In *CESA'96 IMACS Multiconference, Symposium on Discrete Events and Manufacturing Systems*, Lille, July 1996, pp 596–601.

Hybrid Modelling in Discrete-Event Control System Design

D.A. van Beek, J.E. Rooda, and S.H.F. Gordijn
Eindhoven University of Technology
Department of Mechanical Engineering
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
E-mail: vanbeek@wtb.tue.nl

Abstract

Simulation-based testing of discrete-event control systems can be advantageous. There is, however, a considerable difference between languages for real-time control and simulation languages. The χ language, presented in this paper, is suited to specification and simulation of real-time control systems. The hybrid nature of the language makes it also suited to modelling of controlled machines. By connecting such models to models of control systems, simulation-based testing is possible. The language integrates a small number of 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. A case study is presented of a conveyor control system. The example illustrates the suitability of the language for discrete-event control system specification as well as for simulation-based testing of control systems.

1 Introduction

Errors in control systems can have severe consequences. Therefore it is important that control systems are thoroughly tested. Testing a control system by means of simulation can have significant advantages over using real machines [1]: using real machines may be hazardous because of the possibility of damage caused by errors, or such real machines may still be under development. However, simulation-based testing of discrete-event control systems can be difficult, because there are considerable differences between real-time control languages and simulation languages.

The χ language aims to reduce the gap between simulation and control of industrial systems. It is suited to specification and simulation of manufacturing systems, as well as to specification and simulation of real-time control systems. These control systems can be of a discrete-event, discrete-time or continuous-time nature.

Another obstacle in using simulation models for testing discrete-event control systems is the use of a discrete-event language for modelling of the controlled machines. A discrete-event language is impractical because the model of the controlled machines must respond correctly to many different commands from the control system. In this paper, the controlled machines are modelled using a combined continuous-time / discrete-event specification in χ, which results in concise and accurate models.

Many continuous-time languages have been extended with discontinuity handling using time-events or state-events. An example of such a language is Dymola [2]. Such languages are, however, not suited to modelling of discrete-event control systems. Other hybrid modelling languages, such as SIMAN [3], are based on discrete-event concepts, so that equations can only be represented by assignment based constructs. The main difference between the hybrid language gPROMS [4] and χ is that the discrete-event part of χ is based on concurrent programming, whereas the discrete-event part of gPROMS is based on composition of sequential and parallel blocks.

2 The χ Language

The 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. Its symbolic notation makes the specifications easy to read and to develop. For simulation purposes the symbols are replaced by their ASCII equivalents.

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 [5].

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 processes is identical to that of systems. A process may consist of a continuous-time part only (links and DAEs), a discrete-event part only, or a combination of both.

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. A (discrete) synchronization channel is symmetrical; all other channels are used for output in one process and input in the other. Channels are declared in systems and are either discrete (e.g. p : int, s : void) or continuous (e.g. q : [m/s]). Channels are also declared as process or system parameters in which case the usage of the channel is declared as either output (e.g. p : ! int or q : ↑[m/s]), input (e.g. p : ? int or q : ↓[m/s]), or synchronization (s : ~ void). A continuous channel is represented graphically by a line ending in a small circle, a discrete communication channel by an arrow, a discrete synchronization channel by a line; processes and systems are represented by circles.

Data types and variables

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. A real discrete data type can be postfixed with a unit of measurement. For example, the declaration v : real[m/s] defines a discrete variable (or parameter) v of type real with unit m/s (a velocity). Since all continuous variables are assumed to be of type real, they are defined by specifying their units only. For example, the declaration v : [m/s] defines a continuous variable or channel v. Continuous channels are specified likewise in parameter declarations, e.g. v : ↑[m/s] defines a continuous channel v which may be linked to a continuous variable of type [m/s].

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').

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:

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

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.

A continuous channel relates a variable of one process to a variable of another process by means of an equality relation. *Links* are used to associate a continuous channel with a continuous variable:

$$channel \multimap var.$$

The variable and the channel must be of the same (continuous) data type. Consider two variables x_a, x_b in different processes 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.

The discrete-event part of χ

The discrete-event part of χ is a CSP-like [6] real-time concurrent programming language, described in [7] and [8]. Our notation is derived from [9]. We have adopted the CSP-like constructs because of: (1) the concurrent CSP-processes, that are well suited to modelling the parallelism found in industrial systems; (2) the sound mathematical background of the formalism; (3) the synchronous nature of the interactions, which means that buffers are modelled explicitly and cannot be hidden in interactions; (4) the lack of global variables, which enables robust modelling; and (5) the possibility to extend the CSP language constructs with the powerful selective waiting construct, which is explained below together with the other language constructs.

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

$$c!e, \quad c?x \quad \text{or} \quad c^{\sim}.$$

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 $c^{\sim}$. Execution of $c^{\sim}$ in one process causes the process to be blocked until $c^{\sim}$ is executed in the other process.

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

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

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 least one of the guards must be open. 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

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

An event statement E_i which is prefixed by a guard b_i ($b_i ;\ E_1$) 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 five types of event statement: input ($c?e$), output ($c!x$), synchronization ($c\,\tilde{}$), time ($\Delta\, t$), and state ($\nabla\, rel$, 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). 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 $*[$ true $\longrightarrow S\,]$ may be abbreviated to $*[\, S\,]$.

Interaction between the continuous and discrete parts of χ

The interaction between the continuous and discrete parts of χ is illustrated by means of a small example. Consider a tank which is filled with a flow F_{in}. The drain can be switched on and off. If it is on (*drain* equals true), the flow out of the tank equals $k\sqrt{h}$, where h is the level of the liquid in the tank. The surface of the tank is A.

```
proc Tank
|[ h : [m]
, drain : bool, F_in : real
; h ::= 0; drain := false; F_in := 10^-3
| [ ¬drain ⟶ Ah' = F_in
  []  drain ⟶ Ah' = F_in − k√h
  ]
| ∇ h = 0.2; drain := true
; Δ 2; F_in := 2 · 10^-3; Δ 100
]|
```

There is one continuous variable h. The other variables *drain* and F_{in} are discrete. There are two constants: A and k. Initially the height equals 0 ($h ::= 0$) and the drain is off. The discrete-event part of the process first synchronizes with the continuous part until h equals 2 ($\nabla\, h = 0.2$). Subsequently the drain is switched on. Then, after 2 units of time, the outgoing flow is set to $2 \cdot 10^{-3}$.

The interaction between the continuous and discrete parts of χ is now described in more general terms. In the discrete-event part of a process, assignments can be made to discrete variables occurring in DAEs ($F_{in} := 2 \cdot 10^{-3}$) or in the boolean guards of guarded DAEs (*drain* := true). In the first 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 ($h ::= 0$), and may be reinitialized in the discrete-event part (see the next section for an example). In both cases the symbol ::= is used.

By means of the *state event* statement

$$\nabla\, rel,$$

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

3 A Conveyor Line Control System

The example treats a line of three conveyors for the transportation and buffering of boxes (see Figure 1). The

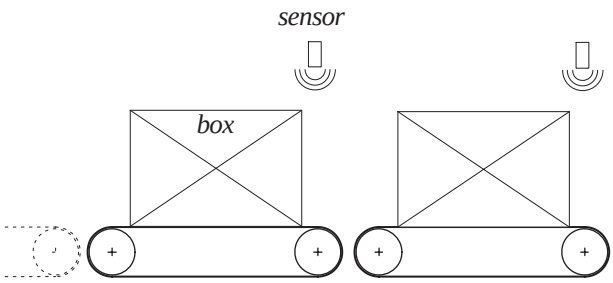

Figure 1: The conveyors.

machines that supply the boxes to the first conveyor and

remove the boxes from the last conveyor are not considered. Each conveyor has its own motor and a sensor for the detection of a box. It is assumed that the movement of a box is determined by the conveyor which supports the major part of the box, and that the control system ensures that the boxes do not collide. Initially the conveyors are empty.

The graphical representation of the conveyor line control system is shown in Figure 2. The textual specification

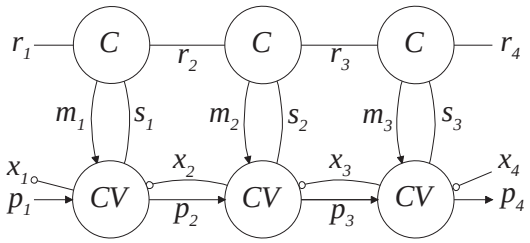

Figure 2: Graphical representation of system *CVC*.

is shown below. The three processes C each control a single conveyor CV.

$$
\begin{array}{l}
\textsf{syst } CVC(\ x_1 : \uparrow \text{pos},\ x_4 : \downarrow \text{pos} \\
\qquad ,\ r_1, r_4 : {}^{\sim}\text{void},\ p_1 : ?\,\text{id},\ p_4 : !\,\text{id} \\
\qquad ,\ l_c, l_b, l_s : \text{length},\ v : \text{vel} \\
\qquad) = \\
[\![\ x_2, x_3 : \text{pos} \\
,\ r_2, r_3 : \text{void},\ p_2, p_3 : \text{id},\ m_1, m_2, m_3 : \text{vel} \\
,\ s_1, s_2, s_3 : \text{sensor} \\
|\ C(r_1, r_2, m_1, s_1) \\
\|\ C(r_2, r_3, m_2, s_2) \\
\|\ C(r_3, r_4, m_3, s_3) \\
\|\ CV(x_1, x_2, p_1, p_2, m_1, s_1, l_c, l_b, l_s, v) \\
\|\ CV(x_2, x_3, p_2, p_3, m_2, s_2, l_c, l_b, l_s, v) \\
\|\ CV(x_3, x_4, p_3, p_4, m_3, s_3, l_c, l_b, l_s, v) \\
]\!]
\end{array}
$$

The types are defined below. The only continuous type is pos. The type sensor is a record of two synchronization channels named *on* and *off*. The parameter declaration $s : {}^{\sim}\text{sensor}$ in a process, defines the two synchronization channels: $s.on$ and $s.off$.

$$
\begin{array}{lll}
\textsf{type} & \text{pos} & = [\text{m}] \\
, & \text{vel} & = \text{real}[\text{m} \cdot \text{s}^{-1}]\ ,\ \text{length} = \text{real}[\text{m}] \\
, & \text{id} & = \text{int} \\
, & \text{sensor} & = (on : \text{void} \times off : \text{void})
\end{array}
$$

The model of a conveyor

In Figure 3 some of the variables and parameters used in the specification of the processes *CV* are shown. The conveyor is divided into two equal areas, each with length $\frac{1}{2}l_c$. The continuous variables x_1 and x_2 represent the positions of the midpoints of a box. If the midpoint of a box lies in the first half of the conveyor it is described by x_1. If it lies in the second half it is described by x_2 ($0 \le x_1 \le \frac{1}{2}l_c$, $\frac{1}{2}l_c \le x_2 \le l_c$). The parameters l_c, l_b, and l_s represent the length of a conveyer, the length of a box and the position of the sensor, respectively.

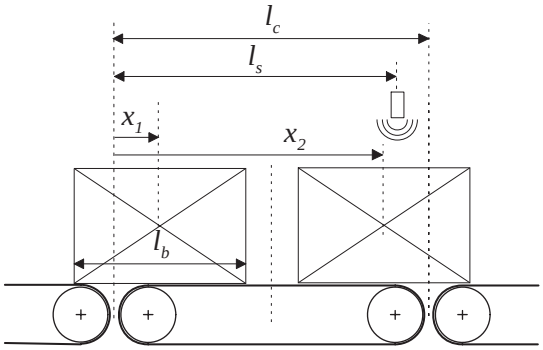

Figure 3: Variables and parameters of process *CV*.

The specification of process *CV* follows below. Each box is associated with a number which is stored in the variable id_1 or id_2. The parameter v represents the velocity of the conveyor-belt. The boolean variables b_1, b_2 and b_3 represent the presence of a box on the first part, second part, or first part of the next conveyor, respectively. The booleans b_s and *on* represent the state of the sensor and the motor, respectively.

$$
\begin{array}{l}
\textsf{proc } CV(\ x_i : \uparrow \text{pos},\ x_j : \downarrow \text{pos} \\
\qquad ,\ p_{in} : ?\,id,\ p_{out} : !\,id,\ m\,?\,\text{bool},\ s : {}^{\sim}\text{sensor} \\
\qquad ,\ l_c, l_b, l_s : \text{length},\ v : \text{vel} \\
\qquad) = \\
[\![\ x_1, x_2, x_3 : \text{pos} \\
,\ on : \text{bool},\ b_1, b_2, b_3, b_s : \text{bool},\ id_1, id_2 : \text{id} \\
;\ x_1 ::= \bot;\ x_2 ::= \bot;\ on := \text{false} \\
|\ x_i \multimap x_1 \\
,\ x_j \multimap x_3 \\
|\ [\quad on \longrightarrow x_1' = v,\ x_2' = v \\
\quad []\ \neg on \longrightarrow x_1' = 0,\ x_2' = 0 \\
\quad] \\
|\ b_1 := \text{false};\ b_2 := \text{false};\ b_3 := \text{false};\ b_s := \text{false} \\
;\ *[\ \neg b_1;\ p_{in}\,?\,id_1 \quad \longrightarrow b_1 := \text{true};\ x_1 ::= 0 \\
\quad []\quad b_1;\ \nabla\, x_1 \ge \tfrac{1}{2}l_c \longrightarrow b_1 := \text{false};\ b_2 := \text{true} \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ id_2 := id_1 \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ x_2 ::= x_1;\ x_1 ::= \bot \\
\quad []\quad b_2;\ \nabla\, x_2 \ge l_c \quad \longrightarrow b_2 := \text{false};\ b_3 := \text{true} \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ p_{out}\,!\,id_2 \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ x_2 ::= \bot \\
\quad []\qquad m\,?\,on \qquad \longrightarrow \text{skip} \\
\quad []\ b_1 \wedge \neg b_s;\ \nabla\, x_1 + \tfrac{1}{2}l_b \ge l_s \\
\qquad\qquad\qquad\qquad\qquad\quad \longrightarrow b_s := \text{true} \\
\quad []\ b_2 \wedge \neg b_s;\ \nabla\, x_2 + \tfrac{1}{2}l_b \ge l_s \\
\qquad\qquad\qquad\qquad\qquad\quad \longrightarrow b_s := \text{true} \\
\quad []\quad b_3;\ \nabla\,(l_c - l_s) + x_3 \ge \tfrac{1}{2}l_b \\
\qquad\qquad\qquad\qquad\qquad\quad \longrightarrow b_s := \text{false};\ b_3 := \text{false} \\
\quad []\quad b_s;\ s.on^{\sim} \qquad \longrightarrow \text{skip} \\
\quad []\ \neg b_s;\ s.off^{\sim} \qquad \longrightarrow \text{skip} \\
\quad] \\
]\!]
\end{array}
$$

The control system ensures that boxes do not collide. This is done by keeping the distance between the midpoints of two consecutive boxes at approximately l_c; a 'new' box is only allowed to enter when the 'old' box is leaving a conveyor. This keeps the process *CV* relatively simple.

A conditional equation determines the positions x_1 and x_2. If the motor is running (*on* is true), the equations $x_1' = v,\ x_2' = v$ are selected, whereas when the motor is switched off, the equations $x_1' = 0,\ x_2' = 0$ are selected.

All variables used in the continuous part are initialized immediately after the declarations ($x_1 ::= \perp$; $x_2 ::= \perp$; $on :=$ false). Initially the conveyors are empty. Therefore the variables x_1 and x_2 are both initialized to $\perp$. In this way they become undefined. When a continuous variable is initialized to $\perp$, the variable and all equations in which it occurs will be temporarily removed from the system, until the variable is initialized to a value other than $\perp$.

To explain the structure of the selective-waiting statement we will follow the path of a box. First b_1 and b_2 are false, so the process waits until it either receives a new box ($p_{in}?id_1$) or the new state of the motor ($m?on$). When a new box is received, x_1 is initialized to 0. If the motor is on, the box will move towards the end of the conveyor. Subsequently, two events can take place. Either the sensor is activated, or the midpoint of the box leaves the first part of the conveyor and enters the second part. The length of the box determines which event takes place first. The former event is modelled by $b_1 \wedge \neg b_s\,;\ \nabla\, x_1 + \frac{1}{2}l_b \geq l_s$, followed by execution of $b_s :=$ true. The latter event is modelled by $b_1\,;\ \nabla\, x_1 \geq \frac{1}{2}l_c$. If this event occurs, x_2 is initialized to the value of x_1, and x_1 becomes undefined ($x_1 ::= \perp$). After this, the next event depends on the state of the sensor. If the sensor is not yet on, it will be switched on in the next event ($b_2 \wedge \neg b_s\,;\ \nabla\, x_2 + \frac{1}{2}l_b \geq l_s$). If it is already on, the next event will be the departure of the box to the next system, which is modelled by $\cdots\, p_{out}\,!\,id_2 \cdots$. The deactivation of the sensor requires an extra channel x_j, because the position of the box is then determined by the next conveyor process. This channel makes the variable x_3 equal to the variable x_1 of the next conveyor process *CV*. The sensor is switched off when $(l_c - l_s) + x_3 \geq \frac{1}{2}l_b$.

The sensor is modelled by means of the two channels $s.on$ and $s.off$. These are used in the control system to wait for the sensor to be activated or deactivated, respectively. The synchronization $s.on\,\tilde{}$ can only take place if the sensor is activated. This is modelled by $b_s\,;\ s.on\,\tilde{}\ \longrightarrow$ skip in the process *CV*, where skip is the empty statement. Execution of the statement $s.off\,\tilde{}$ in the control system causes it to block until the sensor is deactivated.

The Control System

The control system ensures that boxes do not collide, that a conveyor is not switched off unnecessarily when a box reaches the end of the conveyor and can immediately proceed onto the next one, and that a conveyor is switched off when it remains empty for a certain amount of time. The interface between controlling systems and the controlled components is formed by actuators and sensors. In this case, the actuator is modelled by channel m, and the sensor by channel s. The motor is switched on and off by execution of $m\,!\,$true and $m\,!\,$false, respectively.

```
proc C(r_in, r_out : ~ void, m : ! bool, s : ~ sensor) =
|[ r_in ~ ; m ! true ; s.on ~
; *[ [ r_out ~ --> skip
     [] Δ 0.2 --> m ! false ; r_out ~ ; m ! true
     ]
   ; [ r_in ~     --> s.off ~ ; s.on ~
     [] s.off ~ --> [ r_in ~ --> skip
                    [] Δ 5   --> m ! false ; r_in ~ ; m ! true
                    ]
                  ; s.on ~
     ]
   ]
]|
```

Initially a new box is received, the motor is switched on and the process *C* waits until the sensor is activated. After this the repetition is executed. It consists of a sequence of two selective waiting statements.

At the beginning of the first selective waiting statement the motor is running and the front of a box has just reached the sensor. If the next conveyor can receive the box ($r_{out}\,\tilde{}$) within 0.2 seconds (Δ 0.2), the first selective waiting statement is terminated and the second one is executed. If the box cannot be received within 0.2 seconds, the motor is switched off and the process waits until the next conveyor is ready to receive the box.

At the beginning of the second selective waiting statement, the motor is running and the box has just started leaving the conveyor. Now one of two events may occur first. Either a second box may enter the conveyor ($r_{in}\,\tilde{}$), or the box leaving the conveyor may clear the detector ($s.off\,\tilde{}$) before a second box has entered. In the former situation, the control process subsequently waits for the first box to have left the conveyor ($s.off\,\tilde{}$), and then waits for the second box to reach the detector ($s.on\,\tilde{}$). In the latter situation, the control process subsequently tries to receive a new box ($r_{in}\,\tilde{}$). If this is not possible within 5 seconds, the motor is switched off and the process waits until a new box can be received. At the end of the second selective waiting statement the motor is running and the front of a box has just reached the sensor. Therefore the repetition can then be re-executed.

4 Concluding Remarks

The application of the χ language for discrete-event control system design has been illustrated using a simple conveyor based transport system. The approach outlined in this paper uses machine models connected to the control system model. Our aim is to use the simulation model of the control system also for actual real-time control by

simply removing the machine models, and connecting the interface channels to the actual controlled machines. Research in this direction is continuing.

The χ language is not limited to discrete-event control system design. It can also be used for the specification and simulation of discrete-time and continuous-time control systems [10]. Applications range from low-level PLC sequence control to high-level supervisory control, including scheduling and planning (e.g. see [11]).

The example has illustrated the suitability of the language for the specification of discrete-event control systems. The language is flexible, and consists of a small number of orthogonal constructs. This results in concise and clear specifications. The integration of continuous-time and discrete-event concepts in the language makes it highly suitable for machine modelling.

A χ simulator for the discrete-event part of the language has been developed [12]. It is being used for modelling and simulation of complex discrete-event manufacturing systems, such as production facilities for integrated circuits. The χ simulator for combined continuous-time / discrete-event systems is still under construction.

References

[1] D.A. van Beek, *Exception Handling in Control Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1993.

[2] H. Elmqvist, *Dymola–Dynamic Modeling Language–User's Manual*. Dynasim AB, Lund, Sweden, 1994.

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

[4] P.I. Barton, *The Modelling and Simulation of Combined Discrete/Continuous Processes*. PhD thesis, University of London, 1992.

[5] N.W.A. Arends, *A Systems Engineering Specification Formalism*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1996.

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

[7] 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), pp. 421–426, Oct. 1995.

[8] 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*, vol. 3, no. 3, pp. 187–194, 1995.

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

[10] P.A.B.F. van Geldrop, "The specification and analysis of discrete-time systems in χ," MSc thesis WPA 420082, Eindhoven University of Technology, The Netherlands, 1996.

[11] D.A. van Beek, S.H.F. Gordijn, and J.E. Rooda, "Integrating continuous-time and discrete-event concepts in process modelling, simulation and control," in *Proceedings of the First World Conference on Integrated Design and Process Technology*, pp. 197–204, Dec. 1995.

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