In *Proc. of the 15th. IMACS World Congress vol. 2 Numerical Mathematics*, 1997, pp 721-726.

Specification and Simulation of Industrial Systems Using an Executable Mathematical Specification Language

D.A. van Beek, J.E. Rooda
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
E-mail: vanbeek@wtb.tue.nl
URL: http//asterix.urc.tue.nl/~vanbeek

Keywords: symbolic specification, industrial systems, control, simulation, hybrid.

ABSTRACT

Most present day simulation languages use ASCII characters for representation of models. ASCII characters are chosen, because of the requirements of present day computer implementations. In mathematics, on the other hand, symbolic representations are common place, because of better readability. In this article, the χ specification language is presented. The language is used for specification, simulation and control of industrial systems. It uses a symbolic notation that can be easily translated to an ASCII representation for simulation purposes. The χ language is hybrid: its continuous-time part is based on differential algebraic equations; its discrete-event part is based on a CSP-like concurrent programming language. The language constructs have been chosen in such a way, that they resemble mathematical notations. Future work includes the development of an automatic Chi-ASCII to χ-Latex translator.

INTRODUCTION

Current simulation packages are generally based on ASCII languages using keywords. This is due to the fact that the compilers or interpreters need ASCII input. Also, using these packages emphasis is on *simulation* of the models in order to get numerical or graphical output. In mathematics on the other hand a symbolic notation is used, because such a notation is easier to read, write and understand. The χ language also uses a symbolic notation. Emphasis is on the correct *specification* of the dynamic behaviour of industrial systems, in order to gain insight and understanding.

A second aspect in which the χ language differs from generally used simulation languages is the choice of language constructs. Also in this respect we have adopted constructs that are related to mathematical notations. In this paper the χ language constructs are illustrated by some small examples, and are compared with the language constructs that are commonly used in present-day simulation languages. Only part of the χ language is treated.

SOME ARGUMENTS FOR A MATHEMATICAL NOTATION

The χ language has had several predecessors. The first of these was Sole [1]: a language for modelling industrial systems, based on Simula67 [2]. This language was refined into the S84 [3] language based on the Modular Pascal [4] parallel programming language. The successor of S84 was the object oriented interactive modelling environment ProcessTool [5], based on the Smalltalk-80 [6] object oriented programming language. So far, all languages had been based on existing programming languages. The χ language is different in this respect, because it is not based on a programming language. The

reason for this change is that industrial systems are becoming increasingly complex, whereas the time available for (re)design is more and more limited. This makes it increasingly difficult to model the dynamic behaviour of such plants in a short time. The aim of the χ language is to support the development of correct models of complex systems in a limited amount of time. For this purpose, it is based on a small number of orthogonal language constructs that are related to mathematical notations. Using these language constructs enables the user to create elegant specifications, that are easy to read, write and understand. Compare for example the well known mathematical notation of

$$h' = k\sqrt{p_i - p_o}$$

with its ASCII equivalent

```
h' = k * sqrt(pi - po)
```

The mathematical notation is much easier to read and understand. Now compare the symbolic notation of

$$p\,?\,x\,;\,\Delta\,t\,;\,q\,!\,5$$

with a notation using keywords

```
p receive x; waitduring t; q send 5
```

For readers unfamiliar with the symbols ?, ! and Δ, the notation with keywords is clearer. For readers familiar with the meaning of the symbols, the symbolic notation is easier to read, write and understand. Since the χ language is being used for the specification and dynamical analysis of *complex* systems, the time to learn the mathematical notation pays off with better readable models.

Conversion of symbols to ASCII equivalents

Since symbols cannot be used for compilation and simulation of a χ model, they need to be converted to ASCII equivalents. This conversion is a straightforward process by which each symbol is converted to either an ASCII symbol or an ASCII keyword. Table 1 shows the most commonly used χ symbols and their ASCII equivalents together with the pronunciation.

AN INTRODUCTION TO THE χ LANGUAGE

The χ language is used for specification, simulation and control of industrial systems. It is suited to specification of discrete-event systems, continuous-time systems as well as combined discrete / continuous (hybrid) systems. Where possible, the continuous-time and discrete-event parts of the language are based on similar concepts. In this paper a subset of the language is treated. We do not treat the language elements for modelling parallel processes that interact by means of message passing and synchronization, nor do we treat the use of systems for hierarchical modelling. For these and other aspects of the χ language we refer to [7, 8, 9]. The syntax and operational semantics of the language constructs are explained in an informal way.

A process may consist of a continuous-time part only (DAEs: differential algebraic equations), a discrete-event part only, or a combination of both.

proc *name*(*parameter declarations*) =
‖[*variable declarations* ; *initialization*
| *DAEs* | *discrete-event statements*
]‖

χ notation	Chi notation	Pronunciation
proc	`proc`	process
syst	`syst`	system
func	`func`	function
$[\![$	`|[`	begin block
$]\!]$	`]|`	end block
$\|\|$	`||`	parallel with
$\|$	`|`	separator
$*$	`*`	repeat
[	`[`	begin ...
[]	`|`	or
]	`]`	end
$\longrightarrow$	`->`	then
Δ	`delta`	delta
$\uparrow$	`ret`	return
τ	`time`	current time
x'	`x'`	derivative of x
$a \leq x \leq b$	`a<=x and x<=b`	

Table 1: *Conversion from Chi to* χ.

The discrete-event part of χ is a CSP-like [10] real-time concurrent programming language for which we refer to [11] and [12]. Here, we mention *time passing*, which 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.

GUARDED COMMANDS

The χ language borrows several concepts from Dijkstra's guarded command language [13]. The χ *selection* statement is a guarded command

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

which is pronounced as: 'begin 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. Now consider the function

$$f(x) = \begin{cases} -1 & \text{if} \quad x < -1 \\ x & \text{if} \quad -1 \leq x \leq 1 \\ 1 & \text{if} \quad x > 1 \end{cases}$$

which is declared in χ as

$$\begin{array}{l} \text{func } f(x : \text{real}) \rightarrow \text{real} = \\ [\![\, [\; x < -1 \quad\quad \longrightarrow \uparrow -1 \\ \quad []\, -1 \leq x \leq 1 \longrightarrow \uparrow x \\ \quad []\, x > 1 \quad\quad \longrightarrow \uparrow 1 \\ \quad] \\ ]\!] \end{array}$$

The translation to the Chi ASCII representation leads to

```
func f(x: real)-> real =
|[ [ x<-1           -> ret -1
   | -1<=x and x<=1 -> ret x
   | x>1            -> ret 1
   ]
]|
```

Compare this with the commonly used keyword representation using if then else statements

```
function f(x: real): real
begin
  if x<-1 then
    return -1
  else
    if -1<=x and x<=1 then
      return x
    else
      return 1
    end
  end
end
```

The χ function definition closely resembles the mathematical notation, and for someone familiar with the χ symbols, it is better readable than the keyword representation which uses if then else statements.

The function $f(x)$ can also be specified as a *guarded equation*. Guarded equations are used when the set of equations depends on the state of the system. The syntax of a guarded equation statement is

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

which is pronounced as: 'begin guarded equations, if b_1 then $DAEs_1$, or ... or if b_n then $DAEs_n$, end'. Here $DAEs_i$ $(1 \leq i \leq n)$ represents one or more differential algebraic equations separated by commas: $DAE_1, DAE_2, \ldots, DAE_n$. 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. The specification of the function $y = f(x)$ using the continuous variables x and y becomes

$$\begin{array}{ll} [\; x > 1 & \longrightarrow y = 1 \\ []\; -1 \leq x \leq 1 & \longrightarrow y = x \\ []\; x < -1 & \longrightarrow y = -1 \\ \;] & \end{array}$$

The third type of guarded command is the *selective waiting* statement. For the explanation of this command we refer to [9, 14, 8].

AN EXAMPLE: THE SPECIFICATION OF A VALVE

The language elements treated above are now used for the specification of a valve. The control input of the valve is denoted by α_i. The actual position of the valve is denoted by α. It is assumed that the valve operates in its linear mode for $0 \leq \alpha_i \leq 1$ $(0 \leq \alpha_i \leq 1 \longrightarrow \alpha = \alpha_i)$. The pressures at the input and output of the valve are denoted by p_i and p_o, respectively. The variable Q represents the flow through the valve, and k is some valve constant. The equations describing the valve are

$$
\begin{array}{ll}
[\ \alpha_i < 0 & \longrightarrow \alpha = 0 \\
[]\ 0 \leq \alpha_i \leq 1 & \longrightarrow \alpha = \alpha_i \\
[]\ \alpha_i > 1 & \longrightarrow \alpha = 1 \\
\ ] & \\
,\ [\ p_i \geq p_o & \longrightarrow Q = \alpha k \sqrt{p_i - p_o} \\
[]\ p_i < p_o & \longrightarrow Q = -\alpha k \sqrt{p_o - p_i} \\
\ ] &
\end{array}
$$

The valve specification can be simplified considerably by introducing the functions clip and sign.

$$
\begin{array}{l}
\alpha = \mathrm{clip}(0, \alpha_i, 1) \\
,\ Q = \alpha k\, \mathrm{sign}(p_i - p_o) \sqrt{|p_i - p_o|}
\end{array}
$$

where clip is defined as

$$
\begin{array}{l}
\mathsf{func}\ \ \mathrm{clip}(x_{\min}, x, x_{\max} : \mathrm{real}) \rightarrow \mathrm{real} = \\
[\![\ [\ x \leq x_{\min} \longrightarrow \uparrow x_{\min} \\
\quad []\ -x_{\min} \leq x \leq x_{\max} \longrightarrow \uparrow x \\
\quad []\ x \geq x_{\max} \longrightarrow \uparrow x_{\max} \\
\quad \ ] \\
]\!]
\end{array}
$$

and sign is defined as

$$
\begin{array}{l}
\mathsf{func}\ \ \mathrm{sign}(x : \mathrm{real}) \rightarrow \mathrm{int} = \\
[\![\ [\ x < 0 \longrightarrow \uparrow -1 \\
\quad []\ x = 0 \longrightarrow \uparrow 0 \\
\quad []\ x > 0 \longrightarrow \uparrow 1 \\
\quad \ ] \\
]\!]
\end{array}
$$

The absolute function ($|p_i - p_o|$) is a well known mathematical function, and need not be separately defined. For simulation purposes the `abs()` library function is used. The sign and clip functions need be defined only once, and can then be used in many different specifications. In fact, they can also be made available as library functions.

CONCLUDING REMARKS

Currently the user needs to edit both the symbolic specification for documentation purposes and an ASCII version of the specification for simulation purposes. Future work includes the development of an automatic Chi-ASCII to χ-Latex translator.

The solver used in the Chi simulator is the DASSL [15] DAE solver. In future versions ODE solvers will also be included. Other research concentrates on the integration of a nonlinear equation solver in the Chi simulator that will solve both the initial state and the state after a discontinuity. In the current version of the simulator, the user needs to specify consistent initial conditions of the continuous variables.

REFERENCES

[1] J.E. Rooda, *Simulation of Logistics Elements (Sole)*, User manual, Twente University of Technology, Department of Mechanical Engineering, The Netherlands (1982).

[2] O.J. Dahl, B. Myhrhaug, K. Nygaard, *Simula-67 Common Base Language*, Tech. Rep. NCC Publication S-52, Norwegian Computing Centre, Oslo (1970).

[3] J.E. Rooda, S.M.M. Joosten, T.J. Rossingh, R. Smedinga, *Simulation in S84*, User manual, Twente University of Technology, Department of Mechanical Engineering, The Netherlands (1984).

[4] C. Bron, E.J. Dijkstra, *Report on the Programming Language Modular Pascal*, Tech. rep., Groningen University, The Netherlands (1987).

[5] J.E. Rooda, *Processcalculus Part I, New Instrument Describes Industrial Systems (in Dutch)*, I^2 Werktuigbouwkunde 7, p. 3 (1991).

[6] A. Goldberg, D. Robson, *Smalltalk-80, The Language*, Addison Wesley, Reading MA (1989).

[7] N.W.A. Arends, *A Systems Engineering Specification Formalism*, Ph.D. thesis, Eindhoven University of Technology, The Netherlands (1996).

[8] D.A. van Beek, J.E. Rooda, S.H.F. Gordijn, *Hybrid Modelling in Discrete-Event Control System Design*, in *CESA'96 IMACS Multiconference, Symposium on Discrete Events and Manufacturing Systems*, pp. 596–601, Lille (Jul. 1996).

[9] D. Albert van Beek, S.H.F. Gordijn, J.E. Rooda, *Integrating Continuous-Time and Discrete-Event Concepts in Modelling and Simulation of Manufacturing Machines*, Simulation Practice and Theory (1997), To be published.

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

[11] J.M. van de Mortel-Fronczak, 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*, pp. 421–426, Bejing (Oct. 1995).

[12] J.M. van de Mortel-Fronczak, J.E. Rooda, N.J.M van den Nieuwelaar, *Specification of a Flexible Manufacturing System Using Concurrent Programming*, Concurrent Engineering: Research and Applications 3, p. 187 (1995).

[13] E.W. Dijkstra, W.H.J. Feijen, *A Method of Programming*, Addison-Wesley (1988).

[14] D.A. van Beek, J.E. Rooda, M. van den Muyzenberg, *Specification of Combined Continuous-Time / Discrete-Event Models*, in *Modelling and Simulation 1996 ESM'96*, pp. 219–224, Budapest (Jun. 1996).

[15] L.R. Petzold, *A Description of DASSL: A Differential/Algebraic System Solver*, Scientific Computing pp. 65–68 (1983).