Integrating Continuous-Time and Discrete-Event Concepts in Modelling and Simulation of Manufacturing Machines

D.A. van Beek [1], S.H.F. Gordijn, J.E. Rooda

Eindhoven University of Technology, Department of Mechanical Engineering, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Using simulation models for the development and testing of control systems can have significant advantages over using real machines. This paper demonstrates the suitability of the χ language for modelling, simulation and control of manufacturing machines. The language integrates a small number of powerful orthogonal continuous-time and discrete-event concepts. The continuous-time part of χ is based on DAEs; the discrete-event part is based on a CSP-like concurrent programming language. Models are specified in a symbolic mathematical notation. A case study is presented of a transport system consisting of conveyor belts.

Key words: machine modelling; control system testing; hybrid; discrete-event; continuous-time

1 Introduction

Using simulation models for the development and testing of control systems can have significant advantages over using real machines:

- using the real machines may be hazardous because of the possibility of damage due to errors,
- the control system may already be operative so that it must be taken out of production to test the new version,
- or the real machines can still be under development.

The main disadvantage of simulation-based testing is the time and effort needed to model the system. Therefore it is important that a sufficiently accurate model can be developed in a reasonable amount of time. This is possible with the combined

[1] E-mail: vanbeek@wtb.tue.nl

continuous-time / discrete-event language χ presented in this paper. The paper is based on [8].

Many modelling languages are primarily suited to the specification of either discrete-event or continuous-time models. In order to be able to specify combined systems, some languages have been extended with continuous-time or discrete-event constructs, respectively.

Some examples of continuous-time languages that have been extended with discrete-event constructs are: ACSL [18], Dymola [11], Omola [1] and gPROMS [5]. The discrete-event constructs of the first three languages are limited to discontinuity handling using time-events or state-events. The gPROMS language is based on the continuous-language Speedup [3]. It has been developed for modelling systems occurring in the process industries. Its discrete-event part is based on composition of sequential and parallel blocks. The language is, however, not suited to modelling of discrete-event (control) systems.

The language SIMAN [22], is an example of a discrete-event modelling language that has been extended with assignment and pointer based constructs for representing equations, using the general purpose programming language C [16]. This leads to low level code for representing equations, so that the modeller is forced to concentrate on equation *solving* instead of simply *specifying* the equations describing the continuous-time systems. Prosim [10] is another discrete-event based language in which a limited number of continuous constructs are present. Petri Nets (see [9] for a survey paper) are also based on discrete-event concepts. They have been extended with continuous-time constructs (e.g. see [23]). In [4] such constructs are used for a more efficient approximation of discrete-event systems; the continuous constructs are generally not designed to support the specification of complex continuous systems.

In [17] the COSMOS language is described. Its discrete-event part is based on the activation and passivation of processes. In the continuous part differential equations can be specified. The language also comprises an elaborate experiment description part, so that the model and the experiment descriptions are clearly separated.

2 The χ language

The χ language is based on a small number of orthogonal language constructs which makes it easy to use and to learn. Where possible, the continuous-time and discrete-event parts of the language are based on similar concepts. Parametrized processes and systems provide modularity, parallelism with communication and synchronization, and hierarchical modelling. The language is based on mathematical concepts with well defined semantics. Unlike many other simulation languages, the χ language

uses a symbolic notation for specifications and ASCII equivalents for simulations. The symbolic notation makes specifications easier to read and to develop. Compare for example the symbolic notation of

$$h' = k\sqrt{p_\mathrm{i} - p_\mathrm{o}}$$
$$\Delta\, t - \tau$$

with its ASCII equivalent for simulation purposes

```
h' = k * sqrt(p_i - p_o)
delta t - time
```

The pronunciation of the χ symbols used in this paper is shown in Table 1.

Table 1
Pronounciation of the χ symbols

χ	chi	$\multimap$	link	$*$	repeat
proc	process	?	receive	[]	or
syst	system	!	send	]	end
$[\![$	begin block	Δ	delta	$\longrightarrow$	then
$]\!]$	end block	∇	nabla	$\perp$	nil

In this paper, only the language constructs used in the example are treated. The syntax and operational semantics of the language constructs are explained in an informal way. A treatment of all language constructs and design considerations is presented in [2].

The model of a system consists of a number of process (or system) instantiations, and channels connecting these processes. Systems are parametrized.

syst *name*(*parameter declarations*) =
$[\![$ *channel declarations*
| *process or system instantiations*
$]\!]$

The parameter declaration of a process is identical to that of a system. A process may consist of a continuous-time part only (DAEs), a discrete-event part only, or a combination of both. The links (explained in the next section) provide a means to access continuous variables from other processes.

proc *name*(*parameter declarations*) =
$[\![$ *variable declarations*
; *initialization*
| *links*

| *DAEs*
| *discrete-event statements*
]‖

The continuous-time and discrete-event parts of the language are based on similar concepts. Processes have local variables only; all interactions between processes take place by means of channels. A channel connects two processes or systems. Continuous channels are symmetrical; discrete channels are used for output in one process and input in the other. Channels are declared locally in systems and are either discrete (e.g. p : int, q : void) or continuous (e.g. v : [m/s]). The special void data type denotes the absence of data. Therefore the declaration q : void declares a synchronisation channel q. Channels can also be declared as process or system parameters. When declared as parameters, discrete channels are used for either input (e.g. p : ? int, q : ? void), or output (e.g. p : ! int, q : ! void). Continuous channel parameters are declared as c : $-\circ$ [m/s]. A continuous channel is represented graphically by a line (optionally ending in a small circle to denote either the direction of a flow or a cause-effect relationship), a discrete channel by an arrow. Processes and systems are represented by circles.

2.1 Data types, variables and links

All data types and variables are declared as either continuous or discrete. This is an important distinction with other hybrid modelling languages in which the type of a variable is often implicitly inferred from its use.

The value of a discrete variable is determined by assignments. Between two subsequent assignments the variable retains its value. The value of a continuous variable, on the other hand, is determined by equations. An assignment to a continuous variable (initialization) determines its value for the current point of time only.

Some discrete data types are predefined like bool (boolean), int (integer) and real. Since all continuous variables are assumed to be of type real, they are defined by specifying their units only. For example, the declaration c : [m/s] defines a continuous variable (in a process) or channel c (in a system).

In parameter declarations, c : $-\circ$ [m/s] defines a continuous channel parameter c which may be linked to a continuous variable of type [m/s] using the link symbol $-\circ$

$$c -\circ var$$

The variable *var* and the channel c must be of the same (continuous) data type. Linking is the only operation allowed on continuous channels in processes. By

linking, a continuous channel relates a variable of one process to a variable of another process by means of an equality relation. Consider two variables x_a, x_b in different processes, that are linked to a continuous channel c ($c \multimap x_a$ and $c \multimap x_b$). The channel and links cause the equation $x_a = x_b$ to be added to the set of DAEs of the system.

2.2 The continuous-time part of χ

The continuous-time part of χ is based on Differential Algebraic Equations (DAEs). A time derivative is denoted by a prime character (e.g. x'). A summary of the continuous-time language constructs is given in Table 2. In this table *re* denotes a real expression, possibly including derivatives. An informal explanation of the constructs follows below.

Table 2
BNF syntax of the continuous-time part of χ

DAE	::=	$re = re \mid [GE] \mid DAE, DAE$
GE	::=	$b \longrightarrow DAE \mid GE \;[]\; GE$

DAEs are separated by commas

$$DAE_1, DAE_2, \ldots, DAE_n$$

If the set of DAEs depends on the state of a system, *guarded DAEs* can be used

$$[\, b_1 \longrightarrow DAEs_1 \;[]\; \ldots \;[]\; b_n \longrightarrow DAEs_n \,]$$

which is pronounced as: 'guarded equations, if b_1 then $DAEs_1$, or ... or if b_n then $DAEs_n$, end'. The boolean expression b_i ($1 \leq i \leq n$) denotes a *guard*, which is open if b_i evaluates to true and is otherwise closed. At any time at least one of the guards must be open so that the *DAEs* associated with an open guard can be selected.

2.3 The discrete-event part of χ

The discrete-event part of χ is a CSP-like [14] real-time concurrent programming language, described in [19] and [20]. Our notation is derived from [15]. We have adopted the CSP-like constructs because of:

– the concurrent processes, that are well suited to modelling the parallelism found in industrial systems,
– the sound mathematical background of the formalism,

– the synchronous nature of the interactions, which means that buffers are modelled explicitly and cannot be hidden in interactions,
– the lack of global variables, which enables robust modelling,
– the possibility to extend the CSP language with, among others, the selective waiting construct, which is explained below together with the other language constructs.

A summary of the discrete-event language constructs is given in Table 3.

Table 3
BNF syntax of the discrete-event part of χ

E	$::=$	$c\,!\,e \mid c\,?\,x \mid c\,! \mid c\,? \mid \Delta\, t \mid \nabla\, r$
GB	$::=$	$b \longrightarrow S \mid GB \,[]\, GB$
GW	$::=$	$b;\, E \longrightarrow S \mid GW \,[]\, GW$
G	$::=$	$[GB] \mid [GW]$
S	$::=$	$x := e \mid E \mid G \mid *G \mid S;\, S$

Interaction between discrete-event parts of processes takes place by means of synchronous message passing or by synchronization

$$c\,!\,e \qquad c\,?\,x \qquad c\,! \qquad c\,?$$

Consider the channel c connecting two processes. Execution of $c\,!\,e$ in one process causes the process to be blocked until $c\,?\,x$ is executed in the other process, and vice versa. Subsequently the value of expression e is assigned to variable x. Synchronization is denoted by $c\,!$ and $c\,?$. The only difference with communication is that no data is exchanged.

Time passing is denoted by

$$\Delta\, t$$

where t is a real expression. A process executing this statement is blocked until the time is increased by t time-units.

Selection ($[GB]$) is denoted by

$$[\, b_1 \longrightarrow S_1 \,[]\, b_2 \longrightarrow S_2 \,[]\, \ldots \,[]\, b_n \longrightarrow S_n \,]$$

which is pronounced as: 'selection, if b_1 then S_1, or if b_2 then S_2 or ... or if b_n then S_n, end'. The boolean expression b_i $(1 \leq i \leq n)$ denotes a *guard*, which is open if b_i evaluates to true and is otherwise closed. After evaluation of the guards, one of the statements S_i associated with an open guard b_i is executed.

Selective waiting ([*GW*]) is denoted by

$$[\, b_1 ;\ E_1 \longrightarrow S_1 \,[]\, \ldots \,[]\, b_n ;\ E_n \longrightarrow S_n \,]$$

Consider for example the selective waiting statement

$$[\, b_1 ;\ p\,?\,x \longrightarrow S_1 \,[]\, b_2 ;\ \Delta\, t \longrightarrow S_2 \,]$$

this is pronounced as: 'selective waiting, if b_1 and p receive x then S_1, or if b_2 and delta t then S_2, end'. An event statement E_i which is prefixed by a guard b_i ($b_i ;\ E_i$) is enabled if the guard is open and the event specified in E_i can actually take place. Continuous variables are not allowed in these guards. There are four types of event statement: input ($c\,?\,e$ or $c\,?$), output ($c\,!\,x$ or $c\,!$), time ($\Delta\, t$), and state ($\nabla\, r$, explained in the following section). The process executing [*GW*] remains blocked until at least one event statement is enabled. Then, one of these (E_i) is chosen for execution, followed by execution of the corresponding S_i. Time-outs are event statements of the form $\Delta\, t$ that are prefixed by a guard ($b ;\ \Delta\, t$). A process cannot be blocked in a selective waiting statement with time-outs for longer than the smallest time-out time t_s (provided the associated time-out guard is open). If no other event statements are enabled within that period, a statement S associated with an enabled time-out of time t_s is executed. Please note that guards that are always true may be omitted together with the succeeding semicolon. Therefore 'true ; E' may be abbreviated to 'E'.

Repetition of the statements [*GB*] and [*GW*] is denoted by

$$*[GB] \quad \text{and} \quad *[GW]$$

respectively. The repetition terminates when all guards are closed. The repetition $*[\,\text{true} \longrightarrow S\,]$ may be abbreviated to $*[\,S\,]$.

2.4 Interaction between the continuous and discrete parts of χ

In the discrete-event part of a process, assignments can be made to discrete variables occurring in DAEs or in the boolean guards of guarded DAEs. In the former case the DAEs will be evaluated with new values, in the latter case different DAEs may be selected. Continuous variables are initialized immediately after the declarations, and may be reinitialized in the discrete-event part. In both cases the symbol ::= (e.g. $x ::= 0$) is used.

By means of the *state event* statement

$$\nabla\, r$$

the discrete-event part of a process can synchronize with the continuous part of a process. Execution of ∇r, where r is a relation involving at least one continuous variable, causes the process to be blocked until the relation becomes true.

3 A conveyor line example

This section illustrates the use of the χ language by an example of a transport system shown in Figure 1. The system consists of a line of conveyor belts driven by motors, and is used for the transportation and buffering of boxes. Each conveyor belt is equipped with a sensor (represented in the figure by a small rectangle), which detects the presence of a box. Both the model of the physical system and the discrete-event control system are presented.

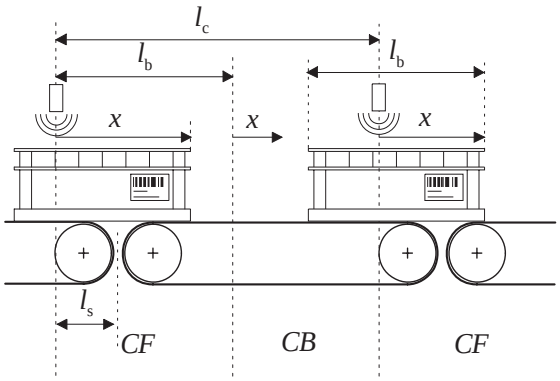

Fig. 1. The transport system.

The iconic model of the transport system is given in Figure 2. The textual model of the transport system T follows below.

$$
\begin{aligned}
&\text{syst } T(\ v_{11} :\ \multimap \text{vel}\\
&\qquad , \ p_1 : ?\,\text{id},\ p_7 : !\,\text{id},\ q_0 : ?\,\text{void},\ q_3 : !\,\text{void}\\
&\qquad , \ l_c, l_b, l_s, v_{set} : \text{real}\\
&\qquad) =\\
&[\![\ v_{11}, v_{12}, v_{13}, v_{21}, v_{22}, v_{23}, v_{31}, v_{32}, v_{33} : \text{vel}\\
&,\ q_1, q_2, on_1, on_2, on_3, off_1, off_2, off_3 : \text{void}\\
&,\ s_1, s_2, s_3, m_1, m_2, m_3 : \text{bool},\ p_2, p_3, p_4, p_5, p_6 : \text{id}\\
&|\ C(q_0, q_1, on_1, off_1, m_1) \parallel M(v_{11}, v_{12}, v_{13}, m_1, v_{set}) \parallel S(s_1, on_1, off_1)\\
&\parallel C(q_1, q_2, on_2, off_2, m_2) \parallel M(v_{21}, v_{22}, v_{23}, m_2, v_{set}) \parallel S(s_2, on_2, off_2)\\
&\parallel C(q_2, q_3, on_3, off_3, m_3) \parallel M(v_{31}, v_{32}, v_{33}, m_3, v_{set}) \parallel S(s_3, on_3, off_3)\\
&\parallel CB(v_{12}, p_1, p_2, l_b, l_c) \parallel CF(v_{13}, v_{21}, p_2, p_3, s_1, l_b, l_s)\\
&\parallel CB(v_{22}, p_3, p_4, l_b, l_c) \parallel CF(v_{23}, v_{31}, p_4, p_5, s_2, l_b, l_s)\\
&\parallel CB(v_{32}, p_5, p_6, l_b, l_c) \parallel CFL(v_{33}, p_6, p_7, s_3, l_b, l_s)\\
&]\!]
\end{aligned}
$$

A conveyor is modelled by the processes *CF* (Conveyor Front) and *CB* (Conveyor

Back). The controller, motor and sensor are modelled by the processes C, M, and S, respectively. The boxes enter the system via the process *CFF* (which is not shown in the figure), and leave the system via process *CFL*. The specifications of the processes *CFF* and *CFL* are not given here, because they depend on the environment of the system. For an example of how they could be specified we refer to [13].

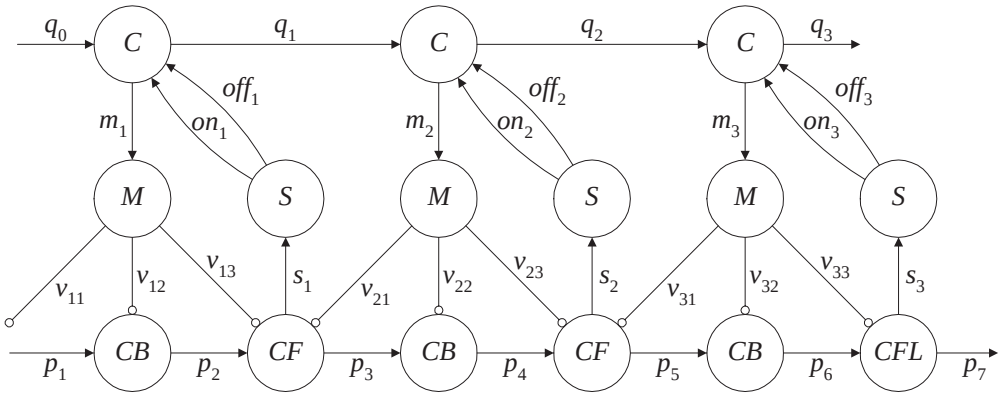

Fig. 2. Model of the transport system.

The user defined types that are used in different processes are listed below.

```
type vel  = [m/s]
,    pos  = [m]
,    id   = int
```

3.1 The model of a motor

The motor is switched on when it receives the value true via channel m_j, and is switched off when it receives the value false. The parameter v_{set} represents the value of the velocity of the conveyor motor when it is switched on. The velocity v of the motor is made available to three processes via the continuous channels v_{j1}, v_{j2} and v_{j3} (see Figures 2 and 3).

```
proc M( v_j1, v_j2, v_j3 : —o vel
      , m_j : ? bool
      , v_set : real
      ) =
‖[ v : vel
, b : bool
; b := false
| v_j1, v_j2, v_j3 —o v
| [   b ⟶ v = v_set
  [] ¬b ⟶ v = 0
  ]
| *[ m_j ? b ]
```

]‖

3.2 The model of a sensor

A sensor is modelled by process S. The control system synchronizes with a sensor by means of the synchronization channels *on* and *off*. These are used by the control system to wait for the sensor to be activated or deactivated. The synchronization *on* ? in a control process can only take place if the sensor is activated. This is modelled by 'b ; *on* ! $\longrightarrow$ skip' in process S, where skip is the empty statement. Execution of the statement *off* ? in the control system causes it to block until the sensor is deactivated. The sensor receives its new state from process *CF* via channel s. The boolean variable b denotes the state of the sensor.

```
proc S(s : ?bool, on, off : !void) =
‖[ b : bool
| b := false
; *[      s?b  ⟶ skip
  []   b; on!  ⟶ skip
  [] ¬b; off!  ⟶ skip
  ]
]‖
```

3.3 The conveyor processes

The assumptions regarding the behaviour of a conveyor are:

- The length (l_b) of a box is less than the length (l_c) of a conveyor.
- If a box is positioned on two conveyors, the movement of the box is determined by the conveyor which supports the major part of the box.
- Initially there are no boxes on the conveyors.
- The control system allows a new box to enter a conveyor only when the conveyor is empty, or its current box is leaving the conveyor. Therefore, the boxes cannot come into contact with each other.

The conveyors are modelled by means of the processes *CF* and *CB* (see Figure 1). Process *CB* models a part of the conveyor immediately preceding the sensor. The first part of the conveyor is modelled by process *CF*, which also models the part after the sensor of the preceding conveyor. The process *CF* determines the state of the sensor. Because the *CF* section starts at the position of the sensor, and the length of the section equals the length of a box, the sensor is activated if and only if the front of a box is present in section *CF*. The second section, *CB*, has a length of $l_c - l_b$.

The specification of the process *CF* is shown below. It is best understood in combination with Figure 3.

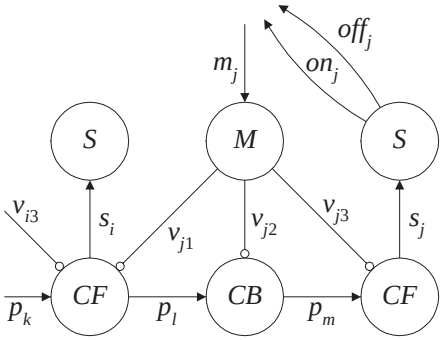

Fig. 3. Model of a conveyor.

Each box is represented by an integer of type id, which is stored in the variable *id*. The variables x, v and v_p represent the position of the front of a box, the velocity of the conveyor belt and the velocity of the preceding conveyor belt, respectively. The boolean variable b equals true when the conveyor supports the major part of the box, and false otherwise.

The variables used in the continuous part are initialized immediately after the declarations ($x ::= \bot$; $b :=$ false). Initially the conveyors are empty. Therefore the variable x is initialized to $\bot$. In this way it becomes undefined. When a continuous variable is initialized to $\bot$, the variable and all equations in which it occurs are temporarily removed from the system, until the variable is initialized to a value other than $\bot$.

The entry of a box onto the conveyor is modelled by the communication $p_k\,?\,id$. As a result, the sensor is activated ($s\,!$ true), and x is initialized to 0. The trajectory of x is determined by the value of its derivative (x'). This value equals either the velocity v_p of the preceding conveyor (if b is false) or the velocity v of the conveyor itself (if b is true), depending on the position of the box. In the discrete-event section, two state-events are used. The first state-event represents the detection that the major part of the box is supported by the conveyor, i.e. $\nabla\, x = l_\mathrm{s} + \frac{1}{2} l_\mathrm{b}$. By subsequently assigning the value true to b, the equation $x' = v$ is selected. The second state-event represents the moment at which the box reaches the process boundary, i.e. $\nabla\, x = l_\mathrm{b}$. Consequently the communication $p_l\,!\,id$ is executed, the sensor is deactivated ($s\,!$ false), and the variable x is reinitialized at $\bot$. From this point on, the position of the box is determined by the next process.

proc $CF($ v_{i3}, v_{j1} : $\multimap$ vel
, p_k : ? id, p_l : ! id, s : ! bool
, $l_\mathrm{b}, l_\mathrm{s}$: real
) =
$[\![\, x$: pos, v, v_p : vel

$$
\begin{array}{l}
\quad , \; b : \text{bool} , \; id : \text{id} \\
\quad ; \; x ::= \bot ; \; b := \text{false} \\
\quad | \; v_{i3} \multimap v_{\text{p}} \\
\quad , \; v_{j1} \multimap v \\
\quad | \; [\; \neg b \longrightarrow x' = v_{\text{p}} \\
\quad\quad [] \quad b \longrightarrow x' = v \\
\quad\quad] \\
\quad | \; *[\; p_k \, ? \, id \, ; \; s \, ! \, \text{true} \, ; \; b := \text{false} \, ; \; x ::= 0 \\
\quad\quad ; \; \nabla \, x = l_{\text{s}} + \tfrac{1}{2} l_{\text{b}} \, ; \; b := \text{true} \\
\quad\quad ; \; \nabla \, x = l_{\text{b}} \, ; \; p_l \, ! \, id \, ; \; s \, ! \, \text{false} \, ; \; x ::= \bot \\
\quad\quad] \\
\quad]\!]
\end{array}
$$

The specification of *CB* is slightly different from the *CF* specification. The conditional equation is no longer necessary, because in this process the box is always carried by the conveyor itself. The discrete event part is also simplified, because the state of the sensor is not affected by process *CB*.

$$
\begin{array}{l}
\text{proc } CB(\; v_{j2} : \; \multimap \text{vel} \\
\qquad\quad , \; p_l : ? \, \text{id} , \; p_m : ! \, \text{id} \\
\qquad\quad , \; l_{\text{b}}, l_{\text{c}} : \text{real} \\
\qquad\quad) = \\
[\![\; x : \text{pos} , \; v : \text{vel} \\
\quad , \; id : \text{id} \\
\quad ; \; x ::= \bot \\
\quad | \; v_{j2} \multimap v \\
\quad | \; x' = v \\
\quad | \; *[\; p_l \, ? \, id \, ; \; x ::= 0 \, ; \; \nabla \, x = l_{\text{c}} - l_{\text{b}} \, ; \; p_m \, ! \, id \, ; \; x ::= \bot \;] \\
\quad]\!]
\end{array}
$$

4 The conveyor line control system

The control system is included, because it clarifies the sensor and actuator based interaction between the control system and the conveyors.

4.1 A simplified control strategy

The specification of the control process *C* follows below (see also Figure 2). The control strategy is straightforward: new boxes are allowed to enter only when the sensor is not activated, and the conveyor is therefore empty. When the synchronization via q_{in} succeeds, the motor of the conveyor is switched on ($m \, ! \, \text{true}$). The control

process then waits until the sensor is activated (*on* ?), so that the box is at the end of the conveyor. The process then checks if the next conveyor is ready to receive a box (q_{out} !). If this is the case, then the synchronization q_{out} ! is executed. If the next conveyor cannot receive a box within 0.1 seconds, the time-out $\Delta\, 0.1$ is executed. As a result the motor is switched off, and the control process waits until the next conveyor can receive a box. Then it starts the motor. Finally, it waits until the box has left the conveyor (*off* ?), before a new box may enter the conveyor.

```
proc C(q_in : ? void, q_out ! void, on, off : ? void, m : ! bool) =
[| *[ q_in ?; m ! true; on ?
   ; [ q_out !   --> skip
     [] Δ 0.1    --> m ! false; q_out !; m ! true
     ]
   ; off ?
   ]
|]
```

4.2 A more efficient control strategy

A more optimal strategy is presented below. In this case, boxes are allowed to enter a conveyor as soon as the box already present on the conveyor has reached the sensor (*on* ?), and is leaving the conveyor. The advantage of this strategy is that there can be two boxes on a conveyor instead of just one, which results in an increased throughput.

Four different states are used in the controller:

(i) EMPTY: At first the conveyor is empty, not running, and is able to receive and transport a box.

(ii) RATS (running at sensor): The conveyor is running with a box that has reached the sensor. At this stage, the controller tries to send to q_{out} to see if the next conveyor can receive the box. If this is not possible within 0.1 seconds ($\Delta\, 0.1$), the motor stops (*m* ! false) and the controller waits until the next conveyor can receive the box.

(iii) ROUT (running out): The box can leave the conveyor. When leaving, two events are possible. Either a second box may enter the conveyor (q_{in} ?), or the box leaving the conveyor may clear the detector (*off* ?) before a second box has entered. If a new box has entered first, the control process subsequently waits for the first box to have left the conveyor (*off* ?), and then waits for the second box to reach the detector (*on* ?).

(iv) REMPTY (running empty): If there is no new box within 5 seconds, the motor stops, and the state *s* becomes EMPTY.

```
proc C(q_in : ? void, q_out ! void, on, off : ? void, m : ! bool) =
```

```
⟦ s : {EMPTY, ROUT, REMPTY,RATS}
| s := EMPTY
; *[ s = EMPTY   ; q_in ?   ⟶ m ! true ; on ? ; s := RATS
   [] s = RATS    ; q_out !  ⟶ s := ROUT
   [] s = RATS    ; Δ 0.1    ⟶ m ! false ; q_out ! ; m ! true ; s := ROUT
   [] s = ROUT    ; q_in ?   ⟶ off ? ; on ? ; s := RATS
   [] s = ROUT    ; off ?    ⟶ s := REMPTY
   [] s = REMPTY ; q_in ?   ⟶ on ? ; s := RATS
   [] s = REMPTY ; Δ 5      ⟶ m ! false ; s := EMPTY
   ]
⟧
```

5 Extensions to the conveyor line example

In this section the conveyors are modelled in more detail. Two extensions are presented: a conveyor motor with start-up and shut-down characteristics, and a conveyor line on which the boxes may collide.

5.1 Extensions to the motor

The motor described in the previous section had two states only: on or off. The motor specified below has two additional states: accelerating and decelerating. The variable s stores the state of the motor, it can take on any of the values in the set {STOPPED, ACC, MAX, DEC}. When the motor is switched on, its velocity v rises with a constant acceleration a_{set} until the maximum velocity v_{set} is reached. The motor process continuously tries to receive a new command from the control system (m_j ? b). When the motor receives a new command via channel m_j, it first checks the state of the motor. If it is turned on and it is not yet at its maximum velocity ($b \wedge s \neq$ MAX) the state is set to accelerate ($s :=$ ACC). If it is turned off and it has not yet stopped ($\neg b \wedge s \neq$ STOPPED), the state is set to decelerate. If the state s equals ACC, the process waits until the motor has achieved its maximum speed ($\nabla v = v_{\text{set}}$), and subsequently switches the state to MAX.

```
proc M( v_j1, v_j2, v_j3 : ⊸ vel
      , m_j : ? bool
      , a_set, v_set : real
      ) =
⟦ v : vel
, b : bool, s : {STOPPED, ACC, MAX, DEC}
; v ::= 0 ; s := STOPPED
| v_j1, v_j2, v_j3 ⊸ v
```

$$
\begin{array}{l}
\mid [\ s = \text{MAX} \vee s = \text{STOPPED} \longrightarrow v' = 0 \\
\quad [] \ s = \text{ACC} \qquad\qquad\qquad\quad \longrightarrow v' = a_{\text{set}} \\
\quad [] \ s = \text{DEC} \qquad\qquad\qquad\quad \longrightarrow v' = -a_{\text{set}} \\
\quad] \\
\mid *[\qquad\quad m_j\,?\,b \qquad \longrightarrow [\ b \wedge s \neq \text{MAX} \qquad \longrightarrow s := \text{ACC} \\
\qquad\qquad\qquad\qquad\quad\ \ [] \ \neg b \wedge s \neq \text{STOPPED} \longrightarrow s := \text{DEC} \\
\qquad\qquad\qquad\qquad\quad\ \ ] \\
\quad [] \ s = \text{ACC}; \nabla v = v_{\text{set}} \longrightarrow s := \text{MAX} \\
\quad [] \ s = \text{DEC}; \nabla v = 0 \quad \longrightarrow s := \text{STOPPED} \\
\quad] \\
]\!]
\end{array}
$$

5.2 Conveyors with colliding boxes

In this section a more accurate model of the conveyors is presented. This model takes into account the possibility of colliding boxes. If a box enters a conveyor section that contains a box which is not moving, the entering box collides with the box already present. It is assumed, that as a result of a collision the velocity of the entering conveyor becomes equal to the minimum of (1) the velocity of the conveyor section by which it is supported and (2) the velocity of the box in front of it.

Only process *CBC* (Conveyor Back Colliding) is specified. The specification of process *CFC* is similar. The specification of the process follows below. Figure 4 shows the process *CBC* together with the surrounding processes *CFC*.

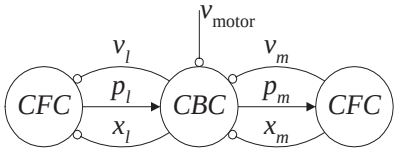

Fig. 4. Model of the conveyors with colliding boxes.

$$
\begin{array}{l}
\textsf{proc}\ \mathit{CBC}(\ v_{\text{motor}}, v_l, v_m : \ \multimap \text{vel},\ x_l, x_m : \ \multimap \text{pos} \\
\qquad\qquad , \ p_l : ?\,\text{id},\ p_m : !\,\text{id} \\
\qquad\qquad , \ l_{\text{b}}, l_{\text{c}} : \text{real} \\
\qquad\qquad) = \\
[\![\ x, x_{\text{next}} : \text{pos},\ v, v_{\text{next}} : \text{vel} \\
,\ \mathit{id} : \text{id},\ s : \{\text{EMPTY, FRONT, TAIL, FRONTTAIL, COLLISION}\} \\
;\ x ::= \bot ;\ s := \text{EMPTY} \\
\mid\ v_{\text{motor}} \multimap v \\
,\ x_l \qquad \multimap x \\
,\ v_l \qquad \multimap v \\
,\ x_m \qquad \multimap x_{\text{next}} \\
,\ v_m \qquad \multimap v_{\text{next}}
\end{array}
$$

$$
\begin{array}{l}
\mid [\ s \neq \text{COLLISION} \longrightarrow x' = v \\
\quad [] s = \text{COLLISION} \longrightarrow x' = \min(v, v_{\text{next}}) \\
\quad] \\
\mid *[\ s = \text{EMPTY} \quad ; p_l ? id \quad \longrightarrow x ::= 0 ; s := \text{FRONT} \\
\quad [] s = \text{FRONT} \quad ; \nabla x = l_{\text{c}} - l_{\text{b}} \longrightarrow p_m ! id ; x ::= \perp ; s := \text{TAIL} \\
\quad [] s = \text{TAIL} \quad ; \nabla x_{\text{next}} = l_{\text{b}} \longrightarrow s := \text{EMPTY} \\
\quad [] s = \text{TAIL} \quad ; p_l ? id \quad \longrightarrow x ::= 0 ; s := \text{FRONTTAIL} \\
\quad [] s = \text{FRONTTAIL} ; \nabla dist = 0 \quad \longrightarrow s := \text{COLLISION} \\
\quad [] s = \text{FRONTTAIL} ; \nabla x_{\text{next}} = l_{\text{b}} \longrightarrow s := \text{FRONT} \\
\quad [] s = \text{COLLISION} ; \nabla x_{\text{next}} = l_{\text{b}} \longrightarrow p_m ! id ; x ::= \perp ; s := \text{TAIL} \\
\quad [] s = \text{COLLISION} ; \nabla dist > 0 \quad \longrightarrow s := \text{FRONTTAIL} \\
\quad] \\
]\!]
\end{array}
$$

The variable s stores the state of the process, it can take on any of the values in the set {EMPTY, FRONT, TAIL, FRONTTAIL, COLLISION}. The different states are also shown in Figure 5.

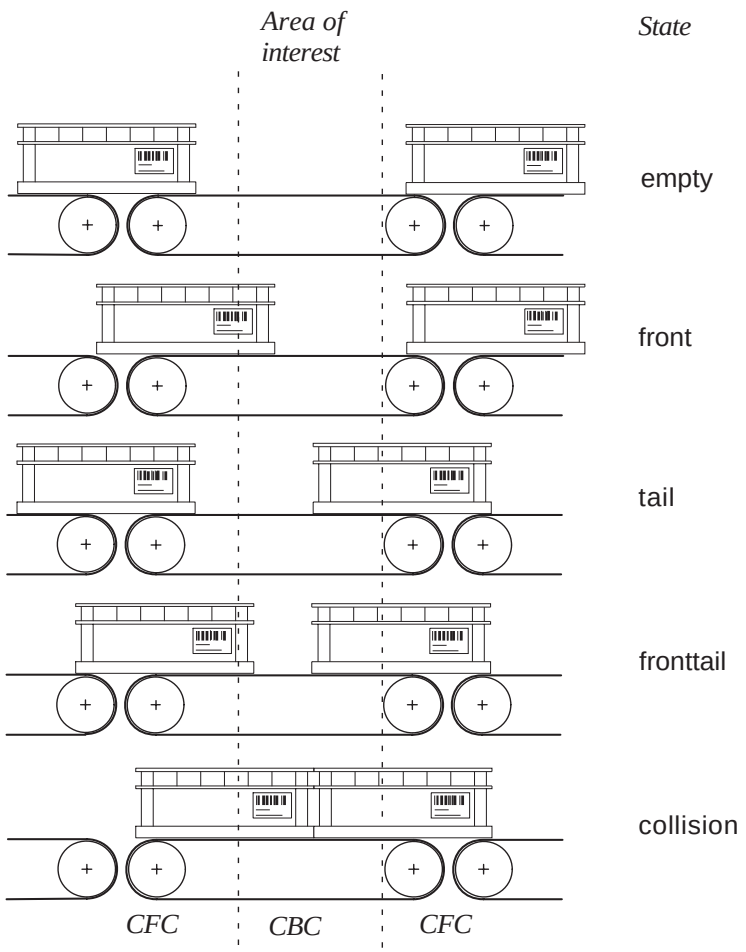

Fig. 5. The states of process *CBC*.

The state depends on the position of the boxes within the process boundaries. Initially

the state is EMPTY. If a new box enters the process, the state changes to FRONT. The box moves across the conveyor until it reaches the end of section *CBC*. From this point on, the position of the box is described by the next process, but its tail remains in section *CBC*. The state now changes to TAIL. Two events can now take place: the tail of the box can leave the section ($\nabla\, x_{\text{next}} = l_{\text{b}}$) thus changing the state to EMPTY, or a new box can enter the section ($p_l\,?\,id$) thus changing the state to FRONTTAIL. When the state equals FRONTTAIL, the two boxes collide when the distance between them becomes zero. The state then changes to COLLISION. As a result, the velocity of the colliding box becomes equal to the minimum of its own velocity and the velocity of the box in front of it ($x' = \min(v, v_{\text{next}})$). In the case of a collision, the two conveyors can either cross the process boundary simultaneously ($\nabla\, x_{\text{next}} = l_{\text{b}}$), or the first box can start moving faster than the second box, causing the boxes to be separated again ($\nabla\, dist > 0$). The expression *dist* has been introduced for reasons of clarity. It denotes the distance between the front of a box and the tail of its successor (if present). This distance equals $(l_{\text{c}} - l_{\text{b}}) - x - (l_{\text{b}} - x_{\text{next}})$ (see Figure 6), so instead of *dist* one should read $x_{\text{next}} - x - 2l_{\text{b}} + l_{\text{c}}$. The position ($x_{\text{next}}$) and velocity ($v_{\text{next}}$) of the succeeding box are made available to process *CBC* by means of the two continuous channels x_m and v_m (see Figure 4).

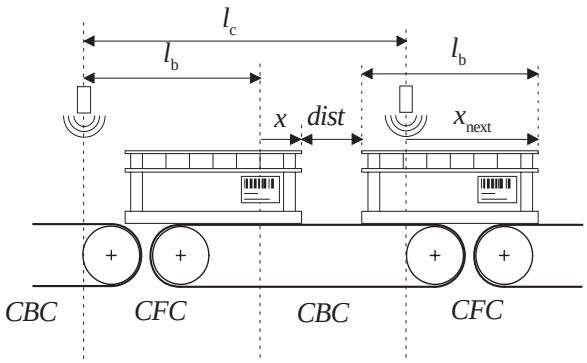

Fig. 6. Conveyors with colliding boxes.

6 Conclusions

The application of the χ language to modelling and simulation of manufacturing machines has been illustrated using a conveyor based transport system. The language is relatively easy to use and to learn because of the small number of orthogonal language constructs. Combined with the symbolic mathematical notation, this leads to concise models. Good modularity is achieved by the use of processes and systems and the absence of global variables.

The integration of continuous-time and discrete-event concepts in the language makes it highly suitable for machine modelling. In the continuous-time parts of the models, positions of moving parts are described by simple DAEs. In order to simplify models, many actions of machines can be abstracted as discrete-event actions. For

this purpose, the χ language provides CSP-based discrete-event constructs. Changes in the velocity of moving parts can be simplified as discontinuous changes, which are modelled in the discrete-event part of the processes. The crossing of process boundaries by products is modelled by synchronous communications. Interaction between the continuous-time and discrete-event parts of processes is achieved by means of assignments to variables occurring in DAEs, or by means of the nabla operator in order to synchronize with state events.

The χ simulator for the discrete-event part of the language [21] is being used in a large number of cases. The cases deal with modelling and simulation of complex discrete-event manufacturing systems, such as production facilities for integrated circuits. The first version of the χ simulator for combined continuous-time / discrete-event systems [12] has recently been completed.

The χ language can also be used for the specification of control systems, including exception handling [6,7]. Such a control model can be tested by connecting it to a model of the controlled machines. The sensors and actuators serve as the interface between the controller and the controlled system. The combined system can then be simulated. In this way, the combined continuous-time / discrete-event approach to modelling and simulation of manufacturing machines can be an important aid for the development of robust machine control systems.

Acknowledgement

We like to thank Ralph Walenbergh for his contribution to the control system specification. Thanks are also due to the reviewers for their helpful comments on the draft of this paper.

References

[1] M. Andersson, Combined object-oriented modelling in Omola, in: *Proceedings of the 1992 European Simulation Multiconference*, York (1992) 246–250.

[2] N.W.A. Arends, A systems engineering specification formalism, Ph.D. Thesis, Eindhoven University of Technology, The Netherlands, 1996.

[3] Aspen Technology, Cambridge, Speedup user guide, 1991.

[4] J. le Bail, H. Alla, and R. David, Asymptotic continuous Petri Nets: An efficient approximation of discrete event systems, in: *IEEE International Conference on Robotics and Automation*, Nice (1992) 1050–1056.

[5] P.I. Barton, The modelling and simulation of combined discrete/continuous processes, Ph.D. Thesis, University of London, 1992.

[6] D.A. van Beek, Exception handling in control systems, Ph.D. Thesis, Eindhoven University of Technology, The Netherlands, 1993.

[7] D.A. van Beek and J.E. Rooda, A new mechanism for exception handling in concurrent control systems, *European Journal of Control* (2) (1996) 88–100.

[8] D.A. van Beek, J.E. Rooda, and S.H.F. Gordijn, A combined continuous-time / discrete-event approach to modelling and simulation of manufacturing machines, in: *Proceedings of the 1995 EUROSIM Conference*, Vienna (1995) 1029–1034.

[9] R. David and H. Alla, Petri Nets for modeling of dynamic systems–a survey, *Automatica* **30** (2) (1994) 175–202.

[10] Dept. of Mathematics Delft University of Technology and Computer Science, Prosim reference manual, Sierenberg en de Gans, Waddinxveen, The Netherlands, 1995.

[11] H. Elmqvist, Dymola–dynamic modeling language–user's manual, Dynasim AB, Lund, Sweden, 1994.

[12] G. Fabian and P. Janson, Simulator for combined continuous-time / discrete-event models, Final report of the Postgraduate Programme Software Technology, Eindhoven University of Technology, Stan Ackermans Institute, The Netherlands, 1996.

[13] S.H.F. Gordijn, Combined continuous-time / discrete-event specifications of industrial systems using the language χ, Master's Thesis, Eindhoven University of Technology, Department of Mechanical Engineering, The Netherlands, 1995.

[14] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice-Hall, Englewood-Cliffs, 1985).

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

[16] B.W. Kernighan and D.M. Ritchie, *The C Programming Language* (ANSI Standard C) (Prentice-Hall, Englewood Cliffs, 1988).

[17] D.L. Kettenis, Issues of parallelization in implementation of the combined simulation language COSMOS, Ph.D. Thesis, Delft University of Technology, 1994.

[18] E.E.L. Mitchell and J.S. Gauthier, Advanced continuous simulation language (ACSL), *Simulation* **26** (3) (1976) 72–78.

[19] J.M. van de Mortel-Fronczak and J.E. Rooda, Application of concurrent programming to specification of industrial systems, in: *Proceedings of the 1995 IFAC Symposium on Information Control Problems in Manufacturing*, Bejing (1995) 421–426.

[20] J.M. van de Mortel-Fronczak, J.E. Rooda, and N.J.M van den Nieuwelaar, Specification of a flexible manufacturing system using concurrent programming, *Concurrent Engineering: Research and Applications* **3** (3) (1995) 187–194.

[21] G. Naumoski and W.T.M Alberts, The χ engine: a fast simulator for systems engineering, Final report of the Postgraduate Programme Software Technology, Eindhoven University of Technology, Stan Ackermans Institute, The Netherlands, 1995.

[22] C.D. Pegden, R.E. Shannon, and R.P. Sadowski, *Introduction to Simulation Using SIMAN* (McGraw-Hill, 1995).

[23] N. Zerhouni and H. Alla, Dynamic analysis of manufacturing systems using continuous Petri Nets, in: *IEEE International Conference on Robotics and Automation*, Cincinnati (1990) 1070–1075.