Model-based engineering of embedded systems using the hybrid process algebra Chi

J.C.M. Baeten, D.A. van Beek, P.J.L. Cuijpers, M.A. Reniers
J.E. Rooda, R.R.H. Schiffelers, R.J.M. Theunissen[1]

Department of Mechanical Engineering and
Department of Mathematics and Computer Science
Eindhoven University of Technology (TU/e)
Eindhoven, The Netherlands

Abstract

Hybrid Chi is a process algebra for the modeling and analysis of hybrid systems. It enables modular specification of hybrid systems by means of a large set of atomic statements and operators for combining these. For the efficient implementation of simulators and the verification of properties of hybrid systems it is convenient to have a model that uses a more restricted part of the syntax of hybrid Chi. To that purpose the linearization of a reasonably expressive, relevant subset of the Chi language is discussed. A linearization algorithm that transforms any specification from this subset into a so-called normal form is presented. The algorithm is applied to a bottle-filling line example to demonstrate tool-based verification of Chi models.

Keywords: hybrid systems, process algebra, hybrid automata, modeling, linearization, simulation, verification.

1 Introduction

The χ (Chi) formalism [5,20] [2] is a hybrid process algebra. The intended use of χ is for modeling, simulation, verification, and real-time control of discrete-event, continuous or combined, so-called hybrid systems. Its application domain ranges from physical phenomena, such as dry friction, to large and complex manufacturing systems, such as integrated circuit manufacturing plants, breweries, and process industry plants [8,18,2]. These plants consist of many independently operating entities such as machines, buffers, liquid storage tanks and reactors. The entities interact with each other in a discrete fashion, for example, the exchange of products or information, or in a continuous fashion, for example, sharing a liquid flow. The χ formalism has been designed to model interacting parallel entities representing both discrete and continuous behavior in an easy and intuitive way. This is ensured

[1] Email: josb@win.tue.nl, D.A.v.beek@tue.nl, P.J.L.Cuijpers@tue.nl, M.A.Reniers@tue.nl, J.E.Rooda@tue.nl, R.R.H.Schiffelers@tue.nl, R.J.M.Theunissen@tue.nl

[2] The χ language as defined in [20] has small corrections w.r.t. the version defined in [5]

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

by strong support for modular composition by allowing unrestricted combination of operators such as sequential and parallel composition, by providing statements for scoping, by providing process definition and instantiation mechanisms, and by providing different interaction mechanisms, namely synchronous communication and shared variables. The fact that the χ process algebra is such a rich language potentially complicates the development of tools for χ, since the implementations have to deal with all possible combinations of the χ atomic statements and the operators that are defined on them. This is where the process algebraic approach of equational reasoning, that allows rewriting models to a simpler form, is essential.

To illustrate the required implementation efforts, consider the following implementations that have been developed:

- a Python simulator implementation for rapid prototyping [6];
- a C simulator implementation for fast model execution;
- an implementation based on the MATLAB Simulink S-functions [22] enabling co-simulation [3];
- an implementation for real-time control [15].

In [7] it has been shown that different (timed) model checkers each have their own strengths and weaknesses. Therefore, for verification, translations to several tools are defined:

- For hybrid models a translation from χ to the hybrid I/O automaton based model checker PHAVer [12] is defined in [4], which has been proven correct.
- For timed models the following translations are defined (see [7]):
  - a translation to the action-based process algebra μCRL [13], used as input language for the verification tool CADP [11];
  - a translation to PROMELA, a state-based, imperative language, used as input language for the verification tool SPIN [16];
  - a translation to the timed automaton based input language of the UPPAAL [19] verification tool.

Instead of defining the implementations mentioned above on the full χ language as defined in [5,20], the process algebraic approach of equational reasoning makes it possible to transform χ models into a normal form, which consists of less operators allowed only in very restricted combinations, and to define the implementations on this normal form. The original χ model and its normal form are (stateless) bisimilar [5,20], which ensures that model properties are preserved.

An advantage of operating on normal forms is that generally it is much easier to define a simulator for process terms that are in normal form than for arbitrary process terms and therefore it is also much easier to generate a (hybrid) transition system from normal forms. For many types of model properties there are reasonably efficient means available for verification on process terms without complex mechanisms such as parallel composition and hierarchy. For example, one can consider translations from (the subset used of) χ to other formalisms such as hybrid automata [20].

In general, one may say that the use of process models in normal form simplifies further analysis and makes the implementation of tools easier and more efficient.

In this paper, an algorithm for the automatic generation of the normal form of a χ model is described, and the correctness of this algorithm is proven. The approach followed is comparable to the approach of [24,10]. However, there are some differences that reduce the complexity of the algorithm considerably. Instead of linearizing the full χ language, a relevant subset of the χ language has been chosen.

First, recursion is allowed only in a well structured form:

- by assuming tail-recursion in the input specification, the complex stacking of recursion variables in [24,10] has been avoided;
- all recursion variables that are used within a recursion scope are assumed to be also defined within that scope, i.e. recursion scopes are complete.

Second, the signal emission operator, variables scope operator, channel scope operator and urgent communication operator as defined in [5,20] are omitted from the linearization algorithm, since these operators in general occur only at the top level of χ models.

The outline of this paper is as follows. In Section 2 the input language for the linearization algorithm is defined. The normal form that is the output of the linearization algorithm is defined in Section 3, and Section 4 contains the linearization algorithm itself. In Section 5, the linearization algorithm is illustrated by means of a case study. Concluding remarks and future work are discussed in Section 6.

2 Syntax and informal semantics of a subset of χ

In this section, a concise definition of the syntax and informal semantics of a subset of χ is defined. The complete χ syntax and semantics are defined in [5,20].

2.1 Syntax

A χ model has the following form [3]:

$$\mathsf{model}\ id(D_{\mathrm{m}}) = [\![\ D :: p_{\mathrm{t}}\ ]\!]$$

$$D_{\mathrm{m}} ::= \mathsf{val}\ \ S\ \{, S\}^* \ \mid\ D_{\mathrm{m}}, D_{\mathrm{m}} \qquad \text{model parameter declaration}$$

$$S \quad ::= id\ \{, id\}^* : t \qquad \text{declaration without initialization}$$

where id is an identifier that represents the name of the model, and D_{m} denotes the model parameters. The model parameter declaration may be also be empty. The body of a χ model consists of a declaration part D, in which channels, variables and modes can be declared, and the process term p_{t}. The syntax of the declaration D is:

[3] The notation introduced here is an abbreviation of a triple $\langle p, \sigma, E\rangle$, where p denotes the process term, σ denotes the valuation, and E denotes an environment. A valuation is a partial function from variables to values. An environment E is a tuple (C, J, L, H, R), where C denotes the set of continuous variables, J denotes the set of jumping variables, L denotes the set of algebraic variables, H denotes the set of channels, and R denotes a recursion definition. A recursion definition is a partial function from recursion variables to process terms.

$D ::=$	$\mathsf{chan}\ \ S\ \{,S\}^*$	channel declaration
	$\mid (\mathsf{var}\ \mid \mathsf{cont}\ \mid \mathsf{alg}\ )\ \ IS\ \{,IS\}^*$	variable declaration
	$\mid \mathsf{mode}\ \ \ X = p\ \{,X = p\}^*$	mode declaration
	$\mid D, D$	
$IS ::=$	$id\ \{,id\}^* : t = e\ \mid\ S$	declaration with initialization

Here, t denotes the type of a variable or channel, e denotes an initialization expression, and id denotes an identifier. The following items can be declared in D:

- Channels, such as in $\mathsf{chan}\ h : \mathsf{real},\ close : \mathsf{void}$. This declares a communication channel h, that communicates values of type real, and a synchronization channel $close$ (no data exchange).
- Discrete variables, such as in $\mathsf{var}\ k, n : \mathsf{int},\ v_{\mathrm{set}} : \mathsf{real} = 1.0$. This declares two uninitialized variables k, n of type int (integer), and a variable v_{set} that has an initial value 1.0.
- Continuous variables, such as in $\mathsf{cont}\ x : \mathsf{real} = 1.0$. Continuous variables are the only variables for which dotted variables (derivatives) can be used. Therefore, the declaration $\mathsf{cont}\ x : \mathsf{real} = 1.0$ implies that x and its dotted version $\dot{x}$, can both be used.
- Algebraic variables, such as in $\mathsf{alg}\ y, z : \mathsf{real}$.
- Modes, such as mode filling $= (V \geq 10 \rightarrow n := 0; \mathrm{emptying})$.

Besides the variables defined above, the existence of the predefined reserved global variable time which denotes the current time, the value of which is initially zero, is assumed. This variable cannot be declared. It can only be used in expressions occurring in the process terms. Using a BNF-like notation (Backus Naur Form), the subset of χ process terms that can be linearized is defined by the following grammar for the process terms $p_{\mathrm{t}} \in \mathcal{P}_{\mathrm{t}}$:

$$p_{\mathrm{t}} ::= \quad [\![\ D :: p_{\mathrm{s}}\ ]\!]$$

where D denotes declarations as already defined. The syntax of the process terms p_{s} defines the set of all process terms $\mathcal{P}_{\mathrm{s}}$:

$p_{\mathrm{s}} ::=$	p_{atom}	atomic process terms
	$[p_{\mathrm{atom}}]$	delayable atomic process terms
	$b \rightarrow p_{\mathrm{s}}$	guard operator
	$p_{\mathrm{s}};\ p_{\mathrm{s}}$	sequential composition operator
	$p_{\mathrm{s}}\ [\!]\ p_{\mathrm{s}}$	alternative composition operator
	$p_{\mathrm{s}} \parallel p_{\mathrm{s}}$	parallel composition operator
	p_{R}	(restricted use of) recursion scope operator
	$\partial_A(p_{\mathrm{s}})$	action encapsulation operator

where b denotes a predicate over variables and dotted continuous variables, and $A \subseteq \mathcal{A}$ is a set of action labels. The set of all possible actions $\mathcal{A}$ is defined as $\mathcal{A} = A_{\text{label}} \cup A_{\text{com}}$, with $A_{\text{label}} \cap A_{\text{com}} = \emptyset$. The set of communication actions A_{com} is defined as $A_{\text{com}} = \{\text{isa}(h, cs), \text{ira}(h, cs, W), \text{ca}(h, cs) \mid h \in \mathcal{H}, cs \in \Lambda^*, W \subseteq \mathcal{V}\}$, where isa, ira and ca denote action labels for the internal send action, the internal receive action, and the communication action respectively. The sets $\mathcal{H}$, Λ^* and $\mathcal{V}$ denote the sets of all possible channels, all possible lists of values, and all possible variables respectively.

We assume that action encapsulation is based on channel names and not on the values sent or received, this restricts the set A as follows:

$$\begin{gathered}\forall_{h\in\mathcal{H},x,y\in\Lambda^*}\ \text{isa}(h,x) \in A \implies \text{isa}(h,y) \in A,\\ \forall_{h\in\mathcal{H},x,y\in\Lambda^*,v_1,v_2\subseteq\mathcal{V}}\ \text{ira}(h,x,v_1) \in A \implies \text{ira}(h,y,v_2) \in A\text{and}\\ \forall_{h\in\mathcal{H},x,y\in\Lambda^*}\ \text{ca}(h,x) \in A \implies \text{ca}(h,y) \in A.\end{gathered}$$

The operators are listed in descending order of their binding strengths as follows $\rightarrow$, ; , $\{\|\ ,\ [\!]\ \}$. The operators inside the braces have equal binding strength. In addition, operators of equal binding strength associate to the right, and parentheses may be used to group expressions. For example, $p;\ q;\ r$ means $p;\ (q;\ r)$.

The syntax of the atomic process terms p_{atom} defines the set of all atomic process terms $\mathcal{P}_{\text{atom}}$:

p_{atom}	::=	p_{act}	action process terms
	$\mid$	u	delay predicate process term

where u is a predicate over variables and dotted continuous variables.

The syntax of the action process terms p_{act} defines the set $\mathcal{P}_{\text{act}}$:

p_{act}	::=	$W : r \gg l_a$	action predicate process term
	$\mid$	$h\,!!\,\mathbf{e}_n$	send process term
	$\mid$	$h\,??\,\mathbf{x}_n$	receive process term
	$\mid$	$h\,!?\,\mathbf{x}_{\text{n}} := \mathbf{e}_{\text{n}}$	communication process term

Here, $W \subseteq \mathcal{V}$ is a set of variables, and r is a predicate over variables, dotted continuous variables, and '$-$' superscripted variables (including the dotted variables). Furthermore, $l_a \in A_{\text{label}}$ is an action label, $h \in \mathcal{H}$ is a channel, $\mathbf{e}_n$ denotes the expressions $e_1, \ldots, e_n$, and $\mathbf{x}_n$ denotes the variables $x_1, \ldots, x_n$. For $n = 0$, $h\,!!\,\mathbf{x}_n$, $h\,??\,\mathbf{x}_n$ and $h\,!?\,\mathbf{x}_{\text{n}} := \mathbf{e}_{\text{n}}$ can be written as $h\,!!$, $h\,??$ and $h\,!?$, respectively.

Note that the communication process term $h\,!?\,\mathbf{x}_{\text{n}} := \mathbf{e}_{\text{n}}$ is not part of the 'core' elements of the χ formalism as defined in [5]. It is, however, defined in [20]. Parallel composition allows the synchronization of matching send and receive actions. The result of the synchronization is a communication action. The syntax of the communication is given in terms of other language elements (send process term, receive process term, parallel composition, and action encapsulation). In normal forms parallel composition is eliminated; therefore the communication process term $h\,!?\,\mathbf{x}_{\text{n}} := \mathbf{e}_{\text{n}}$ is introduced to represent the communication action as a single atomic

process term. The following property holds:

$$h\,!?\,\mathbf{x}_\mathrm{n} := \mathbf{e}_\mathrm{n} \leftrightarrows \partial_{\{\mathrm{isa}(h,cs),\mathrm{ira}(h,cs,v)|cs\in\Lambda^*,v\subseteq\mathcal{V}\}}(h\,!!\,\mathbf{e}_n \parallel h\,??\,\mathbf{x}_n),$$

where $\leftrightarrows$ denotes stateless bisimilarity.

The recursion scope operator p_R is restricted to tail-recursion only. Its syntax is defined as:

$$\begin{aligned} p_\mathrm{R} ::= \quad & [\![_\mathrm{R}\, R :: X\,]\!] \quad \text{satisfying the condition: } X \in \mathrm{dom}(R) \\ \mid \quad & [\![_\mathrm{R}\, R :: p\,]\!] \quad \text{satisfying the condition: } \mathrm{rvar}(p) \subseteq \mathrm{dom}(R), \end{aligned}$$

where $X \in \mathcal{X}$ denotes a recursion variable ($\mathcal{X}$ is the set of all recursion variables), and R is a recursion definition. The syntax of process terms p defines the set of all process terms $\mathcal{P}$:

$$p ::= \quad p_\mathrm{s} \mid b \to p \mid p_\mathrm{s};\, X \mid p_\mathrm{s};\, p \mid p \,[]\, p$$

The function $\mathrm{rvar} : \mathcal{P} \to \mathcal{X}$ extracts the recursion variables used in process terms from $\mathcal{P}$. It is defined as:

$$\begin{aligned} \mathrm{rvar}(p_\mathrm{s}) &= \emptyset \\ \mathrm{rvar}(b \to p) &= \mathrm{rvar}(p) \\ \mathrm{rvar}(p_\mathrm{s}; X) &= \{X\} \\ \mathrm{rvar}(p_\mathrm{s}; p) &= \mathrm{rvar}(p) \\ \mathrm{rvar}(p \,[]\, q) &= \mathrm{rvar}(p) \cup \mathrm{rvar}(q) \end{aligned}$$

where $p_\mathrm{s} \in \mathcal{P}_\mathrm{s}$ and $p, q \in \mathcal{P}$. Sometimes, this function is lifted to recursive definitions by defining: $\mathrm{rvar}(R) = \bigcup_{X\in\mathrm{dom}(R)} \mathrm{rvar}(R(X))$.

An additional restriction is that the recursion definition used in a recursion scope operator process term must be complete, in the sense that all recursion variables used in the recursion definition must be defined in the same recursion definition. Formally, $\mathrm{rvar}(R) \subseteq \mathrm{dom}(R)$. In the sequel, recursion scope will be used as a shorthand for recursion scope operator process term, and a recursion scope with a complete recursion definition is called a complete recursion scope.

For many of the core process terms introduced before, there is additional, more user-friendly syntax available (see [5,20] for a complete overview.) Trivially, the user-friendly syntax that is defined in terms of process terms that belong to the input language of the linearization algorithm as defined above, can be used as input as well. A list of these additional process terms that are used in the bottle filling line example in Section 5 is given below.

- $\delta \triangleq \emptyset$: false $\gg \tau$
- skip $\triangleq \emptyset$: true $\gg \tau$
- $\mathbf{x}_n := \mathbf{e}_n \triangleq \{\mathbf{x}_n\} : x_1 = e_1^- \wedge \cdots \wedge x_n = e_n^- \gg \tau$, where e^- denotes the result of replacing all variables x_i in e by their '$^-$'superscripted version x_i^-.
- $[\![$ mode $X_1 = p_1, \ldots,$ mode $X_n = p_n :: q\,]\!] \triangleq [\![_\mathrm{R}\, \{X_1 \mapsto p_1, \ldots, X_n \mapsto p_n\} :: q\,]\!]$

2.2 Informal semantics

The behavior of χ processes is defined in terms of actions and delays[4]. Actions define instantaneous changes, where time does not change, to the values of variables. Delays involve the passing of time, where for all variables their trajectory as a function of time is defined. The valuation σ and the environment E, together define the variables that exist in the χ process and the variable classes to which they belong.

The variables are grouped into different classes with respect to the delay behavior and action behavior. With respect to the delay behavior, the variables are divided into the following classes:

- The discrete variables, the values of which remain constant while delaying.
- The continuous variables, the values of which change according to an absolutely continuous function[5] of time while delaying. The values of continuous variables are further restricted by delay predicates, that are usually in the form of differential algebraic equations.
- The dotted continuous variables, the values of which change according to an integrable, possibly discontinuous function of time while delaying.
- The algebraic variables, that behave in a similar way as continuous variables. The differences with continuous variables are that algebraic variables may change according to a discontinuous function of time, and that algebraic variables are not allowed to occur as dotted variables.
- The predefined variable 'time', that denotes the current time.

In χ, there are several means to change the value of a variable, depending on the class of variable. The main means for changing the value of a variable are the action predicate, for instantaneous changes, and the delay predicate, for the changes of variables over time.

2.2.1 Action predicates

An instantaneous change of the value of a discrete or continuous variable in χ is always connected to the execution of an action. In action predicates, the action is represented by a label. Other types of action are related to communication, which is treated below in the paragraph on parallelism. *Action predicate* $W : r \gg l_{\mathrm{a}}$ denotes instantaneous changes to the variables from set W, by means of an action labeled l_{a}, such that predicate r is satisfied. The predefined global variable time cannot be assigned. The non-jumping variables that are not mentioned in W remain unchanged, and the jumping variables, dotted continuous variables, and algebraic variables may obtain 'arbitrary' values, provided that the predicate r is satisfied and the process remains consistent.

---

[4] Formally, the behavior of χ processes is defined in terms of action transitions and time transitions. Informally, we use the term actions to refer to action transitions, and the term delays to refer to time transitions.

[5] A function $f(x)$ is *continuous* at $x \in X$ provided that for all $\varepsilon > 0$, there exists $\delta > 0$ so that $|x - y| \leq \delta$ implies $|f(x) - f(y)| \leq \varepsilon$. Roughly speaking, for single-valued functions this means that we can draw the graph of the function without taking the pencil of the paper. The class of absolutely continuous functions consists of continuous functions which are differentiable almost everywhere in Lebesgue sense. This class includes the differentiable functions.

A '$^-$' superscripted occurrence of a variable refers to the value of the variable in the extended valuation [6] prior to execution of the action predicate, and a normal (non-superscripted) occurrence of a variable refers to the value of that variable in the extended valuation that results from the execution of the action predicate. A predicate r is satisfied if evaluating the '$^-$' superscripted variables in the original extended valuation and evaluating the normal occurrences of the variables in the obtained extended valuation means that the predicate is true. The reason to use an extended valuation for evaluating action predicate r is that in such predicates also algebraic and dotted continuous variables may be used. Note that it can be the case that different instantaneous changes satisfy the predicate, this may result in non-determinism.

Note that the (multi-)assignment is not a primitive in χ, in contrast to for example in [9]. This is because action predicates are more expressive than assignments. Consider for example the action predicate $\{x\} : x \in [0,1] \gg \tau$, that changes the value of x to a value in the interval $[0,1]$. Also, the predicate of an action predicate may consist of a conjunction of implicit equations, e.g. $\{\mathbf{x}\} : f_1(\mathbf{x}^-, \mathbf{x}) = 0 \wedge \ldots \wedge f_n(\mathbf{x}^-, \mathbf{x}) = 0 \gg \tau$. The solution of such a system of equations, if present, need not always be expressible in an explicit form. The system may also have multiple solutions.

2.2.2 Delay predicates

In principle, continuous and algebraic variables change arbitrarily over time when delaying, although, depending on the class of the variable, they may have to respect some continuity requirements (see [20,5] for more details). A *delay predicate* u, usually in the form of a differential algebraic equation, restricts the allowed behavior of the continuous and algebraic variables in such a way that the value of the predicate remains true over time. Delay predicates of the form $x \geq e$, where x is a variable, e an expression, and where instead of $\geq$, also $\leq, >, <$ can be used, are comparable to invariants in hybrid automata.

2.2.3 Any delay operator

Besides the specification of delay by means of delay predicates, arbitrary delay can be described by means of the *any delay operator* $[p]$. The resulting behavior is such that arbitrary delays are allowed. When $[p]$ delays, p remains unchanged and its delay behavior is ignored. The action behavior of p remains unchanged in $[p]$.

2.2.4 Sequential composition

The *sequential composition* of two process terms $p;q$ behaves as process term p until p terminates, and then continues to behave as process term q.

2.2.5 Conditional

The *guarded process term* $b \rightarrow p$ can do whatever actions p can do under the condition that the guard b evaluates to true using the current extended valuation. The

[6] An extended valuation contains in addition to the values of the discrete and continuous variables and the variable time also the values of the dotted variables and the algebraic variables.

guarded process term can delay according to p under the condition that for the intermediate extended valuations during the delay, the guard b holds. The guarded process term can perform arbitrary delays under the condition that for the intermediate valuations during the delay, possibly excluding the first and last valuation, the guard b does not hold.

2.2.6 Choice

The *alternative composition operator* $[]$ allows a non-deterministic choice between different actions of a process. With respect to time behavior, the participants in the alternative composition have to synchronize. This means that the trajectories of the variables have to be agreed upon by both participants. This means that $[]$ is a strong time-deterministic [21] choice operator.

2.2.7 Parallelism

Parallelism can be specified by means of the *parallel composition operator* $\|$. Parallel processes interact by means of shared variables or by means of synchronous point-to-point communication/synchronization via a channel. Channels are denoted as labels (identifiers). The parallel composition $p \| q$ synchronizes the time behavior of p and q, interleaves the action behavior (including the instantaneous changes of variables) of p and q, and synchronizes matching send and receive actions. The synchronization of time behavior means that only the time behaviors that are allowed by both p and q are allowed by their parallel composition. The consistent equation semantics of χ enforces that actions by p (or q) are allowed only if the values of the variables before and after the actions are consistent with the other process term q (or p). This means, among others, that the 'active' delay predicates of q must hold before and after execution of an action by p and vice-versa.

By means of the *send process term* $h\,!!\,e_1, \ldots, e_n$, for $n \geq 1$, the values of expressions $e_1, \ldots, e_n$ (evaluated w.r.t. the extended valuation) are sent via channel h. For $n = 0$, this reduces to $h\,!!$ and nothing is sent via the channel. By means of the *receive process term* $h\,??\,x_1, \ldots, x_n$, for $n \geq 1$, values for $x_1, \ldots, x_n$ are received from channel h. We assume that all variables in the sequence $\mathbf{x}_n$ are syntactically different: $x_i \equiv x_j \implies i = j$. For $n = 0$, this reduces to $h\,??$, and nothing is received via the channel. Communication in χ is the sending of values by one parallel process via a channel to another parallel process, where the received values (if any) are stored in variables. For communication, the acts of sending and receiving (values) have to take place in different parallel processes at the same moment in time. The case that no values are sent and received is called synchronization instead of communication.

In order to be able to model open systems (i.e. systems that interface with the environment), it is necessary not to enforce communication via the external channels of the model (e.g. the channels that send or receive from the environment). For communication via internal channels, however, the communication of matching send and receive actions, often is not only an option, but an obligation. In such models, the separate occurrence of the send action and the receive action via an internal channel is undesired. The *encapsulation operator* ∂_A, where $A \subseteq \mathcal{A} \setminus \{\tau\}$ is a set of actions ($\mathcal{A}$ is the set of all possible actions and τ is the predefined internal action), is introduced to block the actions from the set A. In order to assure that,

for internal channels, only the synchronous execution of matching send and receive actions takes place, one can simply put all send and receive actions via internal channels in the set A.

In principle the channels in χ are non-urgent. This means that communication does not necessarily take place as soon as possible. In order to describe also urgent channels, the *urgent communication operator* $\upsilon_H(p)$, where $H \subseteq \mathcal{H}$ is a set of channel labels, ensures that p can only delay in case no communication or synchronization of send and receive actions via a channel from H is possible.

Note that a different kind of urgency can be achieved by means of undelayable process terms. The χ semantics ensures that actions of undelayable process terms have priority over delays. For example in $\dot{x} = 1 \parallel x := 1$ and $\dot{x} = 1 \,[]\, x := 1$, the assignment cannot delay. Therefore, it must be executed before a delay is possible. Also in $h\,!! \parallel \dot{x} = 1$, or $h\,!! \parallel h\,??$, or $h\,!! \parallel [h\,??]$, the parallel composition cannot delay because $h\,!!$ cannot delay. Therefore, a send action must be executed before a delay may be possible. Process term $[h\,!!] \parallel [h\,??]$, however, can do a communication action (or send or receive action), but it can also delay. To enforce the synchronization, the encapsulation operator is used; to enforce this as soon as possible, the urgent communication operator is used: $\upsilon_{\{h\}}([h\,!!] \parallel [h\,??])$.

2.2.8 Recursive definitions

Process term X denotes a recursion variable (identifier) that is defined either in the environment of the process, or in a recursion scope operator process term $[\![_{\mathrm{R}} \ldots :: p\,]\!]$, see subsubsection 2.2.9. Among others, it is used to model repetition. Recursion variable X can do whatever the process term of its definition can do.

2.2.9 Hierarchical modeling

To support the hierarchical modeling of systems, it is convenient to allow local declarations of variables. For this purpose, the *variable scope operator* process term $[\![\, D :: p\,]\!]$ is introduced, where D denotes declarations of local channels, local variables and local modes. It is allowed that the local variables have been declared on a more global level also. Any occurrence of a variable in process term p that is declared in D refers to the local variable and not to any more global declaration of the same variable name.

For similar purposes, local recursive definitions can be declared by means of a *recursion scope* process term $[\![_{\mathrm{R}}\, R :: p\,]\!]$. The recursion scope process term $[\![_{\mathrm{R}}\, R :: p\,]\!]$ is used to declare local recursion definitions $R \subseteq \mathcal{R}$.

2.3 Transitions

In [5,20] the formal semantics of χ is defined. The state of a χ process is defined as a tuple $\langle p, \sigma, E\rangle$ consisting of a process term p, a valuation σ of all data variables, and an environment E that contains, amongst others, definitions for all unbound recursion variables in p. The semantics then defines which changes in the state are possible, denoted by three kinds of arrows. The first kind, $\langle p, \sigma, E\rangle \xrightarrow{\xi, a, \xi'} \langle p', \sigma', E'\rangle$ or $\langle p, \sigma, E\rangle \xrightarrow{\xi, a, \xi'} \langle \surd, \sigma', E'\rangle$, denote a state change due to an action a, were ξ and ξ' show the change in the data resulting in a new process term or in termination

$(\surd)$. The second, $\langle p, \sigma, E\rangle \xmapsto{t,\rho} \langle p', \sigma', E'\rangle$, represents the passage of time t, where ρ is a function representing the flow of the data variables. Finally, the third arrow $\langle p, \sigma, E\rangle \overset{\xi}{\rightsquigarrow}$, denotes that an observation ξ on the data is consistent with the state $\langle p, \sigma, E\rangle$, without changing that state.

3 Syntax of the normal form of χ

The syntax of the normal form of the χ language is defined in terms of a recursion scope, nested in a scope operator by the following grammar for process terms $p_{\mathrm{Nt}} \in \mathcal{P}_{\mathrm{Nt}}$:

$$p_{\mathrm{Nt}} ::= \quad [\![\ D :: p_{\mathrm{N}}\ ]\!]$$

The syntax for the process terms p_{N} defines the set of process terms $\mathcal{P}_{\mathrm{N}}$:

$$p_{\mathrm{N}} ::= \quad [\![_{\mathrm{R}}\, R_{\mathrm{n}} :: X\ ]\!] \quad \text{satisfying the condition: } X \in \mathrm{dom}(R_{\mathrm{n}}),$$

where R_{n} is a recursion definition of the form $R_{\mathrm{n}} : \mathcal{X} \to \mathcal{P}_{\mathrm{n}}$. The set of all recursion definitions of the form R_{n} is denoted as $\mathcal{R}_{\mathrm{n}}$. Note that the subscript 'n' of process terms $p_{\mathrm{n}...}$ refers to syntax definitions that are part of the normal form.

The syntax for the process terms p_{n}, p_{ng}, defines the respective sets $\mathcal{P}_{\mathrm{n}}$, $\mathcal{P}_{\mathrm{ng}}$ of all such process terms:

$$p_{\mathrm{n}} ::= \quad p_{\mathrm{ng}} \ \mid\ p_{\mathrm{n}} \,[]\, p_{\mathrm{n}}$$

$$p_{\mathrm{ng}} ::= \quad u \ \mid\ p_{\mathrm{act}} \ \mid\ [p_{\mathrm{act}}] \ \mid\ p_{\mathrm{act}};\, X \ \mid\ [p_{\mathrm{act}}];\, X \ \mid\ b \to p_{\mathrm{ng}}$$

It can be shown that $\mathcal{P}_{\mathrm{N}} \subset \mathcal{P}_{\mathrm{s}}$. Note that nested guards are allowed in normal forms, e.g. $b \to b' \to h \,!!\, \mathbf{e}_n$. One might expect that nested guards such as $b \to b' \to p$ can be replaced by a single guard as follows $b \to b' \to p \leftrightarrows b \wedge b' \to p$. As shown in [20] this is not the case. Therefore nested guards are necessary in normal forms.

In the remainder of this paper, the following notations are used: $\mathcal{B}$ denotes all boolean expressions, and $p_{\mathrm{t}} \in \mathcal{P}_{\mathrm{t}}$, $p, q \in \mathcal{P}$, $p_{\mathrm{s}} \in \mathcal{P}_{\mathrm{s}}$, $p_{\mathrm{Nt}} \in \mathcal{P}_{\mathrm{Nt}}$, $p_{\mathrm{n}}, q_{\mathrm{n}} \in \mathcal{P}_{\mathrm{n}}$, $R, S \in \mathcal{R}$, $R_{\mathrm{n}}, S_{\mathrm{n}} \in \mathcal{R}_{\mathrm{n}}$, and $X, Y, Z \in \mathcal{X}$. Furthermore, the union on partial functions is defined as follows, if f and g are functions with $\mathrm{dom}(f) \cap \mathrm{dom}(g) = \emptyset$, then $f \cup g$ denotes the unique function h with $\mathrm{dom}(h) = \mathrm{dom}(f) \cup \mathrm{dom}(g)$ satisfying the condition: for each $c \in \mathrm{dom}(h)$, if $c \in \mathrm{dom}(f)$ then $h(c) = f(c)$, and $h(c) = g(c)$ otherwise.

4 Linearization algorithm

The normal form of a process term:

$$p_{\mathrm{t}} = [\![\ D :: p_{\mathrm{s}}\ ]\!]$$

is defined as the process term:

$$p_{\mathrm{Nt}} = [\![\ D :: N(p_{\mathrm{s}})\ ]\!]$$

where $N : \mathcal{P} \to \mathcal{P}_{\mathrm{N}}$. The linearization function N will be defined for all process terms $p \in \mathcal{P}$, because the syntax $\mathcal{P}_{\mathrm{s}}$ refers to process terms with syntax $\mathcal{P}$ (in recursion

definitions), and $\mathcal{P}_\mathrm{s} \subset \mathcal{P}$. Note, however, that $N(p)$ returns a complete scope when $p \in \mathcal{P}_\mathrm{s}$, but may return an incomplete recursion scope when $p \notin \mathcal{P}_\mathrm{s}$. This is especially important for the bisimilarity proofs of sequential composition, parallel composition and encapsulation (which only operate on p_s).

The definition of $N(p)$ is recursive over the structure of p, and to each element of this structure one subsection is dedicated. This section then contains a definition for $N(p)$, as well as an inductive argument proving that the function N is well-defined, that $N(p)$ is in normal form (i.e. $N(p) \in \mathcal{P}_\mathrm{n}$), and that $N(p)$ is stateless bisimilar [5,20] to p (denoted $N(p) \leftrightarrows p$).

Apart from some specific properties for certain operators that are introduced locally in the subsections, the following properties on recursion scopes are used to prove bisimilarity of $N(p)$ and p.

S1	p	$\leftrightarrows$	$[\![_\mathrm{R}\, \emptyset :: p \,]\!]$	
S2	$[\![_\mathrm{R}\, R :: p \,]\!]$	$\leftrightarrows$	$[\![_\mathrm{R}\, S \cup R :: p \,]\!]$	for $\mathrm{dom}(R) \cap \mathrm{dom}(S) = \emptyset$, $\mathrm{rvar}(p) \cap \mathrm{dom}(S) = \emptyset$, and $\mathrm{rvar}(R) \cap \mathrm{dom}(S) = \emptyset$.
S3	$[\![_\mathrm{R}\, R :: p \,]\!]$	$\leftrightarrows$	$[\![_\mathrm{R}\, R :: p[R(X)/X] \,]\!]$	for $X \in \mathrm{dom}(R)$
S4	$[\![_\mathrm{R}\, R :: [\![_\mathrm{R}\, S :: p \,]\!]]\!]$	$\leftrightarrows$	$[\![_\mathrm{R}\, R \cup S :: p \,]\!]$	for $\mathrm{dom}(R) \cap \mathrm{dom}(S) = \emptyset$, and $\mathrm{rvar}(R) \cap \mathrm{dom}(S) = \emptyset$.

In the bisimilarity proofs for sequential and parallel composition, the following theorem regarding the semantics of χ is used.

Theorem 4.1 *For a process $p_\mathrm{n} \in \mathcal{P}_\mathrm{n}$, a recursive specification $R_\mathrm{n} \in \mathcal{R}_\mathrm{n}$ in normal form, and recursion variable $X \in \mathrm{dom}(R_\mathrm{n})$, we find*

- $\langle p_\mathrm{n}, \sigma, E\rangle \overset{t,\rho}{\longmapsto} \langle p', \sigma', E'\rangle$ *implies* $p_\mathrm{n} = p'$,
- $\langle [\![_\mathrm{R}\, R_\mathrm{n} :: X \,]\!], \sigma, E\rangle \overset{t,\rho}{\longmapsto} \langle p, \sigma', E'\rangle$ *implies* $p = [\![_\mathrm{R}\, R_\mathrm{n} :: R_\mathrm{n}(X) \,]\!]$,
- $\langle p_\mathrm{n}, \sigma, E\rangle \xrightarrow{\xi,a,\xi'} \langle p', \sigma', E'\rangle$ *implies* $p' = X'$ *for some* $X' \in \mathrm{rvar}(p_\mathrm{n})$, *and*
- $\langle [\![_\mathrm{R}\, R_\mathrm{n} :: X \,]\!], \sigma, E\rangle \xrightarrow{\xi,a,\xi'} \langle p, \sigma', E'\rangle$ *implies* $p = [\![_\mathrm{R}\, R_\mathrm{n} :: X' \,]\!]$ *for some* $X' \in \mathrm{rvar}(R_\mathrm{n}(X))$.

Proof. Straightforward from the semantics of χ, with induction on the structure of normal forms. □

4.1 Atomic process terms

The normal form of an atomic process term is a recursion scope with a recursion definition that only defines a mapping from a recursion variable to the atomic process term. The normal form of a delayable atomic process term is similar, except

for the delayable delay predicate, which uses the property $[u] \leftrightarrow \text{true}$ (see [5,20]).

$$\begin{aligned} N(p_{\text{atom}}) &= [\![_{\text{R}}\, \{X \mapsto p_{\text{atom}}\} :: X\,]\!] \\ N([u]) &= [\![_{\text{R}}\, \{X \mapsto \text{true}\} :: X\,]\!] \\ N([p_{\text{act}}]) &= [\![_{\text{R}}\, \{X \mapsto [p_{\text{act}}]\} :: X\,]\!] \end{aligned}$$

Well-definedness is guaranteed since there is no mention of recursion for the atomic process terms. Furthermore, righthand sides are all in normal form, so $N(p_{\text{atom}}), N([u]), N([p_{\text{act}}]) \in \mathcal{P}_{\text{n}}$. Bisimilarity of these terms is trivial, using the properties mentioned in the beginning of this section, as an example we will treat $N([u])$. Note, that to use S2, we need (and trivially have) $X \notin \text{rvar}(\text{true})$.

$$\begin{aligned} N([u]) &= [\![_{\text{R}}\, \{X \mapsto \text{true}\} :: X\,]\!] && \text{by definition} \\ &\leftrightarrow [\![_{\text{R}}\, \{X \mapsto \text{true}\} :: \text{true}\,]\!] && \text{using S3} \\ &\leftrightarrow [\![_{\text{R}}\, \emptyset :: \text{true}\,]\!] && \text{using S2} \\ &\leftrightarrow \text{true} && \text{using S1} \\ &\leftrightarrow [u] && \text{using the property above} \end{aligned}$$

4.2 Guard operator

The recursion definition in the normal form of a process term p guarded by guard b is the union of the recursion definition of the normal form of p, and a new recursion definition that defines the process term $R_{\text{n}}(Y)$ guarded by guard b. Note, that the guard b is distributed over the alternative composition operators occurring in $R_{\text{n}}(Y)$ by means of function $T_{\text{g}} : \mathcal{B} \times \mathcal{P}_{\text{n}} \to \mathcal{P}_{\text{n}}$.

$$\begin{gathered} N(b \to p) = [\![_{\text{R}}\, R_{\text{n}} \cup \{X \mapsto T_{\text{g}}(b, R_{\text{n}}(Y))\} :: X\,]\!] \\ \text{where } N(p) = [\![_{\text{R}}\, R_{\text{n}} :: Y\,]\!],\ X \notin \text{dom}(R_{\text{n}}) \end{gathered}$$

$$T_{\text{g}}(b, p_{\text{ng}}) = b \to p_{\text{ng}} \qquad T_{\text{g}}(b, p_{\text{n}} \mathbin{[\!]} q_{\text{n}}) = T_{\text{g}}(b, p_{\text{n}}) \mathbin{[\!]} T_{\text{g}}(b, q_{\text{n}})$$

Since p is a strict subterm of $b \to p$, well-definedness of $N(b \to p)$ depends only on well-definedness of T_{g}, which follows by observing that the recursion in its righthand side only makes use of strict subterms of the lefthand side. Clearly, $N(b \to p) \in \mathcal{P}_{\text{n}}$ if $T_{\text{g}}(b, p_{\text{n}}) \in \mathcal{P}_{\text{n}}$, which follows by induction on the structure of p_{n}. Bisimilarity of $b \to p$ and $N(b \to p)$ follows inductively from the properties in the beginning of this section, and in addition the distribution of guards over the recursion scope:

$$\text{DGS} \quad [\![_{\text{R}}\, R :: b \to p\,]\!] \leftrightarrow b \to [\![_{\text{R}}\, R :: p\,]\!]$$

and the following lemma.

Lemma 4.2 $T_{\text{g}}(b, p) \leftrightarrow b \to p$

Proof. From [20] we have the property $b \to (p \mathbin{[\!]} q) \leftrightarrow b \to p \mathbin{[\!]} b \to q$. The rest follows with induction on the structure of p. □

$$
\begin{aligned}
N(b \to p) &= [\!|_{\mathrm{R}}\, R_{\mathrm{n}} \cup \{X \mapsto T_{\mathrm{g}}(b, R_{\mathrm{n}}(Y))\} :: X \,|\!] && \text{by definition}\\
&\leftrightarrows [\!|_{\mathrm{R}}\, R_{\mathrm{n}} \cup \{X \mapsto T_{\mathrm{g}}(b, R_{\mathrm{n}}(Y))\} :: T_{\mathrm{g}}(b, R_{\mathrm{n}}(Y)) \,|\!] && \text{using S3}\\
&\leftrightarrows [\!|_{\mathrm{R}}\, R_{\mathrm{n}} :: T_{\mathrm{g}}(b, R_{\mathrm{n}}(Y)) \,|\!] && \text{using S2}\\
&\leftrightarrows [\!|_{\mathrm{R}}\, R_{\mathrm{n}} :: b \to R_{\mathrm{n}}(Y) \,|\!] && \text{using Lemma 4.2}\\
&\leftrightarrows b \to [\!|_{\mathrm{R}}\, R_{\mathrm{n}} :: R_{\mathrm{n}}(Y) \,|\!] && \text{using DGS}\\
&= b \to N(p) && \text{by assumption}\\
&\leftrightarrows b \to p && \text{by induction}
\end{aligned}
$$

4.3 Sequential composition

This section defines the normal form of p_{s} sequentially followed by q. The recursion variable that defines the normal form of q is sequentially appended to all terminating process terms in the recursion definition of the normal form of p_{s} by means of function ap : $\mathcal{P}_{\mathrm{n}} \times \mathcal{X} \to \mathcal{P}_{\mathrm{n}}$. Terminating process terms are process terms that can do an action transition that results in the termination of the process; i.e. in normal form the terminating process terms are the p_{act} and $[p_{\mathrm{act}}]$ terms that are not sequentially followed by a recursion variable. This results in a new (incomplete) recursion definition with the same domain as the recursion definition of the normal form of p_{s}, only the range is different.

This new recursion definition joined with the recursion definition of the normal form of q result in the (complete) recursion definition of the normal form of the sequential composition of the process terms p_{s} and q.

$$
\begin{gathered}
N(p_{\mathrm{s}}; q) = [\!|_{\mathrm{R}}\, \{Z \mapsto \mathrm{ap}(R_{\mathrm{n}}(Z), Y) \mid Z \in \mathrm{dom}(R_{\mathrm{n}})\} \cup S_{\mathrm{n}} :: X \,|\!]\\
\text{where } N(p_{\mathrm{s}}) = [\!|_{\mathrm{R}}\, R_{\mathrm{n}} :: X \,|\!],\ N(q) = [\!|_{\mathrm{R}}\, S_{\mathrm{n}} :: Y \,|\!],\ \mathrm{dom}(R_{\mathrm{n}}) \cap \mathrm{dom}(S_{\mathrm{n}}) = \emptyset
\end{gathered}
$$

The case where the second process term is a recursion variable is treated separately, as $q \in \mathcal{P}$ and $X \notin \mathcal{P}$. The recursion variable Y is appended to all terminating process terms in the recursion definition of the normal form of p_{s}. Note that in this case the recursion definition of the normal form is incomplete, as $Y \notin \mathrm{dom}(R_{\mathrm{n}})$.

$$
\begin{gathered}
N(p_{\mathrm{s}}; Y) = [\!|_{\mathrm{R}}\, \{Z \mapsto \mathrm{ap}(R_{\mathrm{n}}(Z), Y) \mid Z \in \mathrm{dom}(R_{\mathrm{n}})\} :: X \,|\!]\\
\text{where } N(p_{\mathrm{s}}) = [\!|_{\mathrm{R}}\, R_{\mathrm{n}} :: X \,|\!],\ Y \notin \mathrm{dom}(R_{\mathrm{n}})
\end{gathered}
$$

For each righthand side in a recursion definition, the recursion variable Y is appended to the terminating process terms defining it, by means of function ap. If a definition is non-terminating, i.e. a process term sequentially composed with a recursion variable X, it is returned unchanged. This behavior is correct as only tail-recursion is allowed; the recursion variable Y is appended to all terminating

process terms in the definition of X.

$$\begin{aligned}
&\mathrm{ap}(u, Y) &&= u && \mathrm{ap}(p_{\mathrm{act}}, Y) &&= p_{\mathrm{act}}; Y\\
&\mathrm{ap}([p_{\mathrm{act}}], Y) &&= [p_{\mathrm{act}}]; Y && \mathrm{ap}(p_{\mathrm{act}}; X, Y) &&= p_{\mathrm{act}}; X\\
&\mathrm{ap}([p_{\mathrm{act}}]; X, Y) &&= [p_{\mathrm{act}}]; X && \mathrm{ap}(b \to p_{\mathrm{ng}}, Y) &&= b \to \mathrm{ap}(p_{\mathrm{ng}}, Y)\\
&\mathrm{ap}(p_{\mathrm{n}} \mathbin{[\!]} q_{\mathrm{n}}, Y) &&= \mathrm{ap}(p_{\mathrm{n}}, Y) \mathbin{[\!]} \mathrm{ap}(q_{\mathrm{n}}, Y)
\end{aligned}$$

The definition of $N(p_{\mathrm{s}};\ q)$ makes recursive use of $N(p_{\mathrm{s}})$ and $N(q)$, the definition of $N(p_{\mathrm{s}};\ Y)$ makes only use of $N(p_{\mathrm{s}})$. Observe that p_{s} and q are strict subterms of $p_{\mathrm{s}};\ q$, and p_{s} is a strict subterm of $p_{\mathrm{s}};\ Y$. Therefore, well-definedness of N is reduced to well-definedness of ap, which follows using the observation that recursion in the righthand sides only makes use of strict subterms of lefthand sides. Furthermore, $N(p_{\mathrm{s}};\ q) \in \mathcal{P}_{\mathrm{n}}$ if $\{Z \mapsto \mathrm{ap}(R_{\mathrm{n}}(Z), Y) \mid Z \in \mathrm{dom}(R_{\mathrm{n}})\} \in \mathcal{R}_{\mathrm{n}}$. The latter follows if $\mathrm{ap}(p_{\mathrm{n}}, Y) \in \mathcal{P}_{\mathrm{n}}$, which is proven by induction on the structure of p_{n}. The bisimilarity $N(p_{\mathrm{s}};\ q) \leftrightarrows p_{\mathrm{s}};\ q$ is hard to prove using properties. But, by induction, we find that it is sufficient to give a witnessing bisimulation relation $\mathcal{D}$ for $N(p_{\mathrm{s}});\ N(q) \leftrightarrows N(p_{\mathrm{s}};\ q)$ instead. That the relation $\mathcal{D}$ given below is indeed a bisimulation relation is straightforward but tedious to verify. Note, that we relied on Theorem 4.1 and on completeness of R_{n} (but not of S_{n}!), while constructing and proving this bisimulation relation.

$$\begin{aligned}
\mathcal{D} = \{&(\llbracket_{\mathrm{R}}\ R_{\mathrm{n}} :: X\ \rrbracket\ ;\ \llbracket_{\mathrm{R}}\ S_{\mathrm{n}} :: Y\ \rrbracket\\
&,\ \llbracket_{\mathrm{R}}\ \{Z \mapsto \mathrm{ap}(R_{\mathrm{n}}(Z), Y) \mid Z \in \mathrm{dom}(R_{\mathrm{n}})\} \cup S_{\mathrm{n}} :: X\ \rrbracket)\\
&\quad \mid R_{\mathrm{n}}, S_{\mathrm{n}} \in \mathcal{R}_{\mathrm{n}}, \mathrm{dom}(S_{\mathrm{n}}) \cap \mathrm{dom}(R_{\mathrm{n}}) = \emptyset, X \in \mathrm{dom}(R_{\mathrm{n}}), Y \in \mathrm{dom}(S_{\mathrm{n}})\}\\
&\cup \{(\llbracket_{\mathrm{R}}\ R_{\mathrm{n}} :: R_{\mathrm{n}}(X)\ \rrbracket\ ;\ \llbracket_{\mathrm{R}}\ S_{\mathrm{n}} :: Y\ \rrbracket\\
&,\ \llbracket_{\mathrm{R}}\ \{Z \mapsto \mathrm{ap}(R_{\mathrm{n}}(Z), Y) \mid Z \in \mathrm{dom}(R_{\mathrm{n}})\} \cup S_{\mathrm{n}} :: \mathrm{ap}(R_{\mathrm{n}}(X), Y)\ \rrbracket)\\
&\quad \mid R_{\mathrm{n}}, S_{\mathrm{n}} \in \mathcal{R}_{\mathrm{n}}, \mathrm{dom}(S_{\mathrm{n}}) \cap \mathrm{dom}(R_{\mathrm{n}}) = \emptyset, X \in \mathrm{dom}(R_{\mathrm{n}}), Y \in \mathrm{dom}(S_{\mathrm{n}})\}\\
&\cup \{(\llbracket_{\mathrm{R}}\ S_{\mathrm{n}} :: x\ \rrbracket\\
&,\ \llbracket_{\mathrm{R}}\ \{Z \mapsto \mathrm{ap}(R_{\mathrm{n}}(Z), Y) \mid Z \in \mathrm{dom}(R_{\mathrm{n}})\} \cup S_{\mathrm{n}} :: x\ \rrbracket)\\
&\quad \mid R_{\mathrm{n}}, S_{\mathrm{n}} \in \mathcal{R}_{\mathrm{n}}, \mathrm{dom}(S_{\mathrm{n}}) \cap \mathrm{dom}(R_{\mathrm{n}}) = \emptyset, \mathrm{rvar}(p) \cap \mathrm{dom}(R_{\mathrm{n}}) = \emptyset\}
\end{aligned}$$

4.4 Alternative composition

The normal form of the alternative composition of the process terms p and q is a recursion scope containing the union of the recursion definitions of the normal forms of p and q, and a new recursion definition that defines the alternative composition of Y and Z. Since we only allow tail-recursion we cannot introduce the equation $X \mapsto Y \mathbin{[\!]} Z$, therefore we use $X \mapsto R_{\mathrm{n}}(Y) \mathbin{[\!]} R_{\mathrm{n}}(Z)$.

$$\begin{aligned}
N(p \mathbin{[\!]} q) = \llbracket_{\mathrm{R}}\ R_{\mathrm{n}} \cup S_{\mathrm{n}} \cup \{X \mapsto R_{\mathrm{n}}(Y) \mathbin{[\!]} S_{\mathrm{n}}(X)\} :: X\ \rrbracket,\\
\text{where } N(p) = \llbracket_{\mathrm{R}}\ R_{\mathrm{n}} :: Y\ \rrbracket,\ N(q) = \llbracket_{\mathrm{R}}\ S_{\mathrm{n}} :: Z\ \rrbracket,\\
\mathrm{dom}(R_{\mathrm{n}}) \cap \mathrm{dom}(S_{\mathrm{n}}) = \emptyset,\ X \notin \mathrm{dom}(R_{\mathrm{n}} \cup S_{\mathrm{n}})
\end{aligned}$$

The observation that p and q are strict subterms of $p \mathbin{[\!]} q$ is sufficient to guarantee

well-definedness of $N(p \,[\!]\, q)$. By induction on the structure of p and q, we may conclude that the righthand side of the definition of $N(p \,[\!]\, q)$ is indeed in normal form. Finally, bisimilarity of $p \,[\!]\, q$ and $N(p \,[\!]\, q)$ follows from the properties given in the beginning of this section, and in addition the distribution of alternative composition of recursion scopes.

$$\text{DAS}\ [\![_{\text{R}}\ R :: p \,[\!]\, q\ ]\!] \ \leftrightarrows\ [\![_{\text{R}}\ R :: p\ ]\!] \,[\!]\, [\![_{\text{R}}\ R :: q\ ]\!]$$

4.5 Parallel composition

The normal form of the parallel composition of p_{s} and q_{s} is a recursion scope containing the union of the recursion definitions of the normal forms of p_{s} and q_{s}, and a new recursion definition that defines the parallel composition of each definition in the normal form of p_{s} with each definition in the normal form of q_{s} by means of function $\mu : \mathcal{R}_{\text{n}} \times \mathcal{R}_{\text{n}} \to \mathcal{R}_{\text{n}}$.

$$\begin{aligned} N(p_{\text{s}} \parallel q_{\text{s}}) &= [\![_{\text{R}}\ R_{\text{n}} \cup S_{\text{n}} \cup \mu(R_{\text{n}}, S_{\text{n}}) :: XY\ ]\!] \\ \text{where } N(p_{\text{s}}) &= [\![_{\text{R}}\ R_{\text{n}} :: X\ ]\!],\ N(q_{\text{s}}) = [\![_{\text{R}}\ S_{\text{n}} :: Y\ ]\!],\ \text{dom}(R_{\text{n}}) \cap \text{dom}(S_{\text{n}}) = \emptyset \\ & \forall_{X \in \text{dom}(R_{\text{n}}), Y \in \text{dom}(S_{\text{n}})}\ XY \notin \text{dom}(R_{\text{n}} \cup S_{\text{n}}) \end{aligned}$$

Here, $XY \in \mathcal{X}$ denotes the recursion variable that is obtained by the syntactical concatenation of the recursion variables represented by X and Y, meaning the parallel composition of the definitions of X and Y, $[\![_{\text{R}}\ R_{\text{n}} \cup S_{\text{n}} \cup \mu(R_{\text{n}}, S_{\text{n}}) :: XY\ ]\!]$ $\leftrightarrows [\![_{\text{R}}\ R_{\text{n}} :: X\ ]\!] \parallel [\![_{\text{R}}\ S_{\text{n}} :: Y\ ]\!]$.

By means of function μ the parallel composition of all definitions in the recursion definitions R_{n} and S_{n} is created. This creates all possible parallel compositions of definitions $R_{\text{n}}(X)$ and $S_{\text{n}}(Y)$. As parallel composition is not part of the syntax of the normal form, it needs to be eliminated. It is eliminated by replacing parallel process terms with process terms that model the behavior of the parallel composition.

The behavior of the parallel composition operator is defined as: the behavior with respect to action transitions of the parallel composition of process terms p and q is the interleaving of the behaviors with respect to action transitions of p and q. The parallel composition allows the synchronization of matching send and receive actions, which results in a communication action. This can be denoted by the communication process term. The time transitions of the parallel composition of process terms p and q have to synchronize. The behavior with respect to time transitions is the same as that of alternative composition. Therefore, the parallel composition of p_{n} and q_{n} can be rewritten as the alternative composition of the process terms in p_{n} and q_{n}, see functions $\alpha_l : \mathcal{P}_{\text{n}} \times \mathcal{X} \to \mathcal{P}_{\text{n}}$ and $\alpha_r : \mathcal{X} \times \mathcal{P}_{\text{n}} \to \mathcal{P}_{\text{n}}$, and the synchronization of send and receive process terms in p_{n} and q_{n}, see function $\gamma : \mathcal{P}_{\text{n}} \times \mathcal{P}_{\text{n}} \to \mathcal{P}_{\text{n}}$.

$$\begin{aligned} \mu(R_{\text{n}}, S_{\text{n}}) = \{ & XY \mapsto \alpha_l(R_{\text{n}}(X), Y) \,[\!]\, \alpha_r(X, S_{\text{n}}(Y)) \,[\!]\, \gamma(R_{\text{n}}(X), S_{\text{n}}(Y)) \\ & \mid X \in \text{dom}(R_{\text{n}}), Y \in \text{dom}(S_{\text{n}}) \,\} \end{aligned}$$

Function α_l returns a process term which models the allowed time transitions of the left process term, and the action transitions of the parallel composition if the

left process term is executed first. Delay predicates are returned unmodified. Terminating process terms, p_{act} and $[p_{\mathrm{act}}]$, continue with the process term of the right side, defined by Y. Non-terminating process terms, $p_{\mathrm{act}};X$ and $[p_{\mathrm{act}}];X$, continue as the parallel composition of the process term defined by X, and the process term of the right side, defined by Y; syntactically denoted as XY. Function α_l distributes over the guard operator and alternative composition operator. Note that the recursion definition for recursion variable XY is also created (for complete recursion definitions), see function μ.

$$\begin{aligned}
\alpha_l(u, Y) &= u\\
\alpha_l(p_{\mathrm{act}}, Y) &= p_{\mathrm{act}};Y\\
\alpha_l([p_{\mathrm{act}}], Y) &= [p_{\mathrm{act}}];Y\\
\alpha_l(p_{\mathrm{act}};X,\ Y) &= p_{\mathrm{act}};XY\\
\alpha_l([p_{\mathrm{act}}];X,\ Y) &= [p_{\mathrm{act}}];XY\\
\alpha_l(b \rightarrow p_{\mathrm{ng}}, Y) &= b \rightarrow \alpha_l(p_{\mathrm{ng}}, Y)\\
\alpha_l(p_{\mathrm{n}} \mathbin{[\!]} q_{\mathrm{n}}, Y) &= \alpha_l(p_{\mathrm{n}}, Y) \mathbin{[\!]} \alpha_l(q_{\mathrm{n}}, Y)
\end{aligned}$$

Function α_r returns a process term which models the allowed time transitions of the right process term, and the action transitions of the parallel composition if the right process term is executed first. The function α_r follows the same schema as function α_l, delay predicates are returned unmodified, terminating process terms continue as X, and non-terminating process terms as XY.

$$\begin{aligned}
\alpha_r(X, u) &= u\\
\alpha_r(X, p_{\mathrm{act}}) &= p_{\mathrm{act}};X\\
\alpha_r(X, [p_{\mathrm{act}}]) &= [p_{\mathrm{act}}];X\\
\alpha_r(X,\ p_{\mathrm{act}};Y) &= p_{\mathrm{act}};XY\\
\alpha_r(X,\ [p_{\mathrm{act}}];Y) &= [p_{\mathrm{act}}];XY\\
\alpha_r(X, b \rightarrow p_{\mathrm{ng}}) &= b \rightarrow \alpha_r(X, p_{\mathrm{ng}})\\
\alpha_r(X, p_{\mathrm{n}} \mathbin{[\!]} q_{\mathrm{n}}) &= \alpha_r(X, p_{\mathrm{n}}) \mathbin{[\!]} \alpha_r(X, q_{\mathrm{n}})
\end{aligned}$$

Parallel send and receive process terms with matching channels synchronize. This behavior is represented in normal forms by the communication process term $h\,!?\,\mathbf{x}_{\mathrm{n}} := \mathbf{e}_{\mathrm{n}}$. Note that the behavior of the parallel composition operator with respect to time transitions is completely modeled by the alternative composition of the process terms returned by α_l and α_r. The communication process terms may not influence time transitions, therefore the any delay operator is applied to all generated communication process terms. The transformation from parallel send and receive process terms to a delayable communication process term is defined by means of function γ. Before function γ is further specified, first some notations are introduced. The syntax for the process terms p_{ns}, p_{nr}, p_{ngs}, p_{ngr} defines the respective sets $\mathcal{P}_{\mathrm{ns}}$, $\mathcal{P}_{\mathrm{nr}}$, $\mathcal{P}_{\mathrm{ngs}}$, $\mathcal{P}_{\mathrm{ngr}}$ as follows:

$$p_{\mathrm{ns}} ::= h \mathbin{!!} \mathbf{e}_{\mathrm{n}} \mid [h \mathbin{!!} \mathbf{e}_{\mathrm{n}}] \mid h \mathbin{!!} \mathbf{e}_{\mathrm{n}};\ X \mid [h \mathbin{!!} \mathbf{e}_{\mathrm{n}}];\ X$$

$$p_{\mathrm{nr}} ::= h \mathbin{??} \mathbf{x}_{\mathrm{n}} \mid [h \mathbin{??} \mathbf{x}_{\mathrm{n}}] \mid h \mathbin{??} \mathbf{x}_{\mathrm{n}};\ X \mid [h \mathbin{??} \mathbf{x}_{\mathrm{n}}];\ X$$

$$p_{\mathrm{ngs}} ::= p_{\mathrm{ns}} \mid b \rightarrow p_{\mathrm{ngs}}$$

$$p_{\mathrm{ngr}} ::= p_{\mathrm{nr}} \mid b \rightarrow p_{\mathrm{ngr}}$$

Function $\mathrm{ch} : \mathcal{P}_{\mathrm{ngr}} \cup \mathcal{P}_{\mathrm{ngs}} \rightarrow \mathcal{H}$ is introduced to extract the channel from (guarded) send and receive process terms:

$$\begin{gathered}
\mathrm{ch}(h \mathbin{!!} \mathbf{e}_{\mathrm{n}}) = \mathrm{ch}([h \mathbin{!!} \mathbf{e}_{\mathrm{n}}]) = \mathrm{ch}(h \mathbin{!!} \mathbf{e}_{\mathrm{n}}; X) = \mathrm{ch}([h \mathbin{!!} \mathbf{e}_{\mathrm{n}}]; X) = h \\
\mathrm{ch}(h \mathbin{??} \mathbf{x}_{\mathrm{n}}) = \mathrm{ch}([h \mathbin{??} \mathbf{x}_{\mathrm{n}}]) = \mathrm{ch}(h \mathbin{??} \mathbf{x}_{\mathrm{n}}; X) = \mathrm{ch}([h \mathbin{??} \mathbf{x}_{\mathrm{n}}]; X) = h \\
\mathrm{ch}(b \rightarrow p_{\mathrm{ngs}}) = \mathrm{ch}(p_{\mathrm{ngs}}) \\
\mathrm{ch}(b \rightarrow p_{\mathrm{ngr}}) = \mathrm{ch}(p_{\mathrm{ngr}})
\end{gathered}$$

Now function γ is further specified. Matching send and receive process terms result in a delayable communication process term:

$$\begin{aligned}
\gamma(h \mathbin{!!} \mathbf{e}_{\mathrm{n}}, h \mathbin{??} \mathbf{x}_{\mathrm{n}}) &= \gamma([h \mathbin{!!} \mathbf{e}_{\mathrm{n}}], h \mathbin{??} \mathbf{x}_{\mathrm{n}}) = \gamma(h \mathbin{!!} \mathbf{e}_{\mathrm{n}}, [h \mathbin{??} \mathbf{x}_{\mathrm{n}}]) \\
&= \gamma([h \mathbin{!!} \mathbf{e}_{\mathrm{n}}], [h \mathbin{??} \mathbf{x}_{\mathrm{n}}]) = [h \mathbin{!?} \mathbf{x}_{\mathrm{n}} := \mathbf{e}_{\mathrm{n}}] \\
\gamma(h \mathbin{??} \mathbf{x}_{\mathrm{n}}, h \mathbin{!!} \mathbf{e}_{\mathrm{n}}) &= \gamma([h \mathbin{??} \mathbf{x}_{\mathrm{n}}], h \mathbin{!!} \mathbf{e}_{\mathrm{n}}) = \gamma(h \mathbin{??} \mathbf{x}_{\mathrm{n}}, [h \mathbin{!!} \mathbf{e}_{\mathrm{n}}]) \\
&= \gamma([h \mathbin{??} \mathbf{x}_{\mathrm{n}}], [h \mathbin{!!} \mathbf{e}_{\mathrm{n}}]) = [h \mathbin{!?} \mathbf{x}_{\mathrm{n}} := \mathbf{e}_{\mathrm{n}}]
\end{aligned}$$

If one of the matching send or receive process terms is non-terminating, then the communication process term continues with the process term defined by X or Y (depending on left or right process terms). If the matching send and receive process terms are both non-terminating, then the communication process terms continues as the parallel composition of the process term defined by X, and the process term defined by Y, syntactically denoted as XY.

$$\begin{aligned}
\gamma(p_{\mathrm{ns}}; X, q_{\mathrm{nr}}) &= \gamma(p_{\mathrm{ns}}, q_{\mathrm{nr}}); X \\
\gamma(q_{\mathrm{nr}}; X, p_{\mathrm{ns}}) &= \gamma(q_{\mathrm{nr}}, p_{\mathrm{ns}}); X \\
\gamma(p_{\mathrm{ns}}; X, q_{\mathrm{nr}}; Y) &= \gamma(p_{\mathrm{ns}}, q_{\mathrm{nr}}); XY \\
\gamma(q_{\mathrm{nr}}, p_{\mathrm{ns}}; Y) &= \gamma(q_{\mathrm{nr}}, p_{\mathrm{ns}}); Y \\
\gamma(p_{\mathrm{ns}}, q_{\mathrm{nr}}; Y) &= \gamma(p_{\mathrm{ns}}, q_{\mathrm{nr}}); Y \\
\gamma(q_{\mathrm{nr}}; X, p_{\mathrm{ns}}; Y) &= \gamma(q_{\mathrm{nr}}, p_{\mathrm{ns}}); XY \\
&\text{where } \mathrm{ch}(p_{\mathrm{ns}}) = \mathrm{ch}(q_{\mathrm{nr}})
\end{aligned}$$

If one of the matching send or receive process terms is guarded by b, then the communication process term is also guarded by b. If the matching send and receive process term are both guarded, then the communication process term is guarded by the conjunction of the guards:

$$\gamma(b \to p_{\mathrm{ngs}}, q_{\mathrm{nr}}) = b \to \gamma(p_{\mathrm{ngs}}, q_{\mathrm{nr}})$$
$$\gamma(q_{\mathrm{nr}}, b' \to p_{\mathrm{ngs}}) = b' \to \gamma(q_{\mathrm{nr}}, p_{\mathrm{ngs}})$$
$$\text{where } \mathrm{ch}(p_{\mathrm{ngs}}) = \mathrm{ch}(q_{\mathrm{nr}})$$
$$\gamma(p_{\mathrm{ns}}, b' \to q_{\mathrm{ngr}}) = b' \to \gamma(p_{\mathrm{ns}}, q_{\mathrm{ngr}})$$
$$\gamma(b \to q_{\mathrm{ngr}}, p_{\mathrm{ns}}) = b \to \gamma(q_{\mathrm{ngr}}, p_{\mathrm{ns}})$$
$$\text{where } \mathrm{ch}(p_{\mathrm{ns}}) = \mathrm{ch}(q_{\mathrm{ngr}})$$
$$\gamma(b \to p_{\mathrm{ngs}}, b' \to q_{\mathrm{ngr}}) = b \wedge b' \to \gamma(p_{\mathrm{ngs}}, q_{\mathrm{ngr}})$$
$$\gamma(b \to q_{\mathrm{ngr}}, b' \to p_{\mathrm{ngs}}) = b \wedge b' \to \gamma(q_{\mathrm{ngr}}, p_{\mathrm{ngs}})$$
$$\text{where } \mathrm{ch}(p_{\mathrm{ngs}}) = \mathrm{ch}(q_{\mathrm{ngr}}),$$

In all other cases no communication action is possible, therefore a delay predicate 'true' is returned. Note that we have: $p \mathbin{[\!]} \mathrm{true} \leftrightarrows p$. The first case is (guarded) send and receive process terms with non-matching channels:

$$\gamma(p_{\mathrm{ngs}}, q_{\mathrm{ngr}}) = \gamma(q_{\mathrm{ngr}}, p_{\mathrm{ngs}}) = \mathrm{true}$$
$$\text{where } \mathrm{ch}(p_{\mathrm{ngs}}) \neq \mathrm{ch}(q_{\mathrm{ngr}})$$

The second case is not to have a combination of a send and a receive process term:

$$\gamma(p_{\mathrm{ng}}, q_{\mathrm{ng}}) = \mathrm{true}$$
$$\text{where } \neg(p_{\mathrm{ng}} \in \mathcal{P}_{\mathrm{ngs}} \wedge q_{\mathrm{ng}} \in \mathcal{P}_{\mathrm{ngr}}) \wedge \neg(p_{\mathrm{ng}} \in \mathcal{P}_{\mathrm{ngr}} \wedge q_{\mathrm{ng}} \in \mathcal{P}_{\mathrm{ngs}})$$

Communication can only be determined for pairs of (guarded) process terms. Therefore alternative process terms are expanded until only pairs of (guarded) process terms remain.

$$\gamma(p_{\mathrm{n}} \mathbin{[\!]} p'_{\mathrm{n}},\ q_{\mathrm{n}}) = \gamma(p_{\mathrm{n}}, q_{\mathrm{n}}) \mathbin{[\!]} \gamma(p'_{\mathrm{n}}, q_{\mathrm{n}})$$
$$\gamma(p_{\mathrm{n}},\ q_{\mathrm{n}} \mathbin{[\!]} q'_{\mathrm{n}}) = \gamma(p_{\mathrm{n}}, q_{\mathrm{n}}) \mathbin{[\!]} \gamma(p_{\mathrm{n}}, q'_{\mathrm{n}})$$

The definition of $N(p_{\mathrm{s}} \parallel q_{\mathrm{s}})$ makes recursive use of $N(p_s)$ and $N(q_s)$. Observe that p_s and q_s are strict subterms of $p_s \parallel q_s$, so well-definedness of N is reduced to well-definedness of the function μ, which in turn reduces to well-definedness of α_l, α_r and γ. Well-definedness of the latter three functions is easily verified by observing that what appears in the recursions on the righthand sides are strict subterms of the lefthand side. Furthermore, $N(p_s \parallel q_s) \in \mathcal{P}_{\mathrm{n}}$ follows if $\mu(R_{\mathrm{n}}, S_{\mathrm{n}}) \in \mathcal{R}_{\mathrm{n}}$. The latter follows if $\alpha_l(p_{\mathrm{n}}, Y), \alpha_r(X, q_{\mathrm{n}}), \gamma(p_{\mathrm{n}}, q_{\mathrm{n}}) \in \mathcal{P}_{\mathrm{n}}$, which is proven by induction on the structure of p_{n} and q_{n}.

Bisimilarity $N(p_s \parallel q_s) \leftrightarrows p_s \parallel q_s$ is hard to prove using properties, so we give the witnessing bisimulation relation $\mathcal{D}$. That this is indeed a bisimulation relation is straightforward to verify. Note, that we used Theorem 4.1 and the completeness of R_{n} and S_{n} (which follows from $p_s, q_s \in \mathcal{P}_s$) while constructing and proving this bisimulation relation.

$$\begin{aligned}\mathcal{D} = \{&(|[\mathrm{R}\ R_\mathrm{n} :: X\ ]|||\ |[\mathrm{R}\ S_\mathrm{n} :: Y\ ]|\ ,\ |[\mathrm{R}\ R_\mathrm{n} \cup S_\mathrm{n} \cup \mu(R_\mathrm{n}, S_\mathrm{n}) :: XY\ ]|)\\ &\mid R_\mathrm{n}, S_\mathrm{n} \in \mathcal{R}_\mathrm{n}, \mathrm{dom}(R_\mathrm{n}) \cap \mathrm{dom}(S_\mathrm{n}) = \emptyset, X \in \mathrm{dom}(R_\mathrm{n}), Y \in \mathrm{dom}(S_\mathrm{n})\}\\ \cup\ \{&(|[\mathrm{R}\ R_\mathrm{n} :: R_\mathrm{n}(X)\ ]|||\ |[\mathrm{R}\ S_\mathrm{n} :: S_\mathrm{n}(Y)\ ]|\\ &,\ |[\mathrm{R}\ R_\mathrm{n} \cup S_\mathrm{n} \cup \mu(R_\mathrm{n}, S_\mathrm{n}) :: \alpha_l(R_\mathrm{n}(X), Y) \ [\!]\ \alpha_r(X, S_\mathrm{n}(Y)) \ [\!]\ \gamma(R_\mathrm{n}(X), S_\mathrm{n}(Y))\ ]|)\\ &\mid R_\mathrm{n}, S_\mathrm{n} \in \mathcal{R}_\mathrm{n}, \mathrm{dom}(R_\mathrm{n}) \cap \mathrm{dom}(S_\mathrm{n}) = \emptyset, X \in \mathrm{dom}(R_\mathrm{n}), Y \in \mathrm{dom}(S_\mathrm{n})\}\\ \cup\ \{&(|[\mathrm{R}\ R_\mathrm{n} :: X\ ]|||\ |[\mathrm{R}\ S_\mathrm{n} :: S_\mathrm{n}(Y)\ ]|, |[\mathrm{R}\ R_\mathrm{n} \cup S_\mathrm{n} \cup \mu(R_\mathrm{n}, S_\mathrm{n}) :: XY\ ]|)\\ &\mid R_\mathrm{n}, S_\mathrm{n} \in \mathcal{R}_\mathrm{n}, \mathrm{dom}(R_\mathrm{n}) \cap \mathrm{dom}(S_\mathrm{n}) = \emptyset, X \in \mathrm{dom}(R_\mathrm{n}), Y \in \mathrm{dom}(S_\mathrm{n})\}\\ \cup\ \{&(|[\mathrm{R}\ R_\mathrm{n} :: R_\mathrm{n}(X)\ ]|||\ |[\mathrm{R}\ S_\mathrm{n} :: Y\ ]|, |[\mathrm{R}\ R_\mathrm{n} \cup S_\mathrm{n} \cup \mu(R_\mathrm{n}, S_\mathrm{n}) :: XY\ ]|)\\ &\mid R_\mathrm{n}, S_\mathrm{n} \in \mathcal{R}_\mathrm{n}, \mathrm{dom}(R_\mathrm{n}) \cap \mathrm{dom}(S_\mathrm{n}) = \emptyset, X \in \mathrm{dom}(R_\mathrm{n}), Y \in \mathrm{dom}(S_\mathrm{n})\}\\ \cup\ \{&(|[\mathrm{R}\ U_\mathrm{n} :: Z\ ]|\ ,\ |[\mathrm{R}\ U_\mathrm{n} \cup V :: Z\ ]|)\\ &\mid U_\mathrm{n} \in \mathcal{R}_\mathrm{n}, V \in \mathcal{R}, \mathrm{dom}(U_\mathrm{n}) \cap \mathrm{dom}(V) = \emptyset, Z \in \mathrm{dom}(U_\mathrm{n})\}\\ \cup\ \{&(|[\mathrm{R}\ U_\mathrm{n} :: U_\mathrm{n}(Z)\ ]|\ ,\ |[\mathrm{R}\ U_\mathrm{n} \cup V :: U_\mathrm{n}(Z)\ ]|) \mid U_\mathrm{n} \in \mathcal{R}_\mathrm{n},\\ &V \in \mathcal{R}, \mathrm{dom}(U_\mathrm{n}) \cap \mathrm{dom}(V) = \emptyset, \mathrm{rvar}(U_\mathrm{n}(Z)) \cap \mathrm{dom}(V) = \emptyset, Z \in \mathrm{dom}(U_\mathrm{n})\}\end{aligned}$$

4.6 Recursion scope

The normal form of the recursion scope operator process term is defined in terms of linearization of the righthand sides of the recursion definition by means of function $N_\mathrm{R} : \mathcal{R} \to \mathcal{R}_\mathrm{n}$.

$$N(|[\mathrm{R}\ R :: X\ ]|) = |[\mathrm{R}\ N_\mathrm{R}(R) :: X\ ]|$$

In the second form $|[\mathrm{R}\ R :: p\ ]|$ of the recursion scope operator, the recursion definition of the normal form of p is joined with the linearized recursion definitions:

$$\begin{gathered}N(|[\mathrm{R}\ R :: p\ ]|) = |[\mathrm{R}\ N_\mathrm{R}(R) \cup S_\mathrm{n} :: X\ ]|\\ \text{where } N(p) = |[\mathrm{R}\ S_\mathrm{n} :: X\ ]|,\ \mathrm{dom}(N_\mathrm{R}(R)) \cap \mathrm{dom}(S_\mathrm{n}) = \emptyset\end{gathered}$$

The normal form of a recursion definition is the union of the recursion definition of the normal form of p with a recursion definition that defines a mapping from the original recursion variable X to the recursion variable that defines the normal form of p. Since we only allow tail-recursion we cannot use the equation $X \mapsto Y$, therefore we use $X \mapsto R_\mathrm{n}(Y)$.

$$\begin{gathered}N_\mathrm{R}(\emptyset) = \emptyset\\ N_\mathrm{R}(\{X \mapsto p\}) = R_\mathrm{n} \cup \{X \mapsto R_\mathrm{n}(Y)\}\\ \text{where } N(p) = |[\ R_\mathrm{n} :: Y\ ]|,\ X \notin \mathrm{dom}(R_\mathrm{n})\\ N_\mathrm{R}(R \cup S) = N_\mathrm{R}(R) \cup N_\mathrm{R}(S)\\ \text{where } \mathrm{dom}(N_\mathrm{R}(R)) \cap \mathrm{dom}(N_\mathrm{R}(S)) = \emptyset\end{gathered}$$

Note that the intermediate normal from can be an incomplete recursion scope. However the normal form of the whole recursion scope operator process term is complete.

Clearly, $N(|[\mathrm{R}\ R :: X\ ]|)$ and $N(|[\mathrm{R}\ R :: p\ ]|)$ are well-defined if N_R is well defined, in turn well-definedness of N_R may be concluded from the observation that all recursive

calls of N are made on righthand sides from R. Furthermore, $N(\llbracket_{\mathrm{R}}\, R :: X \,\rrbracket) \in \mathcal{P}_{\mathrm{n}}$ if $N_{\mathrm{R}}(R) \in \mathcal{R}_{\mathrm{n}}$, which follows by induction on the structure of R. Bisimilarity of terms is proven using induction and the properties in the beginning of this section, and additionally:

$$\text{S5} \quad \llbracket_{\mathrm{R}}\, \{Y \mapsto \llbracket_{\mathrm{R}}\, S :: X \,\rrbracket\} :: p \,\rrbracket \;\leftrightarrows\; \llbracket_{\mathrm{R}}\, S \cup \{Y \mapsto S(X)\} :: p \,\rrbracket$$

for $\mathrm{rvar}(p) \cap \mathrm{dom}(S) = \emptyset$.

Let $N(p) = \llbracket_{\mathrm{R}}\, S_{\mathrm{n}} :: Z \,\rrbracket$, $R = \{X_i \mapsto p_i \mid i \in I\}$, and $N(p_i) = \llbracket_{\mathrm{R}}\, R^i_{\mathrm{n}} :: Y_i \,\rrbracket$. Then,

$$\begin{aligned}
& N(\llbracket_{\mathrm{R}}\, R :: p \,\rrbracket) \\
= \;& \llbracket_{\mathrm{R}}\, \textstyle\bigcup_{i\in I}(R^i_{\mathrm{n}} \cup \{X_i \mapsto R^i_{\mathrm{n}}(Y_i)\}) \cup S_{\mathrm{n}} :: Z \,\rrbracket && \text{definition} \\
\leftrightarrows \;& \llbracket_{\mathrm{R}}\, \textstyle\bigcup_{i\in I}(R^i_{\mathrm{n}} \cup \{X_i \mapsto R^i_{\mathrm{n}}(Y_i)\}) :: \llbracket_{\mathrm{R}}\, S_{\mathrm{n}} :: Z \,\rrbracket\rrbracket && \text{using S4} \\
\leftrightarrows \;& \llbracket_{\mathrm{R}}\, \textstyle\bigcup_{i\in I}(R^i_{\mathrm{n}} \cup \{X_i \mapsto R^i_{\mathrm{n}}(Y_i)\}) :: N(p) \,\rrbracket && \text{definition} \\
\leftrightarrows \;& \llbracket_{\mathrm{R}}\, \textstyle\bigcup_{i\in I}(R^i_{\mathrm{n}} \cup \{X_i \mapsto R^i_{\mathrm{n}}(Y_i)\}) :: p \,\rrbracket && \text{induction} \\
\leftrightarrows \;& \llbracket_{\mathrm{R}}\, \textstyle\bigcup_{i\neq j}(R^i_{\mathrm{n}} \cup \{X_i \mapsto R^i_{\mathrm{n}}(Y_i)\}) :: \llbracket_{\mathrm{R}}\, R^j_{\mathrm{n}} \cup \{X_j \mapsto R^j_{\mathrm{n}}(Y_j)\} :: p \,\rrbracket\rrbracket && \text{using S4} \\
\leftrightarrows \;& \llbracket_{\mathrm{R}}\, \textstyle\bigcup_{i\neq j}(R^i_{\mathrm{n}} \cup \{X_i \mapsto R^i_{\mathrm{n}}(Y_i)\}) :: \llbracket_{\mathrm{R}}\, \{X_j \mapsto \llbracket_{\mathrm{R}}\, R^j_{\mathrm{n}} :: Y_j \,\rrbracket\} :: p \,\rrbracket\rrbracket && \text{using S5} \\
\leftrightarrows \;& \llbracket_{\mathrm{R}}\, \textstyle\bigcup_{i\neq j}(R^i_{\mathrm{n}} \cup \{X_i \mapsto R^i_{\mathrm{n}}(Y_i)\}) \cup \{X_j \mapsto \llbracket_{\mathrm{R}}\, R^j_{\mathrm{n}} :: Y_j \,\rrbracket\} :: p \,\rrbracket && \text{using S4} \\
& \text{and repeating the last three steps over all } j \in I \\
\leftrightarrows \;& \llbracket_{\mathrm{R}}\, \{X_i \mapsto \llbracket_{\mathrm{R}}\, R^i_{\mathrm{n}} :: Y_i \,\rrbracket \mid i \in I\} :: p \,\rrbracket \\
\leftrightarrows \;& \llbracket_{\mathrm{R}}\, \{X_i \mapsto N(p_i) \mid i \in I\} :: p \,\rrbracket && \text{definition} \\
\leftrightarrows \;& \llbracket_{\mathrm{R}}\, \{X_i \mapsto p_i \mid i \in I\} :: p \,\rrbracket && \text{induction} \\
\leftrightarrows \;& \llbracket_{\mathrm{R}}\, R :: p \,\rrbracket && \text{definition}
\end{aligned}$$

4.7 Encapsulation

The normal form of a process term p_{s} encapsulated with the action labels A is the encapsulation distributed over the recursion definition of the normal form of p_{s}, by means of function $T_{\mathrm{e}} : \mathcal{P}_{\mathrm{n}} \times \mathcal{A} \to \mathcal{P}_{\mathrm{n}}$.

$$N(\partial_A(p_{\mathrm{s}})) = \llbracket_{\mathrm{R}}\, \{Y \mapsto T_{\mathrm{e}}(R_{\mathrm{n}}(Y), A) \mid Y \in \mathrm{dom}(R_{\mathrm{n}})\} :: X \,\rrbracket$$
$$\text{where } N(p_{\mathrm{s}}) = \llbracket_{\mathrm{R}}\, R_{\mathrm{n}} :: X \,\rrbracket$$

An action process term is replaced by the deadlock process term δ, if the action label is in the set of action labels to encapsulate, otherwise an action process term is left unchanged. It is assumed that action encapsulation is based on channel names and not on the values sent or received; the notations $\mathrm{isa}(h, *)$, $\mathrm{ira}(h, *, *)$, $\mathrm{ca}(h, *)$

mean the respective action labels for all possible values.

$$T_{\mathrm{e}}(W : r \gg l_a, A) = \begin{cases} \delta & \text{if } l_a \in A \\ W : r \gg l_a & \text{if } l_a \notin A \end{cases}$$

$$T_{\mathrm{e}}(h \,!!\, \mathbf{e}_{\mathrm{n}}, A) = \begin{cases} \delta & \text{if } \mathrm{isa}(h, *) \in A \\ h \,!!\, \mathbf{e}_{\mathrm{n}} & \text{if } \mathrm{isa}(h, *) \notin A \end{cases}$$

$$T_{\mathrm{e}}(h \,??\, \mathbf{x}_{\mathrm{n}}, A) = \begin{cases} \delta & \text{if } \mathrm{ira}(h, *, *) \in A \\ h \,??\, \mathbf{x}_{\mathrm{n}} & \text{if } \mathrm{ira}(h, *, *) \notin A \end{cases}$$

$$T_{\mathrm{e}}(h \,!?\, \mathbf{x}_{\mathrm{n}} := \mathbf{e}_{\mathrm{n}}, A) = \begin{cases} \delta & \text{if } \mathrm{ca}(h, *) \in A \\ h \,!?\, \mathbf{x}_{\mathrm{n}} := \mathbf{e}_{\mathrm{n}} & \text{if } \mathrm{ca}(h, *) \notin A \end{cases}$$

$$\begin{aligned} T_{\mathrm{e}}(u, A) &= u & T_{\mathrm{e}}([p_{\mathrm{act}}], A) &= [T_{\mathrm{e}}(p_{\mathrm{act}}, A)] \\ T_{\mathrm{e}}(p_{\mathrm{act}}; X, A) &= T_{\mathrm{e}}(p_{\mathrm{act}}, A); X & T_{\mathrm{e}}([p_{\mathrm{act}}]; X, A) &= [T_{\mathrm{e}}(p_{\mathrm{act}}, A)]; X \\ T_{\mathrm{e}}(b \to p_{\mathrm{ng}}, A) &= b \to T_{\mathrm{e}}(p_{\mathrm{ng}}, A) & T_{\mathrm{e}}(p_{\mathrm{n}} \,[\!]\, q_{\mathrm{n}}, A) &= T_{\mathrm{e}}(p_{\mathrm{n}}, A) \,[\!]\, T_{\mathrm{e}}(q_{\mathrm{n}}, A) \end{aligned}$$

Since p_s is a strict subterm of $\partial_A(p_s)$, well-definedness of $N(\partial_A(p_s))$ depends only on well-definedness of T_{e}. Well-definedness of T_{e} can be verified by observing that what occurs in recursions on the righthand side is a strict subterm of the lefthand side. Furthermore, $N(\partial_A(p_s)) \in \mathcal{P}_{\mathrm{n}}$ if $\{Y \mapsto T_{\mathrm{e}}(R_{\mathrm{n}}(Y), A) \mid Y \in \mathrm{dom}(R_{\mathrm{n}})\} \in \mathcal{R}_{\mathrm{n}}$, which follows from $T_{\mathrm{e}}(p_{\mathrm{n}}, A) \in \mathcal{P}_{\mathrm{n}}$. The latter is verified by induction on the structure of p_{n}. Bisimilarity of $\partial_A(p_s)$ and $N(\partial_A(p_s))$, rather surprisingly, turned out to be hard to prove using properties. However, with induction it suffices to give a bisimulation relation $\mathcal{D}$ witnessing $N(\partial_A(p_s)) \leftrightarrows \partial_A(N(p_s))$. As before, we use Theorem 4.1 and completeness of R_{n} (following from $p_s \in \mathcal{P}_s$) to construct and prove this (rather trivial) bisimulation relation.

$$\begin{aligned} \mathcal{D} = \{&(\llbracket_{\mathrm{R}} \{Y \mapsto T_{\mathrm{e}}(R_{\mathrm{n}}(Y), A) \mid Y \in \mathrm{dom}(R_{\mathrm{n}})\} :: X \,\rrbracket, \partial_A(\llbracket_{\mathrm{R}} R_{\mathrm{n}} :: X \,\rrbracket)) \\ &\mid R_{\mathrm{n}} \in \mathcal{R}_{\mathrm{n}}, X \in \mathrm{dom}(R_{\mathrm{n}})\} \\ \cup \{&(\llbracket_{\mathrm{R}} \{Y \mapsto T_{\mathrm{e}}(R_{\mathrm{n}}(Y), A) \mid Y \in \mathrm{dom}(R_{\mathrm{n}})\} :: R_{\mathrm{n}}(X) \,\rrbracket, \partial_A(\llbracket_{\mathrm{R}} R_{\mathrm{n}} :: R_{\mathrm{n}}(X) \,\rrbracket)) \\ &\mid R_{\mathrm{n}} \in \mathcal{R}_{\mathrm{n}}, X \in \mathrm{dom}(R_{\mathrm{n}})\} \end{aligned}$$

5 Bottle filling line example

Figure 1 shows a bottle filling line, based on [1], consisting of a storage tank that is continuously filled with a flow Q_{in}, a conveyor belt that supplies empty bottles, and a valve that is opened when an empty bottle is below the filling nozzle, and is closed when the bottle is full. When a bottle has been filled, the conveyor starts moving to put the next bottle under the filling nozzle, which takes one unit of time.

When the storage tank is not empty, the bottle filling flow Q equals Q_{set}. When the storage tank is empty, the bottle filling flow equals the flow Q_{in}. The system should operate in such a way that overflow of the tank does not occur. We assume $Q_{\text{in}} < Q_{\text{set}}$.

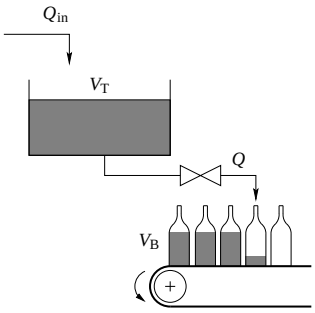

Fig. 1. Filling Line.

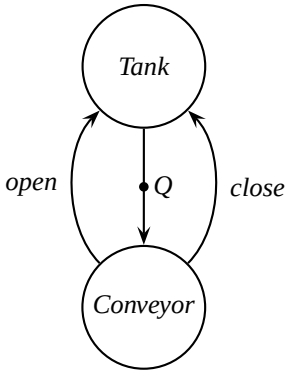

Fig. 2. Iconic model of the filling line.

Figure 2 shows an iconic representation of the model of the filling line. It consists of the processes *Tank* and *Conveyor* that interact by means of the channels *open* and *close*, and shared variable Q. The model is defined below. It has two parameters: the initial volume V_{T0} of the storage tank, and the value Q_{in} of the flow that is used to fill the storage tank. The constants Q_{set}, V_{Tmax}, and V_{Bmax} define the maximum value of the bottle filling flow Q, the maximum volume of the storage tank, and the filling volume of the bottles, respectively. The model *FillingLine* consists of the algebraic variable Q, the channels *open* and *close*, and the parallel composition of the process instantiations for the tank and the conveyor.

$$
\begin{aligned}
&\mathsf{const}\ Q_{\text{set}} : \mathsf{real} = 3\\
&\quad , V_{\text{Tmax}} : \mathsf{real} = 20\\
&\quad , V_{\text{Bmax}} : \mathsf{real} = 10\\
&\mathsf{model}\ \mathit{FillingLine}(\mathsf{val}\ V_{\text{T0}}, Q_{\text{in}} : \mathsf{real}) =\\
&[\![\ \mathsf{alg}\ Q : \mathsf{real}\\
&,\ \mathsf{chan}\ \mathit{open}, \mathit{close} : \mathsf{void}\\
&::\ \mathit{Tank}(Q, \mathit{open}, \mathit{close}, V_{\text{T0}}, Q_{\text{in}})
\end{aligned}
$$

$\|$ *Conveyor*$(Q, open, close)$
$]\!]$

The *Tank* process has a local continuous variable V_{T} that is initialized to V_{T0}. Its process body is a recursion scope consisting of three modes: closed, opened, and openedempty that correspond to the valve being closed, the valve being open, and the valve being open while the storage tank is empty. In the mode opened, the storage tank is usually not empty. When the storage tank is empty in mode opened, the delayable skip statement [skip] may be executed causing the next mode to be openedempty. Due to the consistent equation semantics, the skip statement can be executed only if the delay predicate in the next mode, i.e. openedempty, holds. This means, among others, that $V_{\mathrm{T}} = 0$ must hold. Therefore, the transition to mode openedempty can be taken only when the storage tank is empty. Note that the comma in delay predicates denotes conjunction. E.g. $\dot{V}_{\mathrm{T}} = Q_{\mathrm{in}},\ Q = 0$ means $\dot{V}_{\mathrm{T}} = Q_{\mathrm{in}} \wedge Q = 0$.

proc *Tank*(alg Q : real, chan *open*?, *close*? : void, val $V_{\mathrm{T0}}, Q_{\mathrm{in}}$: real) =
$[\![$ cont V_{T} : real $= V_{\mathrm{T0}}$
:: $[\![$ mode closed =
($\dot{V}_{\mathrm{T}} = Q_{\mathrm{in}},\ Q = 0,\ V_{\mathrm{T}} \leq V_{\mathrm{Tmax}}$ [] *open*?; opened)
, mode opened =
($\dot{V}_{\mathrm{T}} = Q_{\mathrm{in}} - Q,\ Q = Q_{\mathrm{set}},\ 0 \leq V_{\mathrm{T}} \leq V_{\mathrm{Tmax}}$
[] [skip]; openedempty
[] *close*?; closed
)
, mode openedempty =
($V_{\mathrm{T}} = 0, Q = Q_{\mathrm{in}}$ [] *close*?; closed)
:: closed
$]\!]$
$]\!]$

Process *Conveyor* supplies an empty bottle in 1 unit of time ($V_{\mathrm{B}}, t := 0, 1;\ \dot{t} = -1$ [] $t \leq 0 \rightarrow$ skip). Then it synchronizes with the storage tank process by means of the send statement *open* !!, and it proceeds in mode filling. When the bottle is filled in mode filling ($V_{\mathrm{B}} \geq V_{\mathrm{Bmax}}$), the process synchronizes with the storage tank to close the valve and returns to mode moving. The initial mode is moving.

proc *Conveyor*(alg Q : real, chan *open*!, *close*! : void) =
$[\![$ cont V_{B} : real $= 0, t$: real
:: $[\![$ mode moving= ($V_{\mathrm{B}}, t := 0, 1;\ \dot{t} = -1$ [] $t \leq 0 \rightarrow$ skip; *open* !!; filling)
, mode filling = ($V_{\mathrm{B}} \geq V_{\mathrm{Bmax}} \rightarrow$ *close* !!; moving)
:: moving
$]\!]$
$\|$ $\dot{V}_{\mathrm{B}} = Q$
$]\!]$

Figure 3 shows the results of the first 12 time units of a simulation run of the model

FillingLine$(5, 1.5)$, that is with model parameters $V_{\mathrm{T0}} = 5$ and $Q_{\mathrm{in}} = 1.5$. The graph shows that the first bottle is filled from time point 1 until time point $1 + 10/3 \approx 4.33$. Filling of the second bottle starts 1 time unit later, and somewhat after 7 time units, the storage tank becomes empty, so that filling continues at the reduced flow rate.

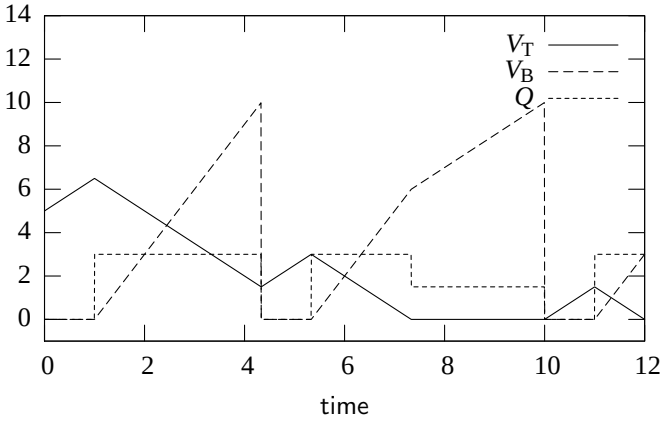

Fig. 3. Simulation results of model *FillingLine*.

5.1 Elimination of process instantiation

Elimination of the process instantiations for the *Tank* and *Conveyor* processes by replacing the process instantiations by their definitions, as defined in [5,20], leads to the following model:

model *FillingLine*(val $V_{\mathrm{T0}}, Q_{\mathrm{in}}$: real) =
‖[alg Q : real, chan *open*, *close* : void
:: ‖[var $V^{\mathrm{L}}_{\mathrm{T0}}$: real $= V_{\mathrm{T0}}$, $Q^{\mathrm{L}}_{\mathrm{in}}$: real $= Q_{\mathrm{in}}$
:: ‖[cont V_{T} : real $= V^{\mathrm{L}}_{\mathrm{T0}}$
:: ‖[mode closed =
($\dot{V}_{\mathrm{T}} = Q^{\mathrm{L}}_{\mathrm{in}}$, $Q = 0$, $V_{\mathrm{T}} \leq V_{\mathrm{Tmax}}$ [] *open*?; opened)
, mode opened =
($\dot{V}_{\mathrm{T}} = Q^{\mathrm{L}}_{\mathrm{in}} - Q$, $Q = Q_{\mathrm{set}}$, $0 \leq V_{\mathrm{T}} \leq V_{\mathrm{Tmax}}$
[] [skip]; openedempty
[] *close*?; closed
)
, mode openedempty =
($V_{\mathrm{T}} = 0, Q = Q^{\mathrm{L}}_{\mathrm{in}}$ [] *close*?; closed)
:: closed
]‖
]‖
]‖
‖ ‖[cont V_{B} : real $= 0, t$: real
:: ‖[mode moving= ($V_{\mathrm{B}}, t := 0, 1$; $\dot{t} = -1$ [] $t \leq 0 \rightarrow$ skip; *open*!!; filling)
, mode filling = ($V_{\mathrm{B}} \geq V_{\mathrm{Bmax}} \rightarrow$ *close*!!; moving)

$$
\begin{array}{l}
\quad :: \text{moving} \\
\quad]\!] \\
\quad \| \ \dot{V}_{\mathrm{B}} = Q \\
\quad]\!] \\
]\!]
\end{array}
$$

To avoid naming conflicts between the formal parameters V_{T0} and Q_{in} declared in the process definition for process *Tank*, and the actual arguments V_{T0} and Q_{in} in the process instantiation $\mathit{Tank}(Q, \mathit{open}, \mathit{close}, V_{\mathrm{T0}}, Q_{\mathrm{in}})$, the newly defined local discrete variables that are used to hold the values of the last two parameters of the process instantiation, are renamed to $V^{\mathrm{L}}_{\mathrm{T0}}$ and $Q^{\mathrm{L}}_{\mathrm{in}}$.

Without changing the semantics, two nested variables scope operators can be combined resulting in one variable scope operator iff both variable scope operators declare different local variables [7]. For example, $[\![\ \mathsf{var}\ x : \mathsf{real} = 0 :: [\![\ \mathsf{var}\ y : \mathsf{real} = 1 :: p \ ]\!] \ ]\!] \ \leftrightarrows [\![\ \mathsf{var}\ x : \mathsf{real} = 0, y : \mathsf{real} = 1 :: p \ ]\!]$.

Similarly, the parallel composition of two variable scope operators can be combined resulting in one variable scope operator applied on a process term that consists of the parallel composition of the original process terms of the variable scopes. For example, $[\![\ \mathsf{var}\ x : \mathsf{real} = 0 :: p \ ]\!] \| [\![\ \mathsf{var}\ y : \mathsf{real} = 1 :: q \ ]\!] \ \leftrightarrows [\![\ \mathsf{var}\ x : \mathsf{real} = 0, y : \mathsf{real} = 1 :: p \parallel q \ ]\!]$.

5.2 Elimination of parallel composition

The combination of the variable scopes, elimination of parallel composition and translation to the normal form as discussed in Section 3 leads to the model:

$$
\begin{array}{l}
\mathsf{model}\ \mathit{FillingLine}(\mathsf{val}\ V_{\mathrm{T0}}, Q_{\mathrm{in}} : \mathsf{real}) = \\
[\![\ \mathsf{alg}\ Q : \mathsf{real},\ \mathsf{chan}\ \mathit{open}, \mathit{close} : \mathsf{void} \\
:: [\![\ \mathsf{cont}\ V_{\mathrm{T}} : \mathsf{real} = V^{\mathrm{L}}_{\mathrm{T0}},\ V_{\mathrm{B}} : \mathsf{real} = 0 \\
\quad ,\ \mathsf{cont}\ t : \mathsf{real},\ \mathsf{var}\ V^{\mathrm{L}}_{\mathrm{T0}} : \mathsf{real} = V_{\mathrm{T0}},\ Q^{\mathrm{L}}_{\mathrm{in}} : \mathsf{real} = Q_{\mathrm{in}} \\
\quad :: [\![\ \mathrm{moving\_closed} = \\
\qquad\qquad (\ \dot{V}_{\mathrm{T}} = Q^{\mathrm{L}}_{\mathrm{in}},\ Q = 0,\ V_{\mathrm{T}} \le V_{\mathrm{Tmax}},\ \dot{V}_{\mathrm{B}} = Q \\
\qquad\qquad [\!] \ V_{\mathrm{B}}, t := 0, 1;\ \mathrm{moving}_0\mathrm{\_closed} \\
\qquad\qquad) \\
\quad ,\ \mathrm{moving}_0\mathrm{\_closed} = \\
\qquad\qquad (\ \dot{V}_{\mathrm{T}} = Q^{\mathrm{L}}_{\mathrm{in}},\ Q = 0,\ V_{\mathrm{T}} \le V_{\mathrm{Tmax}},\ \dot{V}_{\mathrm{B}} = Q,\ \dot{t} = -1 \\
\qquad\qquad [\!] \ t \le 0 \rightarrow \mathrm{skip};\ \mathrm{moving}_1\mathrm{\_closed} \\
\qquad\qquad) \\
\quad ,\ \mathrm{moving}_1\mathrm{\_closed} = \\
\qquad\qquad (\ \dot{V}_{\mathrm{T}} = Q^{\mathrm{L}}_{\mathrm{in}},\ Q = 0,\ V_{\mathrm{T}} \le V_{\mathrm{Tmax}},\ \dot{V}_{\mathrm{B}} = Q \\
\qquad\qquad [\!] \ \mathit{open}\,!?;\ \mathrm{filling\_opened} \\
\qquad\qquad) \\
\quad ,\ \mathrm{filling\_opened} = \\
\qquad\qquad (\ \dot{V}_{\mathrm{T}} = Q^{\mathrm{L}}_{\mathrm{in}} - Q,\ Q = Q_{\mathrm{set}},\ 0 \le V_{\mathrm{T}} \le V_{\mathrm{Tmax}},\ \dot{V}_{\mathrm{B}} = Q \\
\qquad\qquad [\!] \ [\mathrm{skip}];\ \mathrm{filling\_openedempty} \\
\qquad\qquad [\!] \ V_{\mathrm{B}} \ge V_{\mathrm{Bmax}} \rightarrow \mathit{close}\,!?;\ \mathrm{moving\_closed}
\end{array}
$$

[7] This may require renaming of local variables.

```
        )
    , filling_openedempty =
        ( V_T = 0, Q = Q_in^L, V̇_B = Q
        [] V_B ≥ V_Bmax → close!?; moving_closed
        )
    :: moving_closed
    ]|
  ]|
]|
```

5.3 *Substitution of constants and additional elimination*

The model below is the result of substitution of the globally defined constants by their values. Furthermore, the discrete variables Q_{in}^{L} and V_{T0}^{L}, that were introduced by elimination of the process instantiations, are eliminated. Also, the presence of the undelayable statements $V_{\text{B}}, t := 0, 1$ and *open*!? in modes moving_closed and moving$_1$_closed, respectively, allows elimination of the differential equations in these modes.

Most hybrid automaton based model checkers, such as PHAver [12] and HyTech [14], do not (yet) have urgent transitions that can be combined with guards. Therefore, the urgency in the guarded statements is removed by making the statements that are guarded delayable, and adding the closed negation of the guard as an additional delay predicate (invariant). E.g. $t \leq 0 \rightarrow$ skip is rewritten as $t \geq 0$ [] $t \leq 0 \rightarrow$ [skip].

```
model FillingLine(val V_T0, Q_in : real) =
|[ alg Q : real, chan open, close : void
:: |[ cont V_T : real = V_T0, V_B : real = 0, t : real
  :: |[ moving_closed =
          ( V_T ≤ 20, Q = 0
          [] V_B, t := 0, 1; moving_0_closed
          )
    , moving_0_closed =
          ( V̇_T = Q_in, Q = 0, V_T ≤ 20, V̇_B = 0, ṫ = −1, t ≥ 0
          [] t ≤ 0 → [skip]; moving_1_closed
          )
    , moving_1_closed =
          ( V_T ≤ 20, Q = 0
          [] open!?; filling_opened
          )
    , filling_opened =
          ( V̇_T = Q_in − 3, Q = 3, 0 ≤ V_T ≤ 20, V̇_B = 3, V_B ≤ 10
          [] [skip]; filling_openedempty
          [] V_B ≥ 10 → [close!?]; moving_closed
          )
    , filling_openedempty =
          ( V_T = 0, Q = Q_in, V̇_B = Q, V_B ≤ 10
```

```
            [] V_B ≥ 10 → [close !?]; moving_closed
            )
      :: moving_closed
      ]||
   ]||
]||
```

Figure 4 shows a graphical representation of the model. By means of straightforward mathematical analysis of the model, it can be shown that overflow never occurs if $Q_{\text{in}} \leq 30/13$.

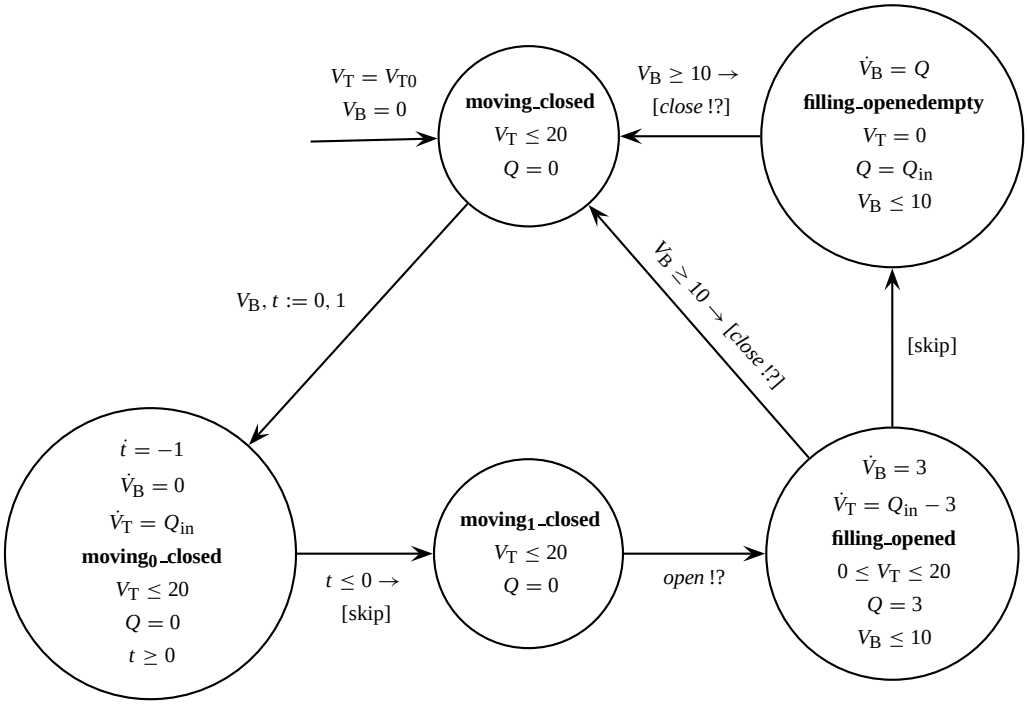

Fig. 4. Graphical representation of the linearized χ model.

5.4 Tool based verification

As a final step, for the purpose of tool-based verification, the model is translated to the input language of the hybrid IO automaton based tool PHAVer [12]. Since most hybrid automata, including PHAVer, do not know the concept of an algebraic variable, first the algebraic variables are eliminated from the χ model. Because of the consistent equation semantics of χ, each occurrence of an algebraic variable in the model can simply be replaced by the right hand side of its defining equation. The urgency due to unguarded undelayable statements is in principle translated by defining the corresponding flow clause as false. The resulting PHAVer model follows below. Note that an additional variable x is introduced and the derivatives of `Vb` and `Vt` need to be defined in all locations, because of the current inability of PHAVer to define false as flow clause.

```
automaton filling_line
```

```
  state_var: Vt,Vb,t,x;
  parameter: Vt0,Qin;
  synclabs : open,close,tau;
  loc moving_closed:
    while Vt <= 20 & x==0 wait {x'==1 & Vb'==0 & Vt'==0};
    when true sync tau do {Vt'==Vt & Vb'==0 & t'==1 & x'==0}
      goto moving0_closed;
  loc moving0_closed:
    while Vt <= 20 & t >= 0 wait {Vb'==0 & t'==-1 & Vt==30/13};
    when t <= 0 sync tau do {Vt'==Vt & Vb'==Vb & t'==t & x'==0}
      goto moving1_closed;
  loc moving1_closed:
    while Vt <= 20 & x==0 wait {x'==1 & Vb'==0 & Vt'==0};
    when true sync open do {Vt'==Vt & Vb'==Vb & t'==t}
      goto filling_opened;
  loc filling_opened:
    while Vt >= 0 & Vt <= 20 & Vb <= 10 wait {Vb'==3 & Vt'==30/13-3};
    when Vt==0 sync tau do {Vt'==Vt & Vb'==Vb & t'==t}
      goto filling_openedempty;
    when Vb >= 10 sync close do {Vt'==Vt & Vb'==Vb & t'==t & x'==0}
      goto moving_closed;
  loc filling_openedempty:
    while Vt == 0 & Vb <= 10 wait {Vb'==30/13};
    when Vb >= 10 sync close do {Vt'==Vt & Vb'==Vb & t'==t & x'==0}
      goto moving_closed;
  initially moving_closed & Vt == Vt0 & Vb==0 & x==0;
end
```

The following properties were derived: if $Q_{\text{in}} = 30/13$ and $0 \leq V_{\text{T0}} \leq V_{\text{Tmax}} - 30/13$, overflow does not occur, and the storage tank does not become empty when filling a bottle. The volume of the storage tank then remains in the region $V_{\text{T0}} \leq V_{\text{T}} \leq V_{\text{T0}} + 30/13$. If $Q_{\text{in}} > 30/13$, eventually overflow occurs. If $Q_{\text{in}} < 30/13$, eventually the container becomes empty every time a bottle is filled. In this small example, these properties can also be derived by means of straightforward mathematical analysis of the χ models of Section 5.2 or 5.3.

6 Conclusions and future work

The χ language is a hybrid process algebra, with a formal syntax and semantics, and is connected to a number of tools that can help in the design and analysis of hybrid systems. It has been applied in a number of case studies thereby showing its usefulness.

In this paper, we investigated equational reasoning in the χ language. A specification can be rewritten to a normal form, in a restricted syntax, so that translations to other languages and tools become easier. In particular, in this way the parallel composition operator can be eliminated, which is called linearization. We define the linearization of a reasonably expressive subset of χ. The correctness of this

linearization has been proved, and an implementation of the algorithm is described in [23].

In our linearization algorithm, the number of recursion variables is linear in the number of variables of a process term, except in the case of parallel composition. In that case, the number of recursion variables can grow exponentially. We expect that this situation can be improved, some progress is reported in [17].

For the future, we plan to work on extending the applicability of the algorithm by considering a larger subset of χ and by performing case studies. Furthermore, building tools for simulation, transition system generation, and translation to the input format of other powerful verification tools can be considered.

References

[1] Baeten, J. C. M. and C. A. Middelburg, "Process Algebra with Timing," EACTS Monographs in Theoretical Computer Science, Springer-Verlag, 2002.

[2] Beek, D. A. v., A. v. d. Ham and J. E. Rooda, *Modelling and control of process industry batch production systems*, in: E. F. Camacho, L. Basanez and J. A. de la Puente, editors, *15th Triennial World Congress of the International Federation of Automatic Control*, Barcelona, 2002, CD-ROM.

[3] Beek, D. A. v., K. L. Man, M. A. Reniers, J. E. Rooda and R. R. H. Schiffelers, *Deriving simulators for hybrid Chi models*, in: *IEEE International Symposium on Computer-Aided Control Systems Design* (2006), pp. 42–49.

[4] Beek, D. A. v., K. L. Man, M. A. Reniers, J. E. Rooda and R. R. H. Schiffelers, *Formal verification of hybrid Chi models using PHAVer*, in: *Proceedings of the conference on Mathematical Modelling 2006*, Vienna, Austria, 2006.

[5] Beek, D. A. v., K. L. Man, M. A. Reniers, J. E. Rooda and R. R. H. Schiffelers, *Syntax and consistent equation semantics of hybrid Chi*, Journal of Logic and Algebraic Programming **68** (2006), pp. 129–210.

[6] Beek, D. A. v., R. R. H. Schiffelers, M. Kvasnica and M. A. P. Remelhe, *Specification of the simulation interface*, Technical Report D 3.4.1, HYCON NoE (2005).

[7] Bortnik, E. M., N. Trčka, A. J. Wijs, B. Luttik, J. M. v. d. Mortel-Fronczak, J. C. M. Baeten, W. J. Fokkink and J. E. Rooda, *Analyzing a Chi model of a turntable system using Spin, CADP and Uppaal*, Journal of Logic and Algebraic Programming **65** (2005), pp. 51–104.

[8] Bos, V. and J. J. T. Kleijn, *Automatic verification of a manufacturing system*, Robotics and Computer Integrated Manufacturing **17** (2000), pp. 185–198.

[9] Bos, V. and J. J. T. Kleijn, "Formal Specification and Analysis of Industrial Systems," Ph.D. thesis, Eindhoven University of Technology (2002).

[10] Brand, P. v. d., , M. A. Reniers and P. J. L. Cuijpers, *Linearization of hybrid processes*, Journal of Logic and Algebraic Programming **68** (2006), pp. 54–104.

[11] Fernandez, J.-C., H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu and M. Sighireanu, *CADP: a protocol validation and verification toolbox*, in: Rajeev Alur and Thomas A. Henzinger, editors, *Eighth International Conference on Computer Aided Verification CAV*, Lecture Notes in Computer Science **1102** (1996), pp. 437–440.

[12] Frehse, G., *PHAVer: Algorithmic verification of hybrid systems past HyTech*, in: M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control, 8th International Workshop*, Lecture Notes in Computer Science **3414**, Springer-Verlag, 2005 pp. 258–273.

[13] Groote, J. F. and M. A. Reniers, *Algebraic process verification*, in: J. A. Bergstra, A. Ponse and S. A. Smolka, editors, *Handbook of Process Algebra*, Elsevier, 2001 pp. 1151–1208.

[14] Henzinger, T. A., P.-H. Ho and H. Wong-Toi, *A user guide to* HyTech, in: E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop*, Lecture Notes in Computer Science **1019** (1995), pp. 41–71.

[15] Hofkamp, A. T., "Reactive machine control, a simulation approach using χ," Ph.D. thesis, Eindhoven University of Technology (2001).

[16] Holzmann, G. J., "The SPIN Model Checker: Primer and Reference Manual," Addison Wesley Professional, Boston, 2003.

[17] Khadim, U., D. A. v. Beek and P. J. L. Cuijpers, *Linearization of hybrid chi using program counters*, Technical Report CS-Report 07-18, Eindhoven University of Technology, Department of Computer Science, The Netherlands (2007).

[18] Kleijn, J. J. T., M. A. Reniers and J. E. Rooda, *Analysis of an industrial system*, Formal Methods in System Design **22** (2003), pp. 249–282.

[19] Larsen, K. G., P. Pettersson and W. Yi, Uppaal *in a Nutshell*, International Journal on Software Tools for Technology Transfer **1** (1997), pp. 134–152.

[20] Man, K. L. and R. R. H. Schiffelers, "Formal Specification and Analysis of Hybrid Systems," Ph.D. thesis, Eindhoven University of Technology (2006).

[21] Nicollin, X. and J. Sifakis, *The algebra of timed processes, ATP: Theory and application*, Information and Computation **114** (1994), pp. 131–178.

[22] The MathWorks, Inc, "Writing S-functions, version 6," http://www.mathworks.com (2005).

[23] Theunissen, R. J. M., "Process algebraic linearization of hybrid Chi," Master's thesis, Eindhoven University of Technology (2006).

[24] Usenko, Y. S., "Linearization in μCRL," Ph.D. thesis, Eindhoven University of Technology (2002).