Proc. 9th Symposium on Information Control in Manufacturing, Vol. II, Nancy, 1998, pp. 77-82.

LANGUAGES AND APPLICATIONS IN HYBRID MODELLING: POSITIONING OF CHI

D.A. van Beek, **J.E. Rooda**

Eindhoven University of Technology, Department of Mechanical Engineering
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail: vanbeek@wtb.tue.nl, URL: http//se.wtb.tue.nl/

Abstract: A widely used classification of modelling languages distinguishes the categories continuous-time (CT), discrete-event (DE), discrete-time (DT), and hybrid. For better insight in the many different hybrid languages, a classification of 5 categories (CT, CT+, DE, DE+, and CT/DE) is proposed. Each category is explained, together with some of the included languages and the associated application fields. Special interest is given to the Chi language used for specification, simulation and real-time control of industrial systems. Its CT part is based on (conditional) DAE's, its DE part on Communicating Sequential Processes. The suitability of the language for DE, CT, and CT/DE modelling is illustrated by an example. *Copyright © 1998 IFAC*

Keywords: modelling, simulation languages, continuous time systems, discrete-event systems, discrete-event dynamic systems.

1. INTRODUCTION

A widely used classification of models and modelling languages is based on the progress of time in the model. In this respect, three categories can be distinguished: continuous-time (CT), discrete-event (DE), and discrete-time (DT). In continuous-time models, an infinite number of state changes occurs in any given finite time interval, whereas there are no state changes at discrete time points. In discrete-event models, the state changes at discrete time points only; in between two adjacent discrete time points the state remains the same. In discrete-time models, the state changes at equidistant time points only. Therefore, discrete-time models can be regarded as a subset of discrete-event models. For this reason, the two concepts CT and DE are sufficient. By combination of CT and DE concepts, hybrid models and hybrid modelling languages are obtained.

A classification of CT, DE, and CT/DE models is sufficient to distinguish model differences with respect to their time-dependent behaviour. A similar classification of modelling languages, however, gives little insight in the diversity (possibilities and restrictions) of the many different hybrid modelling languages. Therefore, a more precise classification is proposed that entails the following five categories: CT, CT+, DE, DE+, and CT/DE. In the sequel, the five categories are treated, and some concrete hybrid languages are categorized. No attempt is made to be complete in this respect. Only some more or less general purpose languages are mentioned, and not the very specialized languages, such as languages suited to modelling of multi-body systems only. Only the current state of the art is described, no attempt is made to describe the history of hybrid languages. A last restriction is that only those languages are considered that use high level language constructs in at least one domain (CT or DE). High level constructs enable the modeller to specify systems in a natural way, independently of implementation details of the language and solving algorithms used. As a consequence of the last-mentioned restriction, packages of library routines based on a general purpose programming language such as Fortran or C(++) are not considered. The relation between the application field of a hybrid modelling language and the choice of the hybrid language constructs is also treated. The hybrid χ (Chi) language is classified and explained in more detail, together with an illustrative example of the wide application field of the language.

2. HYBRID LANGUAGES AND APPLICATIONS

The differences between CT, CT+, DE, DE+, and CT/DE languages is treated below. CT languages are suited to CT model specification only. DE languages are suited to DE model specification only. Hybrid languages are divided in three categories: CT+, DE+, and CT/DE.

CT+ languages are CT languages that are extended with some DE constructs. The languages are not designed for specification of pure DE models. The facilities for DE modelling vary considerably among the languages in this category. Two sub classifications in the CT+ category are enumerated below.

(1) CT + discontinuity modelling.
Different DE language constructs for discontinuity specification may be provided, such as conditional equations, discontinuous functions, or language constructs for time-event and/or state-event specification. An example of such a language is Dymola (Elmqvist, 1994). Dymola allows non-causal specification of implicit DAE's up to index 1. It comes with many ODE/DAE solvers and makes extensive use of symbolic manipulation for solving and simplifying (differential) equations. The added DE constructs are mainly used for handling discontinuities in otherwise continuous systems. An important application field for such languages is modelling and simulation of control systems for continuous systems with discontinuities. Other applications are found in continuous-systems modelling such as electrical networks, multi-body systems etc.

(2) CT + operating procedure modelling.
The DE language constructs are designed for operating procedure modelling, such as modelling of batch recipes. DE model components are designed to interact with CT model components only, not with other DE components. The physical system is modelled as a CT system with discontinuities. A well known language in this category is gPROMS (Barton, 1992), that is used for chemical process modelling. Its CT part allows non-causal specification of implicit DAEs. The simulator contains a very efficient and optimized index 1 DAE solver, that can handle very large sets of equations. A nonlinear solver for calculation of the initial state and the state after a discontinuity is also included. Modularity of continuous components is provided by means of streams. There is no comparable mechanism for DE components.

DE+ languages are DE languages that are extended with elementary language constructs for continuous systems modelling. The languages are not designed for specification of pure CT models. An application field of such languages is modelling of certain batch plants, where scheduling of different products that are produced and/or stored in different tanks is important. The main continuous aspects in such models would be constant (on/off) flows from one vessel to another. Examples of such plants are beer breweries and fruit juice packing plants. The operations like mixing, stirring and brewing can be modelled as time passing (Δ or delay statements). The data flow based modelling language SIMAN is an example of a DE+ language. Because of the low level CT constructs in this language, the user must code the equations (ODE's or explicit DAE's of index 0) into a C programming language function 'cstate', using pointers etc. The user must also specify the 'equations' in the right order. The Personal Prosim (Sierenberg and Gans, 1992) language is another example. This is a process based language, where data at the top level is global. Attributes of objects can be accessed by other objects by means of pointers. The equations are of the ODE type, including DAE's of index 0, but lacking conditional equations.

CT/DE languages are equipped with high level language constructs in both the CT and DE domain, and can be used for specification of pure CT models as well as pure DE models. This makes it possible to model one part of the physical processes in a plant as CT, and another part as DE. The preparation of ice-cream, for example, could be described as a CT or batch process, but the subsequent packing of the individual ice-cornets could be described as a DE process. Such DE modelling requires high level data structures such as lists or queues, and tuples, arrays, or records. High level CT constructs should include conditional differential equations, where the equation chosen depends on a boolean condition. Two languages of the CT/DE category are COSMOS (Kettenis, 1994) and χ (or Chi). The CT part of COSMOS allows the specification of ODE's and explicit DAE's of index 0. The DE part is based on the process-interaction world view. Processes are explicitly activated and deactivated. The CT part of the χ language allows non-causal specification of implicit DAE's up to index 1. Apart from a DAE solver, the simulator also contains a nonlinear equation solver for calculation of the initial state and the state after discontinuities. The DE part is based on CSP (Communicating Sequential Processes) (Hoare, 1985).

The main purpose of the languages treated so far is to help understand the dynamic behaviour of systems by means of modelling and simulation. Another category of languages is mainly used for *analytical* derivation of model properties, such as proving the absence of deadlock, or proving that certain states will be or cannot be reached (e.g. (Alur *et al.*, 1995)). These languages are usually based on a finite state automaton, where each state can be associated with a set of ODE's. Petri nets are also used for analytical derivation of model properties. Many different hybrid Petri nets have been proposed. Several of these can be found in the Petri net survey of (David and Alla, 1994). Many of the hybrid Petri nets are flow nets, that incorporate extensions to the usual Petri nets to enable the modelling of flows. A more generally applicable approach is taken by allowing DAE's to be associated with each place in a Petri net (Daubas *et al.*, 1994). The DAE's associated with a place are activated if the place contains one or more tokens. Continuous variables in this net are global. Modularity and reuse of components is generally not provided for in Petri nets. Simulators for hybrid languages of this category (analytical derivation of model properties) are often not available or limited in functionality.

3. THE χ LANGUAGE

The χ language is used in modelling, simulation and real-time control of industrial systems. Research on formal verification of χ model properties has just started, and results are not yet of practical use in view of the complexity of the models. The language is based on a small number of orthogonal language constructs which makes it easy to learn and to use. Where possible, the continuous-time and discrete-event parts of the language are based on similar concepts. Parametrized processes and systems provide modularity, and enable 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 (Beek and Rooda, 1997). 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 language constructs and design considerations is also presented in (Arends, 1996).

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*) $=$
$[\![$ *process and system declarations*
, *variable and channel declarations*
| *process and 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: differential algebraic equations), 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 channel is declared in systems and is either a discrete communication channel (e.g. p : !? int), a discrete synchronization channel (e.g. s : $^\sim$ void) or a continuous channel (e.g. Q : $\multimap$ [m^3/s]). The special void data type denotes the absence of data. If a communication channel is declared as a process or system parameter, the usage of the channel is declared as either discrete output (e.g. p : ! int), or discrete input (e.g. p : ? int). 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 communication channel by an arrow, and a synchronization channel by a dotted line (optionally ending in an arrow head to denote a cause-effect relationship). Processes and systems are represented by circles.

All data types and variables are declared as either continuous using a double colon (e.g. V :: [m^3]), or discrete using a single colon (e.g. n : int). The value of a discrete variable is determined by assignments (e.g. $n := 1$). 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 (e.g. initialization $V ::= 0$) determines its value for the current point of time only.

Some discrete data types are predefined like bool (boolean), int (integer) and real. Variables may be declared with units (e.g. v :: [m/s]). All variables with units are of type real.

Structured types are defined by means of tuple types and list types. A variable t of a record-like tuple type is declared as $t : id_1.T_1 \times id_2.T_2 \times \cdots \times id_n.T_n$, where $id_1 \cdots id_n$ are identifiers, and $T_1 \cdots T_n$ are types. This defines a variable t with elements $t.id_1 \cdots t.id_n$ of type $T_1 \cdots T_n$, respectively. Initialization of t may be done by assignment: $t := \langle t_1, t_2, \ldots, t_n \rangle$, where t_i is an expression of type T_i $(1 \le i \le n)$. If all elements of the tuple are of the same type and they are not named (array-like type), the declaration is $t : T^n$. This declares an n-tuple t, the elements of which are referenced as $t.i$ (i being a variable or constant with $0 \le i < n$). A list (queue) *xs* is declared as $xs : T^*$. This defines a list of elements of type T. The first element (head) is obtained by hd(*xs*), the list without the first element (tail) by tl(*xs*). Initialization may be done by $xs := [x_1, \ldots, x_n]$, where x_i is an expression of type T.

3.1 *The continuous-time part of* χ

A time derivative is denoted by a prime character (e.g. x'). DAEs are separated by commas: $DAE_1, DAE_2, \ldots, DAE_n$. A DAE can be a normal equation or a *guarded equation.* The latter is used when the set of equations depends on the state of the system. The syntax is $[\, b_1 \longrightarrow DAEs_1 \, [\!] \ldots [\!] \, b_n \longrightarrow DAEs_n \,]$, where $DAEs_i$ $(1 \le i \le n)$ represents one or more DAEs. The boolean expression b_i denotes a *guard.* At any time, at least one of the guards must be open (true), so that the DAEs $DAEs_i$ 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.

3.2 *The discrete-event part of χ*

The discrete-event part of χ is a CSP-like (Hoare, 1985) real-time concurrent programming language, described in e.g. (Mortel-Fronczak *et al.*, 1995).

Interaction between discrete-event parts of processes takes place by means of synchronous communication ($c\,!\,e$ or $c\,?\,x$) or by synchronization only ($s\,^\sim$). Consider channel c (or s) connecting two processes. Execution of $c\,!\,e$ or $c\,?\,x$ in one process causes the process to be blocked until $c\,?\,x$ or $c\,!\,e$ is executed in the other process, respectively. Subsequently the value of expression e is assigned to variable x. Execution of $c\,^\sim$ in one process causes the process to be blocked until $s\,^\sim$ is executed in the other process.

Time passing is denoted by $\Delta\, t$, where t is an expression of type real. 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 $(1 \le i \le n)$ 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 $[\, b_1;\, E_1 \longrightarrow S_1 \,[]\, \ldots \,[]\, b_n;\, E_n \longrightarrow S_n \,]$. There are five types of event statement E_i: output ($c\,!\,e$), input ($c\,?\,x$), synchronization ($c\,^\sim$), time ($\Delta\, t$), and state ($\nabla\, r$, explained in the following section). 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. A time event statement $\Delta\, t$ can take place when t time units have passed. 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. 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 statement [*GB*] or [*GW*] is denoted by $*$[*GB*] or $*$[*GW*], respectively. In this case, it is not necessary that at least one of the guards is open. The repetition terminates when all guards are closed. The repetition $*[\,\text{true} \longrightarrow S\,]$ may be abbreviated to $*[\,S\,]$.

3.3 *Continuous / discrete interaction*

In the discrete-event part of a process, assignments can be made to discrete variables occurring in DAEs (e.g. $\alpha.j := 1$, see Section 4) or in the boolean guards of guarded DAEs (e.g. *full* := 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, and may be reinitialized in the discrete-event part. In both cases the symbol ::= 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 $\nabla\, r$, where r is a relation involving at least one continuous variable, causes the process to be blocked until the relation becomes true.

4. EXAMPLE

The example is inspired by an existing catalyst production plant that produces three different products. The plant consists of three mixing tanks, that can each mix any one of the three products. After mixing, the contents are transferred to a buffer tank. At most two mixing tanks are allowed to transfer their contents to the buffer at the same time. Emptying of the buffer tank is not modelled. A process for emptying the buffer can be connected to channel Q_{BT}. Figure 1 shows the CatalystPlant model structure. The formal textual system definition follows below.

syst CatalystPlant$(Q_{\text{BT}} : \multimap \text{flow}) =$
$[\![\; OG : \text{OrderGen},\; S : \text{Supplier},\; BT : \text{BufferTank}$
$,\; MT : \text{MixingTank}^3,\; j : \text{int},\; Q : (\multimap \text{flow})^3$
$,\; \mathit{prod} : (!?\,\text{prod})^3,\; MT_{\text{full}}, MT_{\text{empty}} : ({}^\sim\,\text{void})^3$
$,\; \mathit{ord} : !?\,\text{order},\; n_{\text{todo}} : !?\,\text{batchcount}$
$\mid\; OG(\mathit{ord}, n_{\text{todo}}) \parallel S(\mathit{ord}, \mathit{prod})) \parallel j : [0..3) :$
$\quad MT.j\,(Q.j, \mathit{prod}.j, MT_{\text{full}}.j, MT_{\text{empty}}.j, 2, 1, 1)$
$\parallel BT(Q,\; Q_{\text{BT}}, MT_{\text{full}}, MT_{\text{empty}}, n_{\text{todo}}, 0.5, 1, 0.1)$
$]\!]$

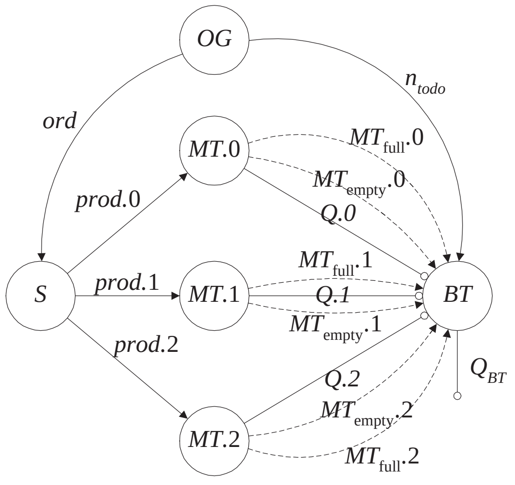

Fig. 1. System CatalystPlant.

Declaration MT : MixingTank^3 defines MT to be a tuple of three processes ($MT.0$, $MT.1$, $MT.2$), each one of type MixingTank (see the declaration proc MixingTank below).

The table below explains the meaning of the identifiers used in the models.

batchcount	number of batches
α	tuple of factors for flow equations
i_{next}	index of next MT
n_{doing}	number of MTs emptying
n_{todo}	number of MTs still to be emptied

The following types are used in the model:

```
type flow       $= [\text{m}^3/\text{s}]$
,    vol        $= [\text{m}^3]$
,    prod       $= \{\text{P1}, \text{P2}, \text{P3}\}$
,    batchcount $=$ int
,    order      $= n.\text{batchcount} \times p.\text{prod}$
```

Order generator OG of type OrderGen is modelled with an order list os. Each order consists of a number, that indicates how many batches of the product are required, and a product type. The list contains the following orders: 5 batches of product P1, followed by 6 batches P2, 3 batches P3, and 10 batches P1. A batch equals one full mixing tank. The order items are sent to supplier S ($ord\,!\,\text{hd}(os)$). The number of products are sent to buffer tank BT. Supplier S needs to know the product type because of the product dependent mixing time. The buffer tank needs to know only the number of batches.

```
proc OrderGen($ord$ : ! order, $n_{\text{todo}}$ : ! batchcount ) =
|[ $os$ : order$^*$
| $os := [\langle 5, \text{P1}\rangle, \langle 6, \text{P2}\rangle, \langle 3, \text{P3}\rangle, \langle 10, \text{P1}\rangle]$
; *[ $\text{len}(os) > 0$
      $\longrightarrow ord\,!\,\text{hd}(os)$ ; $n_{\text{todo}}\,!\,\text{hd}(os).n$ ; $os := \text{tl}(os)$
   ]
]|
```

Parameter declaration $prod$: $(!\,\text{prod})^3$ in process S of type Supplier declares a tuple of 3 output channels through which a product of type prod is sent. The process first receives an order ($ord\,?\,r$). The send action $prod.i_{\text{next}}\,!\,r.p$ indicates that the tank with number i_{next} is the next tank to be filled with product type $r.p$. When mixing tank $MT.i_{\text{next}}$ ($0 \le i_{\text{next}} < 3$) has reached the receive statement $prod\,?\,p$ (see specification of process MixingTank), the interaction takes place and the value $r.p$ from process S is assigned to variable p of process $MT.i_{\text{next}}$. If the number of batches (n_{todo}) equals 4 for example, and i_{next} equals 1, then first mixing tank $MT.1$ is filled, followed by the sequence of $MT.2$, $MT.0$, and $MT.1$ ($i_{\text{next}} := (i_{\text{next}} + 1) \bmod 3$). Buffer tank BT empties the tanks in the same order (see proc BufferTank). Therefore, it cannot accidentally start emptying a mixing tank that contains a different product than the product in the buffer tank.

```
proc Supplier($ord$ : ? order, $prod$ : $(!\,\text{prod})^3$ ) =
|[ $r$ : order, $i_{\text{next}}$ : int , $n_{\text{todo}}$ : batchcount
| $i_{\text{next}} := 0$
; *[ $ord\,?\,r$ ; $n_{\text{todo}} := r.n$
   ; *[ $n_{\text{todo}} > 0 \longrightarrow prod.i_{\text{next}}\,!\,r.p$
                          ; $n_{\text{todo}} := n_{\text{todo}} - 1$
                          ; $i_{\text{next}} := (i_{\text{next}} + 1) \bmod 3$
      ]
   ]
]|
```

Processes $MT.0$, $MT.1$, and $MT.2$ are hybrid processes of type MixingTank. Filling of a tank is modelled in a discrete-event way, emptying of the tank by means of the differential equation $V' = -Q$. After the product type has been received in $prod\,?\,p$, filling of the tank is modelled by $\Delta\, t_{\text{fill}}$, which models passing of t_{fill} units of time. After that, the volume of the tank becomes $V_{\max}$ ($V ::= V_{\max}$). Mixing is also modelled in a discrete-event way. The product dependent mixing time is calculated by the function call prod2mixTime(p). After mixing, the mixing tank tries to synchronize with buffer tank BT ($MT_{\text{full}}\,^{\sim}$). When the buffer tank is ready to receive the contents of the mixing tank, the synchronization $MT_{\text{full}}\,^{\sim}$ succeeds (BT executes $MT_{\text{full}}.i_{\text{next}}\,^{\sim}$), causing the mixing tank to wait until it is empty in $\nabla\, V \le 0$ (the pump is modelled in the buffer tank). When it is empty, BT is informed by means of $MT_{\text{empty}}\,^{\sim}$, and the tank is cleaned ($\Delta\, t_{\text{clean}}$). After this, the cycle is restarted at $prod\,?\,p$.

```
proc MixingTank
( $Q\_$ : $-\!\!\circ$ flow
, $prod$ : ? prod, $MT_{\text{full}}, MT_{\text{empty}}$ : $^{\sim}$ void
, $t_{\text{fill}}, t_{\text{clean}}, V_{\max}$ : real
) =
|[ $V$ :: vol, $Q$ :: flow
, $p$ : prod
; $V ::= 0$
| $Q\_ -\!\!\circ Q$
| $V' = -Q$
| *[ $prod\,?\,p$ ; $\Delta\, t_{\text{fill}}$ ; $V ::= V_{\max}$
   ; $\Delta$ prod2mixTime($p$) ; $MT_{\text{full}}\,^{\sim}$
   ; $\nabla\, V \le 0$ ; $MT_{\text{empty}}\,^{\sim}$ ; $\Delta\, t_{\text{clean}}$
   ]
]|
```

```
func prod2mixTime($p$ : prod ) $\rightarrow$ real =
|[ [ $p = \text{P1} \longrightarrow\, \uparrow 1.0$
   [] $p = \text{P2} \longrightarrow\, \uparrow 1.5$
   [] $p = \text{P3} \longrightarrow\, \uparrow 1.2$
   ]
]|
```

The three pumps that empty the respective MT tanks are modelled in process BT of type BufferTank by means of the tuple α and the three equations $Q.j = \alpha.j \cdot Q_{\text{set}}$ ($0 \le j < 3$). If $\alpha.j$ equals 1, pump j is on; if it equals 0, the pump is off. The boolean $full$ is normally false. It becomes true when the buffer tank becomes completely full before the contents of a mixing tank (a batch) has been transferred ($n_{\text{doing}} > 0 \wedge \neg full$; $\nabla V > V_{\max} \longrightarrow full :=$ true). The incoming flows then become zero ($full \longrightarrow Q.j = 0$ for $0 \le j < 3$). In the repetitive selective waiting statement

($*$[...]) four events can take place, each one with its own condition: (1) If not all batches of an order have been received ($n_{\text{todo}} - n_{\text{doing}} > 0$), and there less than two tanks being emptied ($n_{\text{doing}} < 2$), a new tank may be emptied ($MT_{\text{full}}.i_{\text{next}}{}^{\sim}$). (2) If there are mixing tanks that are being emptied ($n_{\text{doing}} > 0$), the synchronization that the mixing tank is empty is awaited ($MT_{\text{empty}}.j^{\sim}$). The enumeration variable j runs from 0 to 2 (because MT_{empty} is a tuple of three channels), and thus creates an alternative for every mixing tank. For instance $j : b; MT_{\text{empty}}.j^{\sim} \longrightarrow \alpha.j := 0$, where b is a boolean expression, is equivalent to $b; MT_{\text{empty}}.0^{\sim} \longrightarrow \alpha.0 := 0$ [] ... [] $b; MT_{\text{empty}}.2^{\sim} \longrightarrow \alpha.2 := 0$. If the last mixing tank of an order has been emptied ($n_{\text{todo}} = 0$), the buffer tank must be emptied ($\nabla\, V \leq 0$) before an new order can be received ($n_{\text{todo\_}}\, ?\, n_{\text{todo}}$). (3) If there are mixing tanks that are being emptied ($n_{\text{doing}} > 0$), and the buffer tank is not yet full ($\neg$*full*), the buffer tank may become full ($\nabla V > V_{\max}$). As a result variable *full* becomes true (*full* := true), which causes flows $Q.j$ to become 0. Event (4) can be explained in a similar way.

```
proc BufferTank
( $Q\_ : (\multimap \text{flow})^3,\ Q_{\text{BT\_}} : \multimap \text{flow}$
, $MT_{\text{full}}, MT_{\text{empty}} : ({}^{\sim}\,\text{void})^3,\ n_{\text{todo\_}} : ?\,\text{batchcount}$
, $Q_{\text{set}}, V_{\max}, V_{\text{hys}} : \text{real}$
) =
$[\![\ V : \text{vol},\ Q : \text{flow}^3,\ Q_{\text{BT}} : \text{flow},\ \mathit{full} : \text{bool}$
, $n_{\text{todo}}, n_{\text{doing}} : \text{batchcount},\ i_{\text{next}} : \text{int}\ ,\ \alpha : \text{int}^{\ 3}$
; $V ::= 0;\ \mathit{full} := \text{false};\ \alpha := (0, 0, 0)$
| $j : [0..3) : Q\_.j \multimap Q.j$
, $Q_{\text{BT\_}} \multimap Q_{\text{BT}}$
| $V' = Q.0 + Q.1 + Q.2 - Q_{\text{BT}}$
, $j : [0..3) :$ [   $\mathit{full} \longrightarrow Q.j = 0$
                 [] $\neg \mathit{full} \longrightarrow Q.j = \alpha.j \cdot Q_{\text{set}}$
                 ]
| $n_{\text{doing}} := 0;\ i_{\text{next}} := 0;\ n_{\text{todo\_}}\,?\,n_{\text{todo}}$
; $*$[ $n_{\text{todo}} - n_{\text{doing}} > 0 \wedge n_{\text{doing}} < 2;\ MT_{\text{full}}.i_{\text{next}}{}^{\sim}$
      $\longrightarrow \alpha.i_{\text{next}} := 1;\ n_{\text{doing}} := n_{\text{doing}} + 1$
         ; $i_{\text{next}} := (i_{\text{next}} + 1) \bmod 3$
   [] $j : n_{\text{doing}} > 0;\ MT_{\text{empty}}.j^{\sim}$
      $\longrightarrow \alpha.j := 0$
         ; $n_{\text{doing}} := n_{\text{doing}} - 1;\ n_{\text{todo}} := n_{\text{todo}} - 1$
         ; [ $n_{\text{todo}} = 0 \longrightarrow \nabla\, V \leq 0;\ n_{\text{todo\_}}\,?\,n_{\text{todo}}$
           [] $n_{\text{todo}} > 0 \longrightarrow \text{skip}$
           ]
   [] $n_{\text{doing}} > 0 \wedge \neg \mathit{full};\ \nabla\, V > V_{\max}$
      $\longrightarrow \mathit{full} := \text{true}$
   [] $\mathit{full};\ \nabla\, V < V_{\max} - V_{\text{hys}}$
      $\longrightarrow \mathit{full} := \text{false}$
   ]
$]\!]$
```

5. CONCLUDING REMARKS

The proposed new classification of modelling languages in CT, CT+, DE, DE+, and CT/DE categories gives better insight in the diversity of the so called hybrid modelling languages. The classification in different categories does not imply languages from one category to be better than languages from another. Depending on the type of applications that a modeller deals with, he or she should choose a language that fulfils his or her needs best. The χ language has been shown to have a wide field of application. It has been successfully applied to a large number of complex industrial cases, such as an integrated circuit manufacturing plant, and a beer brewery.

ACKNOWLEDGMENT

We would like to thank G. Fábián for helpful comments.

REFERENCES

Alur, R., C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.H. Ho, X. Nicollin, A. Olivero, J. Sifakis and S. Yovine (1995). The algorithmic analysis of hybrid systems. *Theoretical Computer Science* **138**, 3–34.

Arends, N.W.A. (1996). A Systems Engineering Specification Formalism. PhD thesis. Eindhoven University of Technology. The Netherlands.

Barton, P.I. (1992). The Modelling and Simulation of Combined Discrete/Continuous Processes. PhD thesis. University of London.

Beek, D.A. van and J.E. Rooda (1997). Specification and simulation of industrial systems using an executable mathematical specification language. In: *Proc. of the 15th. IMACS World Congress Vol. 2 Numerical Mathematics*. Berlin. pp. 721–726.

Daubas, B., A. Pages and H. Pingaud (1994). Combined simulation of hybrid processes. In: *IEEE International Conference on Systems, Man and Cybernetics*. San Antonio, Texas. pp. 320–325.

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

Elmqvist, H. (1994). *Dymola – Dynamic Modeling Language – User's Manual*. Dynasim AB. Lund, Sweden.

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

Kettenis, D.L. (1994). Issues of Parallelization in Implementation of the Combined Simulation Language COSMOS. PhD thesis. Delft University of Technology.

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

Sierenberg, R.W. and O.B. de Gans (1992). Personal Prosim: A fully integrated simulation environment. In: *Proc. Of 1992 European Simulation Symposium*. SCS, San Diego. pp. 167–173.