Specification of a Flexible Manufacturing System Using Concurrent Programming

Specification of a Flexible Manufacturing System Using Concurrent Programming

J.M. van de Mortel-Fronczak[*], J.E. Rooda[†],
and N.J.M. van den Nieuwelaar
Department of Mechanical Engineering,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven,
The Netherlands

Abstract

Because of the growing complexity, the design of and reasoning about modern industrial systems becomes increasingly difficult. In order to understand and estimate the dynamic system behaviour, appropriate models have to be used. In many cases, existing mathematical models like queuing networks, Markov chain models, or perturbation analysis cannot be applied. In such cases, usually a model is constructed that can be validated by means of computer simulation. Since industrial systems exhibit concurrency, formalisms developed to reason about concurrent systems are also well suited for developing models in this specific application area. Models of systems can be expressed, for instance, in terms of Petri nets or in terms of programs written in a concurrent programming language, like Timed CSP. Both approaches, originating from computer science, are increasingly often applied in modelling of manufacturing systems. In this paper, we present a simple modular approach to the specification of (discrete) industrial systems that is based on concurrent programming. As an illustration, we present a model of a flexible manufacturing cell. The specification language used is modular and, therefore, allows for hierarchical modelling. To specify the information and control flows, a VDM-like notation is used.

1 Introduction

Modern industrial systems become increasingly complex. Because of this complexity, such systems are difficult to design and it is also difficult to estimate their temporal behaviour. One way to tackle the design and analysis problem is to construct a model that can be validated by means of computer simulation. In the sequel, we restrict our attention to discrete industrial systems such as flow-shop, job-shop, and flexible manufacturing systems.

Industrial systems are typical examples of concurrent systems that already for a long time belong to important subjects of research in computer science. Many formalisms for modelling and analysis of concurrent systems have been developed and, more recently, some of them found many applications in the area of manufacturing systems. Among those formalisms, Petri nets are probably most often used (numerous applications are surveyed in [18]). Beside many virtues, they have one basic weakness ([5]): the lack of modularity. Petri net models usually depend on the product recipe: for different recipes different models must be constructed (see for instance [3] and [17]). The specification language ExSpect ([6]) built upon the Petri net theory and developed for distributed information systems, offers more flexibility in this respect ([22]).

The purpose of this paper is to illustrate the use of an alternative formalism: real-time concurrent programming. Concurrent programming can be used for specification, simulation, and control of discrete industrial systems. It is based on a view of a system as a collection of active, interacting with each other, components. In this paper, we concentrate on the specification (modelling) phase. According to our approach, every component of the system is modelled by a sequential process. Interaction among components is modelled by send and receive actions along fixed communication channels. An introduction to concurrent programming principles can be found in

[*]Email address: vdmortel@urc.tue.nl, fax: +40 44 86 65, phone: +40 47 2892

[†]Email address: rooda@urc.tue.nl

[4]. In modelling, this view of discrete-event systems is an important and convenient abstraction. It can be seen as a first step towards a standard simulation environment as discussed in [20]. Models resulting from this approach abstract from the underlying simulation algorithm and are therefore easier to grasp than models explicitely using techniques known as activity scanning, event scheduling, and process interaction. To specify data and data processing operations facilitating the modelling of information and control flows, a VDM-like notation ([21]) is used.

Using a formal framework, such as Petri nets or real-time concurrent programming, as a basis for modelling allows also for calculational derivation of certain static properties of models (for example, the presence of deadlocks).

In this paper, we show how a real-time concurrent programming language can be used for specification of discrete industrial systems. At Eindhoven University of Technology, Department of Mechanical Engineering, a limited version of the set of basic concepts of this language, implemented in Smalltalk-80, was for the last five years successfully applied to a number of complex problems in the area of manufacturing systems. A couple of representative examples are: the design of a tyre manufacturing factory ([1]) and the design of a control for an IC manufacturing system ([19]). The design and analysis of models were supported by a software tool ([23]). A comparable approach based on a different language is presented in [9]. The language is also essentially based on CSP ([7]) except for the fact that a system is viewed as a collection of sequential components that operate concurrently (we have no parallel composition among the program statements). Other examples of related specification languages (accompanied by supporting software tools such as editors and simulators) are Lotos ([10]) and PSF ([11]) where, however, no explicit notion of time is incorporated.

The paper is organized as follows. In Section 2, the specification language is introduced. A model of a flexible manufacturing cell is presented in Section 3. Section 4 contains some concluding remarks.

2 Specification language

In the modelling approach based on concurrent programming, a system is viewed as a collection of communicating processes. A process is specified by a program in a CSP-like programming language (adopted from [8]) preceded by declarations of local variables and statistical distributions. The language is similar to Occam ([15]). Processes do not share variables — they interact exclusively by using the communication and synchronization primitives (synchronous message-passing). Every process is sequential. Processes in a system can execute concurrently between synchronization points.

TC	$\Delta r \mid c!e \mid c?x \mid c! \mid c?$
G	$[GB] \mid [GC]$
GB	$b \longrightarrow S \mid GB \, [\,] \, GB$
GC	$b\,;TC \longrightarrow S \mid GC \, [\,] \, GC$
S	$\mathsf{skip} \mid x := e \mid TC \mid S\,;\,S \mid G \mid {*G} \mid nG$

Table 1: The syntax of the language

The language ([13]) consists of: assignment, selection (nondeterministic choice), repetition, sequential composition, communication and synchronization primitives, time passing and selective waiting ([4]). Two differences with respect to [8] are: we allow multiple timeouts (an example of this is shown in [13]) and besides inputs also outputs in guarded commands. Moreover, we introduce a special variable (τ) representing the current time (of a process), which may be used in expressions. The syntax of the language is given in Table 1. In the definition, we use n for a natural number ($n \geq 0$), r for a real expression, c for a channel name, b for a boolean expression, x for a variable name, and e for an expression (of the same type as x). Expressions r, b, and e may explicitly refer to τ. Sequences of capitals denote nonterminal symbols of the grammar.

The expressions b and $b\,;\gamma$ (γ belongs to the category TC) appearing before the arrows are called guards. We say that a guard is open if it evaluates to true or, in the case of guards with interactions, if its boolean part evaluates to true.

The statements have the following informal meaning:

- skip terminates without influencing the program state.

- $x := e$ denotes the assignment of the value of e to the variable x.

- Δr denotes time passing.

- Communication statements $c!e$ and $c?x$ are used to send the value of e along channel c and assign it to x. The communication takes place as soon as both communication partners are ready for it.

- Synchronization statements $c!$ and $c?$ are similar to communication statements except that no value is passed when they are executed.

- $S_1 ; S_2$ denotes sequential composition of two statements: S_1 is executed first and if it terminates S_2 is executed next.

- Guarded command $[GB]$ denotes nondeterministic choice: it terminates if there are no open guards. Otherwise, a statement associated with an open guard is nondeterministically chosen for execution.

- Guarded command $[GC]$ denotes selective waiting: it terminates immediately if there are no open guards. Otherwise, the execution will proceed as soon as an interaction (communication or synchronization) associated with an open guard is possible. After the execution of the (possibly nondeterministically chosen) interaction, the corresponding statement is performed.

- If $[GC]$ contains timeouts (Δr) in open guards and no interaction is possible within the period determined by the smallest value of all timeouts, then the time determined by this smallest timeout-value passes and a statement associated with a timeout that evaluates to this value is chosen for execution. Otherwise, the interactions that are possible within the smallest timeout are considered as above. If the smallest timeout-value is negative the execution of the statement associated with it starts immediately.

- $*G$ denotes repetition: it terminates if none of the guards is open.

- nG denotes iteration: guarded command G is executed n times $(n \geq 0)$.

The repetition $*[\mathsf{true} \longrightarrow S]$ may be abbreviated to $*[S]$. Commands guarded by interactions or timeouts of the form $\mathsf{true} ; \gamma \longrightarrow S$ may be abbreviated to $\gamma \longrightarrow S$. Commands guarded by interactions or timeouts of the form $b ; \gamma \longrightarrow \mathsf{skip}$ may be abbreviated to $b ; \gamma$. If b is syntactically equal to true this can be replaced by γ.

Informally, the specification of a process has the following general form:

$$
\begin{array}{l}
\quad\mathsf{process}\ \mathit{name}\ (\mathit{parameter\ list}) \\
= \\
\quad[\![\ \mathit{declarations} \\
\quad|\ S \\
\quad]\!].
\end{array}
$$

All channels have to be listed as process parameters. If a channel is used for passing values of some type, this type must be associated with the channel.

Informally, the specification of a system has the following general form:

$$
\begin{array}{l}
\quad\mathsf{system}\ \mathit{name}\ (\mathit{parameter\ list}) \\
= \\
\quad[\![\ \mathit{channel\ declarations} \\
\quad|\ P_1 \parallel P_2 \parallel \ldots \parallel P_n \\
\quad]\!],
\end{array}
$$

where $n \geq 1$ and P_i, for $1 \leq i \leq n$, is a process or (sub)system instantiation.

The channels between the components of the system have to be declared. Connecting two processes by a channel is realized by using the same name in both instantiations: once as an input and once as an output (which must be of the same type). Input and output channels of process instantiations that are not connected have to be listed as system parameters. Moreover, every channel name may occur in at most two process instantiations (once as an input and once as an output).

In more advanced applications, usually containing complex data, the need for the user arises to define own, problem-oriented, data types. For this purpose, a VDM-like ([21]) notation is used. For the application of unary operations and functions to elements of some type the dot-notation is used. To spare brackets, the application is defined to be left-associative and has a higher priority than other operations. In this paper, we use:

- basic primitive types: bool (true and false), real (real numbers), nat (natural numbers),

- sequence type,

- tuple type (similar to cartesian product).

The sequence type corresponds to the list type from functional programming ([2]). A sequence can contain any number of not necessarily unique elements of some base type that are totally ordered by their position counting from left to right. The type consisting of sequences with elements of type T is denoted by T*. For instance: $\langle 1,2,3\rangle$ is of type nat*. The empty sequence is denoted by $\langle\rangle$. In VDM, the sequence type comes equipped with a collection of predicates and functions. We mention the predicate = (equality): two sequences are equal if they have the same length and if their elements are identical in each position. To primitive operations on sequences belong:

- $+\!\!+$ — the concatenate (join) operation (defined only for lists of identical types that is, with the same base type): $\langle 1,2,3\rangle +\!\!+ \langle 3,2,1\rangle = \langle 1,2,3,3,2,1\rangle$,
- hd — the operation returning the first element of the sequence (defined only for non-empty sequences): $\mathrm{hd}.\langle 1,2,3,3,2,1\rangle = 1$,
- tl — the operation returning the sequence with the first element removed (defined only for non-empty sequences): $\mathrm{tl}.\langle 1,2,3,3,2,1\rangle = \langle 2,3,3,2,1\rangle$.

If T_1 and T_2 are predefined types then $T_1 \times T_2$ is the type consisting of pairs $p = (t_1, t_2)$, with t_1 of the type T_1 and t_2 of the type T_2. On this type two operations are defined: $p.1$ delivers the value of the first element and $p.2$ the value of the second element of p. In the same way tuple types with three or more elements can be defined.

3 Flexible assembly cell

In this section, we present a hierarchical specification of the flexible manufacturing cell described in [3] and schematically presented in Figure 1. This cell consists of five workstations, one input-output station, and an automatic transport system. The production is organized by a scheduler that decides what task has to be performed on which workstation. The specification of the scheduler is beyond the scope of this paper. A model of the same manufacturing cell including a scheduler can be found in [14]. Here, we only present a specification of workstations and the transport system with their controllers. For the purpose of this paper, it does not matter what kind of workstations belong to the manufacturing system. A relevant assumption is that they work independently

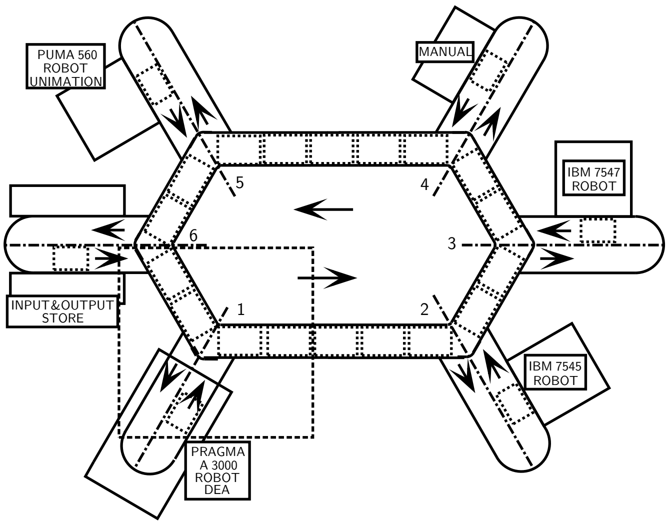

Figure 1: The layout of the flexible manufacturing cell

and that in the system an all-or-nothing resource allocation strategy is followed. The latter means that all parts (and tools if necessary) needed for production stay on one pallet. The manufacturing cell is used to assemble a family of products (for instance, electrical switches, as in the system described in [3]).

The transport system consists of the conveyor that can hold a limited number of pallets and six branches to move pallets to and from the (work)stations. We assume that every branch may hold at most one pallet. In the model, the conveyor is logically split into six parts associated with the (work)stations. At each branch, we distinguish the conveyor switch (as schematically drawn in Figure 2) able to direct the pallet to the corresponding branch (1), or to pass it to the next conveyor part (2), or to move it from the branch to the next conveyor part (3).

The structure of the model, as drawn in Figure 3, corresponds to the structure of the flexible manufacturing cell. As already mentioned, the scheduler is not specified but it is included in the scheme for completeness. In the model, six subsystems are distinguished: each of them associated with one (work)station and each of them including the conveyor part and the branch leading to this station. Five subsystems are of the same kind (S_w). Subsystem S_{io}, taking care of the input and output of the manufacturing cell, is slightly different. In the sequel, these two subsystems are specified in detail. The structure of the subsystem S_w is graphically presented in Figure 4.

To model material and information flows, we use

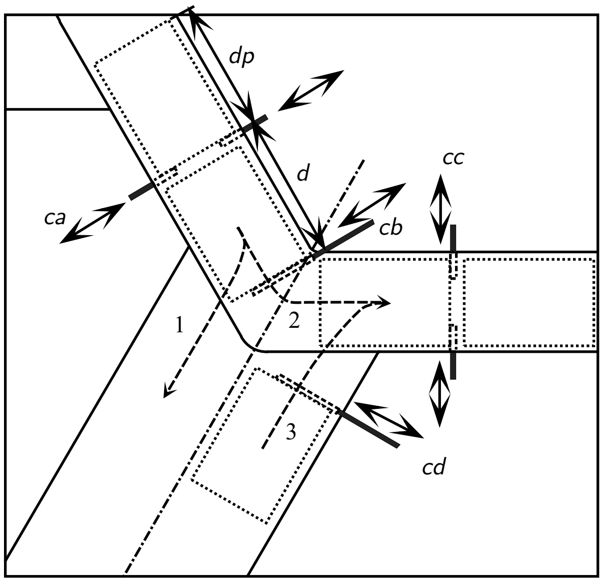

Figure 2: The conveyor switch

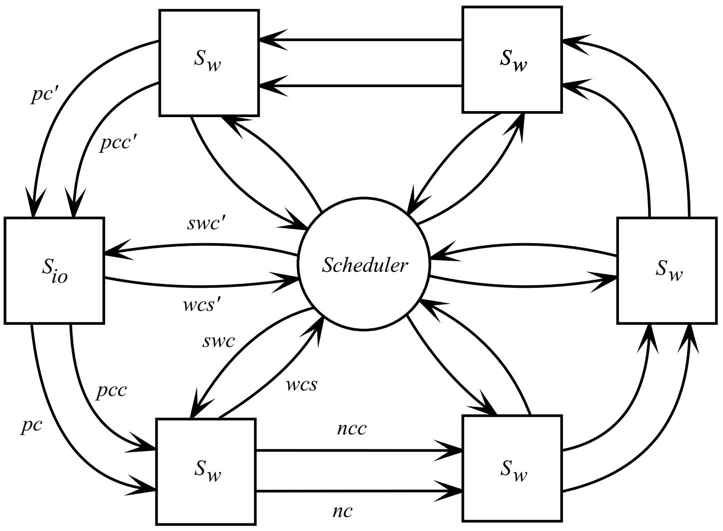

Figure 3: The structure of the FMS model

the following datatypes:

- ptype = $\{1,2,3,4\}$ — the numbers correspond with four product types,
- pallet = ptype,
- task = $\{1,\ldots,99\}$ — the operations,
- time — non-negative real numbers,
- wid = $\{1,\ldots,6\}$ — workstation identification numbers,
- pinf = ptype$\times$task*$\times$wid — the information associated with pallets.

In the specification of the workstation we use function asstime that provides the value of the assembly time for possible combinations of product type and task (this time depends also on the workstation). The workstation (W) performs the following cycle controlled by the workstation controller (WC) via channel cwn. It receives a pallet from the conveyor (communication via jw) and a task number from the controller (communication via cwt). It works on the pallet for the by asstime specified period and sends a signal to its controller when it is ready with the job (synchronization via wc). Then it waits on the subsequent job. If a new pallet has to be processed, it passes the current pallet to the conveyor (via wcj) before the new pallet can be received.

```
process $W$ (in $cwn$ : bool, $cwt$ : task, $jw$ : pallet,
             out $wcj$ : pallet, $wc$, val $id$ : wid)
=
  $[\![$ var $new$ : bool ; $t$ : task ; $p$ : pallet ;
    $cwn?new$ ; $jw?p$ ;
    $*[$ $cwt?t$ ; $\Delta$ asstime.$p.t.id$ ; $wc!$ ;
       $cwn?new$ ; $[$ $new \longrightarrow wcj!p$ ; $jw?p$ $]$
     $]$
  $]\!]$
```

Each time the workstation receives or sends a pallet, the corresponding controller receives (via cwc) or sends (via wcc) the information associated with the pallet. The partner of this information exchange is the conveyor controller (CC). A part of the received information is immediately sent (via cwt) to the workstation: the number of the task to be performed. When the workstation has finished the processing, the controller sends via wcs the pallet information to the scheduler that determines the next task to be performed on the pallet and the workstation that has to perform it. Via channel swc it receives these data and updates the pallet information. Via cwn the controller then informs the workstation whether or not the current pallet has to move on. The pallet does not need to move on if it has to be processed again at this workstation.

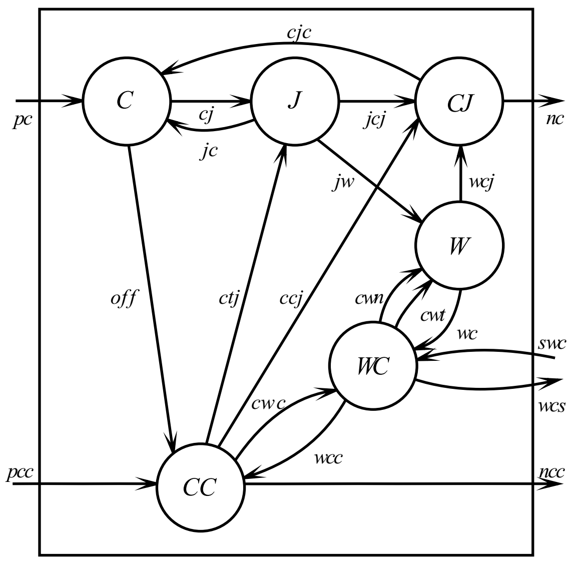

Figure 4: The structure of the subsystem S_w

```
process $WC$ (in $swc$ : task×wid, $cwc$ : pinf, $wc$,
              out $wcs$ : pinf, $cwn$ : bool, $cwt$ : task,
                  $wcc$ : pinf,
              val $id$ : wid)
=
  $[\![$ var $new$ : bool ; $tid$ : task×wid ; $p$ : pinf ;
    $cwn!$true ; $cwc?p$ ;
    $*[$ $cwt!\mathrm{hd}.(p.2)$ ; $wc?$ ; $wcs!p$ ; $swc?tid$ ;
       $p.2 := \langle tid.1\rangle +\!\!+\, p.2$ ; $p.3 := tid.2$ ;
       $new := p.3 \neq id$ ; $cwn!new$ ;
       $[\, new \longrightarrow wcc!p$ ; $cwc?p\,]$
     $]$
  $]\!]$
```

The conveyor part (C) models the transport of the pallets. If there are free places on the conveyor, a pallet can be placed on it (modelled by a communication on the channel pc). To properly model the fact that the conveyor is continuously moving, with every pallet the moment in time is associated at which the pallet can reach the end of this conveyor part if it is not obstructed by the pallets in front of it. If a pallet leaves the conveyor part (communication via cj) the time values associated with remaining pallets are updated to take a possible obstruction into account. The obstruction takes place if pallets on the conveyor are held up by catches ca and cb of Figure 2 to allow passing a pallet to the workstation or from the workstation to the next conveyor part. For pallets p obstructed either by the catch ca or by pallets in front of them the moment in time at which they can reach the end of the conveyor part becomes then (see Figure 2):

$$\tau +\ d +$$
$$\text{(the number of pallets between } p \text{ and } ca) * dp.$$

The time values of the pallets that are not obstructed remain the same. Function upd:

$$(\text{pallet}\times\text{time})^* \times \text{time}\times\text{time}\times\text{nat} \rightarrow (\text{pallet}\times\text{time})^*$$

takes care of the update of these time values. It is formally defined by:

$$\mathrm{upd}.\langle\rangle.t.d.n = \langle\rangle$$
$$\mathrm{upd}.(\langle a\rangle +\!\!+\, l).t.d.n = \langle(a.1, (a.2)\mathsf{max}(t + d * n))\rangle +\!\!+ \\ \mathrm{upd}.l.t.d.(n+1).$$

Each time a pallet is about to leave the conveyor part, a signal (*off*) is given to the conveyor controller to take care of the proper material and information flow. When the pallet actually left the conveyor part, that is, it is passed to the workstation (modelled by jc) or moved on the next conveyor part (modelled by cjc), the time values associated with remaining pallets are updated.

```
process  $C$  (in $pc$ : pallet, $jc, cjc$, out $cj$ : pallet,
              $off$,
              val $cap$ : nat, $d, dp$ : time)
=
  $[\![$ var $p$ : pallet ; $pl$ : (pallet×time)$^*$ ; $w$ : bool ;
    $pl := \langle\rangle$ ; $w :=$false ;
    $*$ [ len.$pl < cap$ ; $pc?p$
                $\longrightarrow pl := pl +\!\!+ \langle(p, \tau + d * cap)\rangle$
      $[\!]\ \neg w \wedge$ len.$pl > 0$ ; $\Delta(\mathrm{hd}.pl.2 - \tau)$
                $\longrightarrow cj!(\mathrm{hd}.pl.1)$ ; $off!$ ; $w :=$true
      $[\!]\ w$ ; $cjc?$ $\longrightarrow pl :=$ upd.(tl.$pl$).$(\tau + d).dp.0$ ;
                $w :=$false
      $[\!]\ w$ ; $jc?$  $\longrightarrow pl :=$upd.(tl.$pl$).$(\tau + d).dp.0$ ;
                $w :=$false
      ]
  $]\!]$
```

The following two components model the conveyor switch. The junction (J) passes a pallet either to the next conveyor part through the conjunction (via jcj) or to the workstation (via jw). The conveyor controller decides what operation needs to be performed (communication via ctj). Time passing commands $\Delta d1$ and $\Delta d2$ model the duration of the physical movements of a pallet ($d1 \geq d$ and $d2 \geq d$).

```
process $J$ (in $cj$ : pallet, $ctj$ : bool,
             out $jcj, jw$ : pallet, $jc$,
             val $d1, d2$ : time)
=
  $[\![$ var $p$ : pallet ; $tow$ : bool ;
    $*[\ cj?p \longrightarrow ctj?tow$ ;
            $[\ \neg tow \longrightarrow jcj!p\ ;\ \Delta d2$
            $[\!]\ tow \quad \longrightarrow jw!p\ ;\ \Delta d1\ ;\ jc!$
            $]$
     $]$
  $]\!]$
```

The conjunction (CJ) passes a pallet to the next conveyor part either from the junction (via jcj) or from the workstation (via wcj). Also in this case, the conveyor controller decides what operation needs to be performed (communication via ccj). Time passing commands $\Delta d2$ and $\Delta d3$ model, again, the duration of the physical movements of a pallet ($d3 \geq d$). Time passing command $\Delta d3'$ models additional spacing before a pallet can be placed on the next conveyor part ($d3' \geq 0$).

```
process $CJ$ (in $jcj, wcj$ : pallet, $ccj$ : bool,
              out $nc$ : pallet, $cjc$,
              val $d2, d3, d3'$ : time)
=
  $[\![$ var $p$ : pallet ; $fromw$ : bool ;
    $*[\ ccj?fromw$
       $\longrightarrow [\ \neg fromw \longrightarrow jcj?p\ ;\ nc!p\ ;\ cjc!\ ;\ \Delta d2$
           $[\!]\ fromw \quad \longrightarrow wcj?p\ ;\ \Delta d3'\ ;\ nc!p\ ;\ \Delta d3$
           $]$
     $]$
  $]\!]$
```

The conveyor controller (CC) decides how the switch operates. If the workstation does not contain any pallet and a pallet destined for it arrives at the end of the conveyor part, the controller makes the junction pass the pallet to the workstation (operation 1). If the workstation does contain a pallet or the pallet at the end of the conveyor is not destined for it, the controller makes the junction and the conjunction pass the pallet to the next conveyor part (operation 2). If the workstation controller requested removing the current pallet from the workstation (modelled by the communication via wcc), the conveyor controller makes the conjunction pass the pallet from the workstation to the next conveyor part (operation 3). Moreover, the conveyor controller assures the synchronization of the material and the information flows. Because the conveyor controller must allways be able to communicate via pcc, two additional booleans are used in the specification: tow and $fromw$.

```
process $CC$ (in $wcc, pcc$ : pinf, $off$,
              out $ncc, cwc$ : pinf, $ctj, ccj$ : bool,
              val $id$ : wid)
=
  $[\![$ var $p$ : pinf ; $pl$ : pinf* ; $emptyw, tow, fromw$ : bool ;
    $pl := \langle\rangle$ ; $emptyw :=$ true ; $tow :=$ true ;
    $fromw :=$ false ;
    $*\ [\ pcc?p \qquad\qquad \longrightarrow pl := pl +\!\!+ \langle p \rangle$
      $[\!]$ hd.$pl$.3 $= id \wedge emptyw$ ; $off$?
                          $\longrightarrow ctj$!true ; $cwc$!(hd.$pl$) ;
                              $pl$ :=tl.$pl$ ; $emptyw$ :=false
      $[\!]$ hd.$pl$.3 $\neq id \vee \neg emptyw$ ; $off$?
                          $\longrightarrow ctj$!false ; $tow$ :=false
      $[\!]\ \neg tow$ ; $ccj$!false $\longrightarrow ncc$!(hd.$pl$) ; $pl$ :=tl.$pl$ ;
                              $tow$ :=true
      $[\!]\ wcc?p \qquad\qquad \longrightarrow fromw$ :=true
      $[\!]\ fromw$ ; $ccj$!true $\longrightarrow ncc!p$ ; $emptyw$ :=true ;
                              $fromw$ :=false
      $]$
  $]\!]$
```

Formally, the subsystem S_w is defined by:

```
system $S_w$ (in $pc$ : pallet, $pcc$ : pinf, $swc$ : task$\times$wid,
              out $nc$ : pallet, $ncc$ : pinf, $wcs$ : pinf,
              val $cap$ : nat, $id$ : wid,
                  $d, dp, d1, d2, d3, d3'$ : time)
=
  $[\![$ channel $cj, jcj, jw, wcj$ : pallet,
               $cwn, ctj, ccj$ : bool, $cwc, wcc$ : pinf,
               $cwt$ : task, $off, wc, jc, cjc$ ;
    $W(cwn, cwt, jw, wcj, wc, id)\ \|$
    $WC(swc, cwc, wc, wcs, cwn, cwt, wcc, id)\ \|$
    $C(pc, jc, cjc, cj, off, cap, d, dp)\ \|$
    $J(cj, ctj, jcj, jw, jc, d1, d2)\ \|$
    $CJ(jcj, wcj, ccj, nc, cjc, d2, d3, d3')\ \|$
    $CC(wcc, pcc, off, ncc, cwc, ctj, ccj, id)$
  $]\!]$.
```

As already mentioned, the subsystem S_{io} slightly differs from S_w. To be more specific, replacing components W and WC by IO and IOC, and removing the channels cwn and cwt in Figure 4 gives exactly the structure of S_{io}. The two components are defined below. In this model, ready pallets are further not considered.

In the input-output station, pallets are generated and accepted when the processing is completed. After the initial generation of n pallets, the amount of pallets in the cell is kept constant. The identification number of this station is 6.

```
process $IO$ (in $jw$ : pallet, out $wcj, wc$ : pallet,
              val $n$ : nat)
=
   $[\![$ var $p$ : pallet ;
     distribution $\lambda$ : discrete uniform (1,4) ;
     $n[\, p := \text{sample}.\lambda\, ;\, wc!p\, ;\, wcj!p\, ]$ ;
     $*[\, jw?p\, ;\, p := \text{sample}.\lambda\, ;\, wc!p\, ;\, wcj!p\, ]$
   $]\!]$

   process $IOC$ (in $swc$ : task$\times$wid, $cwc$ : pinf,
                      $wc$ : pallet,
                  out $wcs$ : pinf, $wcc$ : pinf,
                  val $n$ : nat)
=
   $[\![$ var $tid$ : task$\times$wid ; $p$ : pinf ; $pt$ : pallet ;
     $n[\ wc?pt\, ;\, wcs!(pt, \langle\rangle, 6)\, ;\, swc?tid\, ;$
         $wcc!(pt, \langle tid.1\rangle, tid.2)$
       $]$ ;
     $*[\ wc?pt\, ;\, cwc?p\, ;\, wcs!(pt, \langle\rangle, 6)\, ;\, swc?tid\, ;$
         $wcc!(pt, \langle tid.1\rangle, tid.2)$
       $]$
   $]\!]$
```

Formally, the subsystem S_{io} is defined by:

```
   system $S_{io}$ (in $pc$ : pallet, $pcc$ : pinf,
                   $swc$ : task$\times$wid,
              out $nc$ : pallet, $ncc$ : pinf, $wcs$ : pinf,
              val $cap, n$ : nat,
                  $d, dp, d1, d2, d3, d3'$ : time)
=
   $[\![$ channel $cj, jcj, jw, wcj$ : pallet, $ctj, ccj$ : bool,
              $cwc, wcc$ : pinf, $\mathit{off}$, $wc$ ;
     $IO(jw, wcj, wc, n)\ \|$
     $IOC(swc, cwc, wc, wcs, wcc, n)\ \|$
     $C(pc, jc, cjc, cj,\ \mathit{off},\ cap, d, dp)\ \|$
     $J(cj, ctj, jcj, jw, jc, d1, d2)\ \|$
     $CJ(jcj, wcj, ccj, nc, cjc, d2, d3, d3')\ \|$
     $CC(wcc, pcc,\ \mathit{off},\ ncc, cwc, ctj, ccj, 6)$
   $]\!]$.
```

Using the modelling approach based on concurrent programming results in a well-structured hierarchical model consisting of concise and comprehensible component specifications. The processes are almost straightforward translations of the informal component descriptions. The behaviour of the system components is specified on a reasonable detail level.

4 Concluding remarks

In this paper, we illustrate the application of a real-time concurrent programming language to the specification of discrete industrial systems. The language is well suited to specify complex systems, as it is shown in Section 3. For this purpose, every component of the system is modelled as a collection of parametrized processes. The same language is used for modelling the physical components and their control. Also a higher level control and a scheduler can be modelled using this language ([14]).

Concurrent programming approach is suitable for the specification of industrial systems. It supports modularity and allows separate descriptions of the structure and of the component behaviour. The descriptions of component behaviours are concise and comprehensible, and they are suited for reuse. The definition of the structure of a system is also suited for a graphical representation. The specification language together with the graphical structure representation supports reasoning about models of production processes.

The formal meaning of specifications can be defined by means of an operational semantics that respects concurrency and that is based on the notion of transition systems. The behaviour of specified systems can be analysed by means of sequential or distributed simulation (comparable to, for instance, the ones discussed in [9] or [12]). For the verification of the communication behaviour embedded in a model, the underlying formal framework of CSP can be used.

A software tool supporting specification activities based on the presented language is under development. The purpose of this tool is to facilitate a direct transition from specifications to implementations (as described in [16] for a different specification language). Our goal is to use the executable specification of the control part of the validated model directly in the industrial system. In this way, the development time of the control system can be shortened and its correctness with respect to the specification can be guaranteed.

References

[1] J.H.A. Arentsen. *Factory Control Architecture. A System Approach.* Ph.D. Thesis, Department of Mechanical Engineering, Eindhoven University of Technology, 1989.

[2] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[3] H. van Brussel, Y. Peng, and P. Valckenaers. “Modelling Flexible Manufacturing Systems Based on Petri Nets”. *Annals of the CIRP*, vol. 42 (1), 1993, pp. 479 – 484.

[4] A. Burns and G. Davies. *Concurrent Programming.* International Computer Science Series. Addison-Wesley, 1993.

[5] R. David and H. Alla. "Petri Nets for Modeling of Dynamic Systems — A Survey". *Automatica*, Vol. 30 (2), 1994, pp. 175 – 202.

[6] K.M. van Hee, L.J. Somers, and M. Voorhoeve. "Executable Specifications for Distributed Information Systems". In E.D. Falkenberg and P. Lindgreen (eds.), *Information System Concepts: An In-depth Analysis*, North-Holland, 1989.

[7] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, New York, 1985.

[8] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* Springer-Verlag, 1991.

[9] J. Koster. *Modelling Industrial Systems: Theory and Applications.* Ph.D. Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1993.

[10] ISO 8807. *Information processing systems — Open systems interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour*, 1989.

[11] S. Mauw and G.J. Veltink. "A Process Specification Formalism". *Fundamenta Informaticae*, vol. 13, 1990, pp. 85 – 139.

[12] J. Misra. "Distributed Discrete-Event Simulation". *Computing Surveys*, vol. 18 (1), 1986, pp. 39–65.

[13] J.M. van de Mortel-Fronczak and J.E. Rooda. "Application of Concurrent Programming to Specification of Industrial Systems". Accepted for INCOM'95.

[14] N.J.M. van den Nieuwelaar. *Modelling of Industrial Systems Using Petri Nets and Concurrent Programming — a Comparison.* Internal report WPA 420051, Eindhoven University of Technology, 1995.

[15] INMOS Limited. *Occam 2 Reference Manual*, 1988.

[16] R. Overwater and J.E. Rooda. "Developing Industrial Systems According to the Process Interaction Approach". In E.A. Puente (ed.), *Preprints of the IFAC Symposium on Information Control Problems in Manufacturing Technology*, 1989, pp. 171–176.

[17] G.B. Reddy, S.S.N. Murty, and K. Ghosh. "Timed Petri Net: An Expeditious Tool for Modelling and Analysis of Manufacturing Systems". *Mathematical Computer Modelling*, vol. 18 (9), 1993, pp. 17 – 30.

[18] M. Silva and R. Valette. "Petri Nets and Flexible Manufacturing Systems". In G. Rozenberg (ed.), *Advances in Petri Nets*, LNCS, vol. 424, Springer-Verlag, 1990, pp. 374 – 417.

[19] H. Smit. *A Hierarchical Control Architecture for Job-shop Manufacturing Systems.* Ph.D. Thesis, Department of Mechanical Engineering, Eindhoven University of Technology, 1992.

[20] O. Tanir and S. Sevinc. "Defining Requirements for a Standard Simulation Environment". *Computer*, vol. 27 (2), 1994, pp. 28 – 34.

[21] J.G. Turner and T.L. McCluskey. *The Construction of Formal Specifications. An Introduction to the Model-based and Algebraic Approaches.* McGraw-Hill, 1994.

[22] P.A.C. Verkoulen and M.M. de Brouwer. "Two Case Studies in ExSpect". *Computing Science Notes*, 90/19, Eindhoven University of Technology, 1990.

[23] A.M. Wortmann. *Modelling and Simulation of Industrial Systems.* Ph.D. Thesis, Department of Mechanical Engineering, Eindhoven University of Technology, 1991.