , trans),
                            update-rec(pf-tab, id(j)))
            + [in-table(req-tab, trans) = false] ->
-- transporter is not available; enqueue transport job;
-- remove subjob from recipe
                MANFACCONTR(self,
                            enqueue(tportjob(id(j), res0, store, mat(j)),
                                    job-q),
                            req-tab,
                            update-rec(pf-tab, id(j)))
            )
        + [finished(get-updated-pf(pf-tab, j)) = false] ->
-- superjob is not yet finished
            ( [capable(next-job(get-updated-pf(pf-tab, j)), req-tab) = true] ->
-- next subjob can be executed
```

```
                ( [needs-transport(get-updated-pf(pf-tab, j)) = true] ->
-- material must be transported to the next resource
                    ( [in-table(req-tab, trans) = true] ->
-- transporter is available; send the poper transport job
                        send(port(self, trans),
                             tportjob(id(j), res0, new-loc(pf-tab, j),
                                      mat(j))) .
-- remove transport request; block resource; remove subjob from recipe
                        MANFACCONTR(self, job-q,
                                    delete(block-resource(req-tab, pf-tab, j),
                                           trans),
                                    update-rec(pf-tab, id(j)))
                    + [in-table(req-tab, trans) = false] ->
-- transporter is not available; enqueue transport job; block resource;
-- remove subjob from recipe
                        MANFACCONTR(self,
                                    enqueue(tportjob(id(j), res0,
                                                     new-loc(pf-tab, j),
                                                     mat(j)),
                                            job-q),
                                    block-resource(req-tab, pf-tab, j),
                                    update-rec(pf-tab, id(j)))
                    )
                + [needs-transport(get-updated-pf(pf-tab, j)) = false] ->
-- material does not need to be transported to the next resource;
-- send next subjob
                    send(port(self, res0),
                         next-job(get-updated-pf(pf-tab, j))) .
-- remove resource’s request; remove subjob from recipe
                    MANFACCONTR(self, job-q,
                                delete(req-tab, new-loc(pf-tab, j)),
                                update-rec(pf-tab, id(j)))
                )
            + [capable(next-job(get-updated-pf(pf-tab, j)), req-tab) = false] ->
-- next subjob can not be executed; remove subjob from recipe
                MANFACCONTR(self, job-q, req-tab, update-rec(pf-tab, id(j)))
            )
        )))

-- receive a report from the transporter
      + sum(j in JOB, read(port(self, trans), report(j)) .
        ( [trans2store(j) = true]  ->
-- destination of transport job was ’store’; superjob is finished;
-- send report to supercontroller
            send(port(self, contr), report(job(retrieve(pf-tab, id(j))))) .
-- remove progressform
            MANFACCONTR(self, job-q, req-tab, delete(pf-tab, id(j)))
        + [trans2store(j) = false] ->
-- destination of transport job was not ’store’; superjob is not yet finished;
-- material has been transported to a resource; send next subjob to resource
            send(port(self, resource(next-job(retrieve(pf-tab, id(j))))),
                 next-job(retrieve(pf-tab, id(j)))) .
-- adjust location of material in progressform; unblock the resource
            MANFACCONTR(self, job-q,
                        unblock-resource(req-tab, pf-tab, j),
                        update-place(pf-tab, j))
        ))

    SEND-REQUEST(self, job-q, req-tab, pf-tab, rec-tab) =
      SR(self, req-tab, rec-tab, thats-all-folks) .
      MANFACCONTR(self, job-q, req-tab, pf-tab)

-- none of the operations can be executed now
    SR(self, req-tab, empty-rec-table, thats-all-folks) = skip
-- there are operations that can be executed; send the proper request
```

```
    SR(self, req-tab, empty-rec-table, and-also(op, cp)) =
      send(port(self, contr),
           tformreq(controlled-resource(self), and-also(op, cp)))
    SR(self, req-tab, entry(op, rec)~rec-tab, cp) =
      ( [executable(rec, req-tab) = true] ->
-- subjob can be executed
          SR(self, req-tab, rec-tab, and-also(op, cp))
      + [executable(rec, req-tab) = false] ->
-- subjob can not be executed
          SR(self, req-tab, rec-tab, cp)
      )
end ManufacturingControllers


process module FactoryResourceController
begin
  imports
    ManufacturingControllers{
      Name bound by
        [self -> facres-contr]
        to Resources
      ManfacResources bound by
        [ RES-SET -> factres-resources]
        to ResourceSets
      ManfacRecipes bound by
        [recipes -> facres-recipes]
        to Cookbook
    }
end FactoryResourceController


process module Transformer-1-Controller
begin
  imports
    ManufacturingControllers{
      Name bound by
        [self -> tr-1-contr]
        to Resources
      ManfacResources bound by
        [RES-SET -> tr-1-resources]
        to ResourceSets
      ManfacRecipes bound by
        [recipes -> tr-1-recipes]
        to Cookbook
    }
end Transformer-1-Controller


process module Transformer-2-Controller
begin
  imports
    ManufacturingControllers{
      Name bound by
        [self -> tr-2-contr]
        to Resources
      ManfacResources bound by
        [RES-SET -> tr-2-resources]
        to ResourceSets
      ManfacRecipes bound by
        [recipes -> tr-2-recipes]
        to Cookbook
    }
```

```
end Transformer-2-Controller


process module Transformer-3-Controller
begin
  imports
    ManufacturingControllers{
      Name bound by
        [self -> tr-3-contr]
        to Resources
      ManfacResources bound by
        [RES-SET -> tr-3-resources]
        to ResourceSets
      ManfacRecipes bound by
        [recipes -> tr-3-recipes]
        to Cookbook
    }
end Transformer-3-Controller


process module Transformer-4-Controller
begin
  imports
    ManufacturingControllers{
      Name bound by
        [self -> tr-4-contr]
        to Resources
      ManfacResources bound by
        [RES-SET -> tr-4-resources]
        to ResourceSets
      ManfacRecipes bound by
        [recipes -> tr-4-recipes]
        to Cookbook
    }
end Transformer-4-Controller


process module Transformer-5-Controller
begin
  imports
    ManufacturingControllers{
      Name bound by
        [self -> tr-5-contr]
        to Resources
      ManfacResources bound by
        [RES-SET -> tr-5-resources]
        to ResourceSets
      ManfacRecipes bound by
        [recipes -> tr-5-recipes]
        to Cookbook
    }
end Transformer-5-Controller


process module FactoryControllers
begin
  parameters
    Name
      begin
        functions
          self: -> RESOURCE
      end Name,
    FacResource
      begin
        functions
          res: -> RESOURCE
      end FacResource,
    FactRecipes
      begin
        functions
          products: -> PRODUCT-TABLE
      end FactRecipes
  exports
    begin
      processes
        FACTORYCONTR: RESOURCE
    end
  imports
    Control-functions, Communication, Job-queues
  processes
    FACTORYCONTR: RESOURCE # JOB-QUEUE # REQ-TABLE # PF-TABLE
  variables
    job-q:   ->   JOB-QUEUE
    req-tab: ->   REQ-TABLE
    pf-tab:  ->   PF-TABLE
  definitions
    FACTORYCONTR(self) =
      FACTORYCONTR(self, empty-queue, empty-req-table, empty-pf-table)
    FACTORYCONTR(self, job-q, req-tab, pf-tab) =

        sum(o in ORDER, read(port(self, outside), o) .
        ( [capable(first-job(o, products), req-tab) = true] ->
            ( [in-table(req-tab, trans) = true] ->
                send(port(self, trans), tportjob(id(o), store, res, mat(o))) .
                FACTORYCONTR(self, job-q,
                             delete(block-resource(req-tab, o, products),
                                    trans),
                             insert(pf-tab, make-pf(o, products), id(o)))
            + [in-table(req-tab, trans) = false] ->
                FACTORYCONTR(self,
                             enqueue(tportjob(id(o), store, res, mat(o)),
                                     job-q),
                             block-resource(req-tab, o, products),
                             insert(pf-tab, make-pf(o, products), id(o)))
            )
        + [capable(first-job(o, products), req-tab) = false] ->
            FACTORYCONTR(self, job-q, req-tab,
                         insert(pf-tab, make-pf(o, products), id(o)))
        ))

      + sum(cp in CAPABILITY, read(port(self, res), tformreq(res, cp)) .
        ( [blocked(res, req-tab) = true] ->
            FACTORYCONTR(self, job-q, req-tab, pf-tab)
        + [blocked(res, req-tab) = false] ->
            ( [eq(wanted-by(tformreq(res, cp), pf-tab), default-id) = true] ->
                FACTORYCONTR(self, job-q,
                             insert(req-tab, tformreq(res, cp), res),
                             pf-tab)
            + [eq(wanted-by(tformreq(res, cp), pf-tab), default-id) = false] ->
                ( [needs-transport(get-pf(pf-tab, res, cp)) = true] ->
                    ( [in-table(req-tab, trans) = true] ->
                        send(port(self, trans),
                             tportjob(wanted-by(tformreq(res, cp), pf-tab),
                                      store, res,
                                      mat(next-job(get-pf(pf-tab, res, cp))))) .
                        FACTORYCONTR(self, job-q,
```

```
                                     delete(block-resource(req-tab, res),
                                            trans),
                                     pf-tab)
                  + [in-table(req-tab, trans) = false] ->
                      FACTORYCONTR(self,
                                   enqueue(tportjob(
                                      wanted-by(tformreq(res, cp), pf-tab),
                                      store, res,
                                      mat(next-job(get-pf(pf-tab, res, cp)))),
                                           job-q),
                                   block-resource(req-tab, res),
                                   pf-tab)
                  )
              + [needs-transport(get-pf(pf-tab, res, cp)) = false] ->
                  send(port(self, res), next-job(get-pf(pf-tab, res, cp))) .
                  FACTORYCONTR(self, job-q, delete(req-tab, res), pf-tab)
              )
          )
      ))

    + read(port(self, trans), tportreq) .
      ( [eq(length(job-q), nat(^0)) = true] ->
          FACTORYCONTR(self, job-q, insert(req-tab, tportreq, trans), pf-tab)
      + [eq(length(job-q), nat(^0)) = false] ->
          send(port(self, trans), serve(job-q)) .
          FACTORYCONTR(self, dequeue(job-q), req-tab, pf-tab)
      )

    + sum(j in JOB, read(port(self, res), report(j)) .
      ( [finished(get-updated-pf(pf-tab, j)) = true] ->
          ( [in-table(req-tab, trans) = true] ->
              send(port(self, trans), tportjob(id(j), res, store, mat(j))) .
              FACTORYCONTR(self, job-q,
                           delete(req-tab , trans),
                           update-rec(pf-tab, id(j)))
          + [in-table(req-tab, trans) = false] ->
              FACTORYCONTR(self,
                           enqueue(tportjob(id(j), res, store, mat(j)),
                                   job-q),
                           req-tab,
                           update-rec(pf-tab, id(j)))
          )
      + [finished(get-updated-pf(pf-tab, j)) = false] ->
          ( [capable(next-job(get-updated-pf(pf-tab, j)), req-tab) = true] ->
              send(port(self, res), next-job(get-updated-pf(pf-tab, j))) .
              FACTORYCONTR(self, job-q, delete(req-tab, res),
                           update-rec(pf-tab, id(j)))
          + [capable(next-job(get-updated-pf(pf-tab, j)), req-tab) = false] ->
              FACTORYCONTR(self, job-q, req-tab, update-rec(pf-tab, id(j)))
          )
      ))

    + sum(j in JOB, read(port(self, trans), report(j)) .
      ( [trans2store(j) = true]  ->
          send(port(self, outside), report(order(retrieve(pf-tab, id(j))))) .
          FACTORYCONTR(self, job-q, req-tab, delete(pf-tab, id(j)))
      + [trans2store(j) = false] ->
          send(port(self, res), next-job(retrieve(pf-tab, id(j)))) .
          FACTORYCONTR(self, job-q, unblock-resource(req-tab, pf-tab, j),
                       update-place(pf-tab, j))
      ))
end FactoryControllers
```

```
process module FactControl
begin
  imports
    FactoryControllers{
      Name bound by
        [self -> fac-contr]
        to Resources
      FacResource bound by
        [res -> facres]
        to Resources
      FactRecipes bound by
        [products -> factory-recipes]
        to Cookbook
    }
end FactControl


process module ResourceSets
begin
  exports
    begin
      sets
        of RESOURCE
          factres-resources =
            { transfres-1, transfres-2, transfres-3, transfres-4, transfres-5 }
          tr-1-resources = { transf1 }
          tr-2-resources = { transf2 }
          tr-3-resources = { transf3 }
          tr-4-resources = { transf4 }
          tr-5-resources = { transf5 }
    end
  imports
    Resources
end ResourceSets


process module TransformationResource-1
begin
  exports
    begin
      processes
        TRANSFRES-1
    end
  imports
    Transformer-1-Controller, Stores, Transporters, Transformer-1
  processes
    TRANSF-1-CONTR
    TRANSF-1-STORE
    TRANSF-1-TRANSP
    TRANSF-1
  sets
    of PORT
      P = { port(tr-1-transp, contr), port(transf1, contr),
            port(tr-1-transp, store), port(transf1, trans) }
    of atoms
      H = { read(p, j), send(p, j), read(p, req), send(p, req),
            read(p, rep), send(p, rep), read(p, mr), send(p, mr),
            read(p, m), send(p, m) | p in P, j in JOB, req in REQUEST,
            rep in REPORT, mr in MATERIALREQUEST, m in MATERIAL }
  definitions
    TRANSFRES-1 = encaps(H, TRANSF-1-CONTR || TRANSF-1-STORE ||
                            TRANSF-1-TRANSP || TRANSF-1)
```

```
    TRANSF-1-CONTR = MANFACCONTR(tr-1-contr)
    TRANSF-1-STORE = STORE(tr-1-store)
    TRANSF-1-TRANSP = TRANSPORTER(tr-1-transp)
    TRANSF-1 = TRANSFORMER(transf1)
end TransformationResource-1


process module TransformationResource-2
begin
  exports
    begin
      processes
        TRANSFRES-2
    end
  imports
    Transformer-2-Controller, Stores, Transporters, Transformer-2
  processes
    TRANSF-2-CONTR
    TRANSF-2-STORE
    TRANSF-2-TRANSP
    TRANSF-2
  sets
    of PORT
      P = { port(tr-2-transp, contr), port(transf2, contr),
            port(tr-2-transp, store), port(transf2, trans) }
    of atoms
      H = { read(p, j), send(p, j), read(p, req), send(p, req),
            read(p, rep), send(p, rep), read(p, mr), send(p, mr),
            read(p, m), send(p, m) | p in P, j in JOB, req in REQUEST,
            rep in REPORT, mr in MATERIALREQUEST, m in MATERIAL }
  definitions
    TRANSFRES-2 = encaps(H, TRANSF-2-CONTR || TRANSF-2-STORE ||
                        TRANSF-2-TRANSP || TRANSF-2)

    TRANSF-2-CONTR = MANFACCONTR(tr-2-contr)
    TRANSF-2-STORE = STORE(tr-2-store)
    TRANSF-2-TRANSP = TRANSPORTER(tr-2-transp)
    TRANSF-2 = TRANSFORMER(transf2)
end TransformationResource-2


process module TransformationResource-3
begin
  exports
    begin
      processes
        TRANSFRES-3
    end
  imports
    Transformer-3-Controller, Stores, Transporters, Transformer-3
  processes
    TRANSF-3-CONTR
    TRANSF-3-STORE
    TRANSF-3-TRANSP
    TRANSF-3
  sets
    of PORT
      P = { port(tr-3-transp, contr), port(transf3, contr),
            port(tr-3-transp, store), port(transf3, trans) }
    of atoms
      H = { read(p, j), send(p, j), read(p, req), send(p, req),
            read(p, rep), send(p, rep), read(p, mr), send(p, mr),
            read(p, m), send(p, m) | p in P, j in JOB, req in REQUEST,
            rep in REPORT, mr in MATERIALREQUEST, m in MATERIAL }
  definitions
    TRANSFRES-3 = encaps(H, TRANSF-3-CONTR || TRANSF-3-STORE ||
                        TRANSF-3-TRANSP || TRANSF-3)

    TRANSF-3-CONTR = MANFACCONTR(tr-3-contr)
    TRANSF-3-STORE = STORE(tr-3-store)
    TRANSF-3-TRANSP = TRANSPORTER(tr-3-transp)
    TRANSF-3 = TRANSFORMER(transf3)
end TransformationResource-3


process module TransformationResource-4
begin
  exports
    begin
      processes
        TRANSFRES-4
    end
  imports
    Transformer-4-Controller, Stores, Transporters, Transformer-4
  processes
    TRANSF-4-CONTR
    TRANSF-4-STORE
    TRANSF-4-TRANSP
    TRANSF-4
  sets
    of PORT
      P = { port(tr-4-transp, contr), port(transf4, contr),
            port(tr-4-transp, store), port(transf4, trans) }
    of atoms
      H = { read(p, j), send(p, j), read(p, req), send(p, req),
            read(p, rep), send(p, rep), read(p, mr), send(p, mr),
            read(p, m), send(p, m) | p in P, j in JOB, req in REQUEST,
            rep in REPORT, mr in MATERIALREQUEST, m in MATERIAL }
  definitions
    TRANSFRES-4 = encaps(H, TRANSF-4-CONTR || TRANSF-4-STORE ||
                        TRANSF-4-TRANSP || TRANSF-4)

    TRANSF-4-CONTR = MANFACCONTR(tr-4-contr)
    TRANSF-4-STORE = STORE(tr-4-store)
    TRANSF-4-TRANSP = TRANSPORTER(tr-4-transp)
    TRANSF-4 = TRANSFORMER(transf4)
end TransformationResource-4


process module TransformationResource-5
begin
  exports
    begin
      processes
        TRANSFRES-5
    end
  imports
    Transformer-5-Controller, Stores, Transporters, Transformer-5
  processes
    TRANSF-5-CONTR
    TRANSF-5-STORE
    TRANSF-5-TRANSP
    TRANSF-5
  sets
    of PORT
      P = { port(tr-5-transp, contr), port(transf5, contr),
```

```
            port(tr-5-transp, store), port(transf5, trans) }
    of atoms
      H = { read(p, j), send(p, j), read(p, req), send(p, req),
            read(p, rep), send(p, rep), read(p, mr), send(p, mr),
            read(p, m), send(p, m) | p in P, j in JOB, req in REQUEST,
            rep in REPORT, mr in MATERIALREQUEST, m in MATERIAL }
  definitions
    TRANSFRES-5 = encaps(H, TRANSF-5-CONTR || TRANSF-5-STORE ||
                        TRANSF-5-TRANSP || TRANSF-5)

    TRANSF-5-CONTR = MANFACCONTR(tr-5-contr)
    TRANSF-5-STORE = STORE(tr-5-store)
    TRANSF-5-TRANSP = TRANSPORTER(tr-5-transp)
    TRANSF-5 = TRANSFORMER(transf5)
end TransformationResource-5


process module FactoryResource
begin
  exports
    begin
      processes
        FACTRES
    end
  imports
    FactoryResourceController, Stores, Transporters,
    TransformationResource-1, TransformationResource-2,
    TransformationResource-3, TransformationResource-4,
    TransformationResource-5
  processes
    FACRESCONTR
    FACRESSTORE
    FACRESTRANSP
  sets
    of PORT
      P = { port(facres-transp, contr), port(tr-1-contr, contr),
            port(tr-2-contr, contr), port(tr-3-contr, contr),
            port(tr-4-contr, contr), port(tr-5-contr, contr),
            port(facres-transp, store), port(tr-1-store, extern),
            port(tr-2-store, extern), port(tr-3-store, extern),
            port(tr-4-contr, contr), port(tr-5-contr, contr) }
    of atoms
      H = { read(p, j), send(p, j), read(p, req), send(p, req),
            read(p, rep), send(p, rep), read(p, mr), send(p, mr),
            read(p, m), send(p, m) | p in P, j in JOB, req in REQUEST,
            rep in REPORT, mr in MATERIALREQUEST, m in MATERIAL }
  definitions
    FACTRES = encaps(H, FACRESCONTR || FACRESSTORE || FACRESTRANSP ||
                    TRANSFRES-1 || TRANSFRES-2 || TRANSFRES-3 ||
                    TRANSFRES-4 || TRANSFRES-5)

    FACRESCONTR = MANFACCONTR(facres-contr)
    FACRESSTORE = STORE(facres-store)
    FACRESTRANSP = TRANSPORTER(facres-transp)
end FactoryResource


process module Factory
begin
  exports
    begin
      processes
        FACTORY
    end
  imports
    FactControl, FactoryStore, Transporters, FactoryResource
  processes
    FACTCONTR
    FACTRANSP
  sets
    of PORT
      P = { port(fac-transp, contr), port(facres-contr, contr),
            port(fac-transp, store), port(facres-store, extern) }
    of atoms
      H = { read(p, j), send(p, j), read(p, req), send(p, req),
            read(p, rep), send(p, rep), read(p, mr), send(p, mr),
            read(p, m), send(p, m) | p in P, j in JOB, req in REQUEST,
            rep in REPORT, mr in MATERIALREQUEST, m in MATERIAL }
  definitions
    FACTORY = encaps(H, FACTCONTR || FACTSTORE || FACTRANSP || FACTRES)

    FACTCONTR = FACTORYCONTR(fac-contr)
    FACTRANSP = TRANSPORTER(fac-transp)
end Factory


process module Consumer
begin
  exports
    begin
      processes
        CONSUMER
    end
  imports
    Communication
  processes
    CONSUMER: ID
  variables
    oid: -> ID
  definitions
    CONSUMER = CONSUMER(id(nat(^0)))
    CONSUMER(oid) =
        sum(o in ORDER, read(port(cons, fac), report(o))) . CONSUMER(oid)
      + sum(p in PRODUCT, send(port(cons, fac), order(oid, p))) .
        CONSUMER(next(oid))
end Consumer


process module Market
begin
  exports
    begin
      processes
        MARKET
    end
  imports
    Consumer, Factory
  sets
    of atoms
      H = {
        read(port(fac-contr, outside), o),   send(port(fac-contr, outside), o),
        read(port(fac-contr, outside), rep), send(port(fac-contr, outside), rep)
        | o in ORDER, rep in REPORT }
  definitions
    MARKET = encaps(H, CONSUMER || FACTORY)
end Market
```