Linearization of hybrid Chi using program counters

Linearization of Hybrid Chi Using Program Counters

U.Khadim, D.A. van Beek, P.J.L. Cuijpers
Department of Mathematics and Computer Science
Department of Mechanical Engineering
Eindhoven University of Technology, P.O. Box 513
5600 MB Eindhoven, The Netherlands
{u.khadim, d.a.v.beek, p.j.l.cuijpers}@tue.nl

July 31, 2007

1 Introduction

The language χ was developed some years back as a modelling and simulation language for industrial systems [1, 2]. Originally, the language χ included features for modelling discrete event systems only. Later on it was extended with features to model dynamic behavior of a system as well [3, 4]. Hybrid χ was redesigned as a process algebra for hybrid systems and was given a formal semantics in [5]. Recently, its syntax has been improved further and some complexities from its semantics have been removed [6]. A number of tools for linearization [8], simulation and verification of hybrid χ models [7] are available. This report describes a method of linearizing hybrid χ process terms. By linearization, we mean the procedure of rewriting a process term into a linear form. A linear form consists of only basic operators of a process language. Linearization is synonymous to *elimination* found in many ACP style process algebras such as [9, 10, 11, 12, 13, 7]. In these process algebras we find elimination theorems that state that any process specification in a given process algebra can be rewritten into a simpler form, called a basic term. Each of these process algebras also contains a set of basic terms into which all closed terms of that process algebra can be rewritten. A basic term consists of only atomic actions, basic operators of the given process algebra (like choice and sequential composition) and guarded tail recursion. Elimination theorems are very useful in proving properties about closed terms of a process algebra as with these theorems proofs by structural induction become smaller. An important property of a basic term is that it does not contain parallelism. Hence, the term elimination is often used for elimination of a parallel operator from a process term. In this report the words *linear term* and *basic term*, *linearization* and *elimination* are used interchangeably.

Historically, μCRL [14], a process algebra with data, first used the term "linear process equation" (LPE) for its basic terms, and referred to the procedure of rewriting a process specification into a basic term as *linearization*. The terms *linear* and *linearization* refer to the fact that a linear process equation resembles a right linear data parameterized grammar [14]. A linear process equation or an LPE is a subclass of recursive process specifications that defines a complete system specification in a single recursive equation. A Linear Process Equation in μCRL has the following form:

$$\begin{aligned} X(d:D) = \quad & \textstyle\sum_{i\in I}\sum_{e\in E_i} a_i \cdot X(g_i(d,e)) \lhd c_i(d,e) \rhd \delta \\ & + \textstyle\sum_{j\in I}\sum_{e\in E_j} a_j \lhd c_j(d,e) \rhd \delta \end{aligned}$$

where I and J are disjoint finite sets of indexes and d denotes a state vector. Normally we are interested in a solution of the LPE in a particular initial state d_0. The equation is explained as follows. The process X being in a state d can, for any $e \in E_i$ that satisfy the condition $c_i(d,e)$, perform an action a_i, and then proceed to the state $g_i(d,e)$. Moreover, it can, for any $e \in E_j$ that satisfy the condition $c_j(d,e)$, perform an action a_j, and then terminate successfully. The actions in μCRL are parameterized. To simplify the definition of an LPE, we omit parameters from actions. For a comprehensive definition of an LPE, please refer to [14]. The format of an LPE is limited to basic operators of μCRL, a single recursion variable and guarded tail recursion. Depending upon the value of the parameter d and the guard conditions (i.e. $c_i(d,e)$ and $c_j(d,e)$) different options of an LPE are activated.

Advantages of rewriting a specification into a linear process equation are that many tools and techniques for analysis and verification of specifications operate only on linear terms [15]. Linearization / elimination in process algebra, can be compared to flattening in state charts [17] and in automata theory [16].

Following the work on μCRL, linearization algorithms and tools for hybrid process algebra [18] and hybrid χ [8] have also been developed. In [18], a similar approach to that of μCRL has been adopted and the final linear form is a linear process equation. For hybrid χ [8], the final linear form contains a set of linear recursive equations.

A concern in linearization is the size of the resulting linear term. When operators are removed from a specification its size may increase so much so that it becomes impossible to automatically linearize specification of a large system [14, 18]. Techniques like symbolic reasoning on data variables and variable abstraction [14, 18] have been developed to reduce the size of the resulting linear term in linearization. In the process of linearization, a parallel composition operator is eliminated from a specification. A parallel composition operator represents the result of simultaneous execution of two processes. Its semantics includes details such as synchronization, communication and interleaving of actions of the process terms executing in parallel. A linear form of the specification of a multi-component system with components running in parallel models the behaviour of the system using only basic operators. Hence the size of a linear form of such a system specification could be very large. In [14, 18] are used stack

like data structures to model interleaving in the linear form of a parallel composition. It has been pointed out in [14] that in cases where process variables are not parameterized by data, a counter with values in natural numbers can also serve the same purpose. In this report, we give an algorithm for linearization of hybrid χ specifications using such counters. We call these counters *program counters*.

In hybrid χ language, a program counter is a new discrete variable defined locally in the linear form of a process specification. Different values of a program counter activate different atomic constructs of the specification. Using program counters, a hybrid χ process term is linearized as follows:

Consider a parallel composition, $(a;\ b \parallel d;\ e;\ f)$, where a, b, c, d, e, f are non-communicating atomic actions. The symbol ; denotes sequential composition in hybrid χ. Eliminating the parallel operator from $(a;\ b \parallel d;\ e;\ f)$ results in the following linear form:

Let R denote the linear form of $(a;\ b \parallel d;\ e;\ f)$. Then,

$$\begin{aligned} R = \quad & a;\ (b;\ d;\ e;\ f \mathbin{[\!]} d;\ (b;\ e;\ f \mathbin{[\!]} e;\ (f;\ b \mathbin{[\!]} b;\ f))) \\ & \mathbin{[\!]} d;\ (a;\ (b;\ e;\ f \mathbin{[\!]} e;\ (f;\ b \mathbin{[\!]} b;\ f))) \end{aligned}$$

The symbol $[\!]$ denotes choice or alternative composition in hybrid χ.

Using program counters, the linear form $\widetilde{p}$ of $(a;\ b \parallel d;\ e;\ f)$ is modelled as follows:

$$\begin{aligned} \widetilde{p} = \quad & [\![_{\mathrm{V}}\ \{i_1 \mapsto \bot, i_2 \mapsto \bot\}, \emptyset, \emptyset \\ & :: [\![_{\mathrm{R}}\ \{X \ \mapsto \ (i_1 = 4) && \to \ a, i_1 := 1;\ X \\ & \qquad\qquad \mathbin{[\!]}\ (i_1 = 2) \wedge \neg \mathrm{Odd}(\{i_2\}) && \to \ b, i_1 := 2;\ X \\ & \qquad\qquad \mathbin{[\!]}\ (i_1 = 2) \wedge \mathrm{Odd}(\{i_2\} && \to \ b, i_1 := 0, i_2 := 0 \\ & \qquad\qquad \mathbin{[\!]}\ (i_2 = 6) && \to \ d, i_2 := 4;\ X \\ & \qquad\qquad \mathbin{[\!]}\ (i_2 = 4) && \to \ e, i_2 := 2;\ X \\ & \qquad\qquad \mathbin{[\!]}\ (i_2 = 2) \wedge \neg \mathrm{Odd}(\{i_1\}) && \to \ f, i_2 := 1;\ X \\ & \qquad\qquad \mathbin{[\!]}\ (i_2 = 2) \wedge \mathrm{Odd}(\{i_1\}) && \to \ f, i_2 := 0, i_1 := 0\} \\ & \quad :: (i_1 = 4 \wedge i_2 = 6) \curvearrowright X \\ & \quad]\!] \\ &]\!] \end{aligned}$$

where $[\![_{\mathrm{V}} \ldots]\!]$ represents a new variable scope in which two local discrete data variables i_1 and i_2 are declared. $[\![_{\mathrm{R}} \ldots]\!]$ represents a new recursion scope in which a new recursion variable X is declared. The process definition of X is a linear process term that defines the behaviour of $(a;\ b \parallel d;\ e;\ f)$. i_1 and i_2 are the program counters used in the linear process equation. The variables i_1, i_2 and X are not part of the original specification. To make them unobservable to an outside observer, they are declared as local variables in new variable and recursion scopes respectively.

An alternative of the process definition of X is explained as follows:

$$(i_1 = 2) \wedge \mathrm{Odd}(\{i_2\}) \to b, i_1 := 1;\ X$$

The predicate $i_1 = 2$ is the condition guarding the given alternative, a is an action and $i_1 := 1$ is an assignment updating the value of i_1. If $(i_1 = 2)$ evaluates to true, the action b is performed and the program counter i_1 is set to 1. After the action, the recursion variable X is called again.

The initialization operator $\curvearrowright$ sets the values of i_1 and i_2 to 4 and 6 respectively before the first call to variable X. In each recursive call, depending upon the action performed, the value of one of the program counters i_1 or i_2 is updated. A requirement of the linearization is that the resulting linear form must be bisimilar to the original process term. In the linearization of a process term with parallel composition, a different program counter for each component of parallel composition is used. The program counter of each component can be updated independently of the other program counters. For example, during the execution of X, the program counter i_1 can have value 4 and the program counter i_2 can have value 2, indicating that only actions d and e have been executed so far. By independently updating the two program counters, all possible interleaving of actions of parallel components are modelled and we do not need to explicitly include these interleavings in the linear form. In this way, the size of the linear form of a parallel composition is approximately of the order of the sum of the sizes of its components. An advantage of doing linearization this way is that parallel components can still be recognized in the linear form.

The structure of the report is as follows: In Section 2, we define the set of input process terms to our linearization algorithm. In Section 3, we define the syntax of the linear form. Section 4 informally gives a visualization of the linear form. The purpose of this visualization is to help in understanding the essentials of the linearization procedure. Section 5 inductively defines the linearization of a hybrid χ specification by giving a linearization algorithm for atomic constructs of hybrid χ and for all the operators allowed in an input process term. In Section 6, the Conclusion, we compare the new linearization algorithm with the previous one for hybrid χ [8]. We also discuss its position among other linearizations [14, 18].

2 Input to the algorithm

The set of hybrid χ process terms is defined in [6]. We impose some restrictions on the set of process terms $\mathcal{P}_\mathrm{s}$ which are allowed as input to the linearization algorithm. The BNF definition below defines the grammar for the process terms $p_\mathrm{s} \in \mathcal{P}_\mathrm{s}$.

$$
\begin{array}{rcll}
p_{\mathrm{s}} & ::= & p_{\mathrm{atom}} & \text{atomic actions} \\
& | & p_{\mathrm{u}} & \text{invariant and} \\
& & & \text{urgency conditions} \\
& | & u \curvearrowright p_{\mathrm{s}} & \text{initialization} \\
& | & p_{\mathrm{s}};\, p_{\mathrm{s}} & \text{sequential composition} \\
& | & p_{\mathrm{s}} \,[]\, p_{\mathrm{s}} & \text{alternative composition} \\
& | & p_{\mathrm{s}} \parallel p_s & \text{parallel composition} \\
& | & \partial_A(p_{\mathrm{s}}) & \text{Encapsulation} \\
& & \partial_H(p_{\mathrm{s}}) & \text{Send and receive} \\
& & & \text{action encapsulation} \\
& | & \upsilon_H(p_{\mathrm{s}}) & \text{urgent channel communication} \\
& | & [\![_{\mathrm{H}}\, H \;::\; p_{\mathrm{s}}\,]\!] & \text{channel scope } [\![_{\mathrm{H}}\,]\!] \\
& | & p_{\mathrm{R}} & \text{restricted use of recursion} \\
& | & [\![_{\mathrm{V}}\, \sigma_\perp, C, L :: p_{\mathrm{s}}\,]\!] & \text{variable scope } [\![_{\mathrm{V}}\,]\!]
\end{array}
$$

where u is a predicate on a set of model variables $\mathcal{V}$, $A \in A_{\mathrm{label}}$ is a set of labels of non-communicating actions and H is a set of channels. In $[\![_{\mathrm{V}}\, \sigma_\perp, C, L :: p_{\mathrm{s}}\,]\!]$, C is a set of local continuous variables, L is a set of local algebraic variables and $\sigma_\perp$ is a valuation of local variables.

The set $\mathcal{P}_{\mathrm{atom}}$ consists of atomic actions. An atomic action process term $p_{\mathrm{atom}} \in \mathcal{P}_{\mathrm{atom}}$ is defined below:

$$
\begin{array}{rcll}
p_{\mathrm{atom}} & ::= & l_{\mathrm{a}}, W : r & \text{action process term} \\
& | & h\,!\,\mathbf{e}_n, W : r & \text{send process term} \\
& | & h\,?\,\mathbf{x}_n, W : r & \text{receive process term} \\
& | & h\,!?\,\mathbf{x}_n := \mathbf{e}_n, W : r & \text{communication process term}
\end{array}
$$

An atomic action consists of an action label, a set of non-jumping model variables W and a predicate r. We restrict the syntax of predicates on model variables to the set $\mathcal{R}$, defined as follows:

Let $r \in \mathcal{R}$.

$$
\begin{array}{rcl}
r & ::= & \text{true} \\
& | & \text{false} \\
& | & x^+ \ \mathrm{op}_{\mathrm{r}}\ c \\
& | & x^- \ \mathrm{op}_{\mathrm{r}}\ c \\
& | & r \wedge r
\end{array}
$$

where x is a model variable, x^- denotes the values of variable x before an action, x^+ denotes the values of x after an action, c is any value in the set Λ and op_{r} is the set of relational operators, i.e.

$$\mathrm{op}_{\mathrm{r}} = \{<, >, =, \geq, \leq\}$$

The action labels can be labels of communication actions (as explained next), or labels of non-communicating actions such as a, b, c. By means of a communication, values are communicated from one process to the other, i.e. synchronization of send action $h\,!\,\mathbf{e}_n$ and receive action $h\,?\,\mathbf{x}_n$ yields communication

$h\,!?\,\mathbf{x}_n := \mathbf{e}_n$ by which data $\mathbf{e}_n$ is transferred from the sender to the receiver. The notation $\mathbf{e}_n$ denotes a vector of expressions whose values are sent and $\mathbf{x}_n$ is a vector of variables in which the received values are stored.

The set $\mathcal{P}_{\mathrm{u}}$ consists of invariants and urgency conditions.

$$\begin{array}{rcl} p_{\mathrm{u}} & ::= & \mathsf{inv}\ u \\ & | & \mathsf{urg}\ u \end{array}$$

A recursion scope operator (denoted by $[\![_{\mathrm{R}}\,]\!]$) is allowed only if no recursion definition of a recursion variable defined within the scope, refers to a recursion variable defined outside the scope. The syntax of the process terms defining a recursion variable is also restricted. Furthermore, only tail recursion is allowed and an occurrence of a recursion variable in a process definition must be guarded.

The restricted recursion scope operator process term is defined by:

$$\begin{array}{rcll} p_{\mathrm{R}} & ::= & [\![_{\mathrm{R}}\, R :: X_i\,]\!] & \mathrm{Complete}(R) \wedge X_i \in \mathrm{dom}\,(R) \\ & | & [\![_{\mathrm{R}}\, R :: p\,]\!] & \mathrm{Complete}(R) \wedge \mathrm{Recvars}(p) \in \mathrm{dom}\,(R) \end{array}$$

where,

1. $i \in \mathbb{N}^{>0}$;
2. $R \in \mathcal{R}$, where $\mathcal{R} : \mathcal{X} \mapsto \mathcal{P}$ is the set of all functions from recursion variables to process terms. R is known as a recursion definition. Syntactically, a recursion definition is denoted by a set of pairs $\{X_1 \mapsto p_1, \ldots, X_m \mapsto p_m\}$, where X_i denotes a recursion variable and p_i denotes a process term defining X_i;
3. The set $\mathcal{P}$ of process terms includes the following process terms:

$$\begin{array}{rcl} p & ::= & p_{\mathrm{s}} \\ & | & p_{\mathrm{s}}; X_i \\ & | & p_{\mathrm{s}}; p \\ & | & p \,[]\, p \end{array}$$

4. Another restriction on the set of possible process definitions of recursion variables is as follows:

   Recursion variables with process definitions that declare a variable scope operator followed by self recursion are not allowed in the input to the algorithm. For example the following recursion definition is not allowed:

   $$\{X_1 \mapsto [\![_{\mathrm{V}}\, \sigma_\perp, C, L :: p_{\mathrm{s}}\,]\!]\ ;\ X_1\}$$

   The reason for this restriction is explained in detail in sections 5.7 and 5.12.

5. The function Recvars : $\mathcal{P} \cup (\mathcal{X} \times \mathcal{R}) \to 2^{\mathcal{X}}$ takes a process term of the form p, or a recursion variable and a recursion definition. It returns the recursion variables present in the given process term or in the defining process term of the given recursion variable. We make sure that whenever the function Recvars is called with a recursion variable and a recursion definition, the recursion definition contains the definition of the given recursion variable.

$$\begin{array}{lcl}
\mathrm{Recvars}(p_\mathrm{s}) & = & \emptyset \\
\mathrm{Recvars}(p_\mathrm{s};\ X_i) & = & \{X_i\} \\
\mathrm{Recvars}(p_\mathrm{s};\ p) & = & \mathrm{Recvars}(p_\mathrm{s}) \cup \mathrm{Recvars}(p) \\
\mathrm{Recvars}(p \mathbin{[\!]} q) & = & \mathrm{Recvars}(p) \cup \mathrm{Recvars}(q) \\
\\
\mathrm{Recvars}(X_i, \{X_i \mapsto p\}) & = & \mathrm{Recvars}(p) \\
\mathrm{Recvars}(X_i, \{X_j \mapsto p\})_{j \neq i} & = & \emptyset \\
\mathrm{Recvars}(X_i, R \cup R') & = & \mathrm{Recvars}(X_i, R) \cup \mathrm{Recvars}(X_i, R')
\end{array}$$

6. The function Complete : $\mathcal{R} \to \mathcal{B}\mathrm{ool}$ takes a recursion definition R. It collects the recursion variables that are mentioned in the defining process terms of all recursion variables in the domain of R. If the set thus obtained is a subset of the domain of R, then Complete returns true else it returns false.

$$\begin{array}{lll}
\mathrm{Complete}(R) = & \mathrm{true} & \text{if } \bigcup_{X_i \in \mathrm{dom}(R)} \mathrm{Recvars}(X_i, R) \subseteq \mathrm{dom}(R) \\
& \mathrm{false} & \text{otherwise}
\end{array}$$

3 Output Form of the algorithm

The set of linearized process terms $\widetilde{\mathcal{P}}$, with $\widetilde{p} \in \widetilde{\mathcal{P}}$, is defined as:

$$\begin{array}{rcl}
\widetilde{p} & ::= & [\![_\mathrm{V}\, \sigma_\mathrm{pc} \cup \sigma, C, L :: \\
& & \qquad [\![_\mathrm{R}\, \{X \mapsto \overline{p}\} :: u \wedge u_\mathrm{pc} \curvearrowright X\ ]\!] \\
& &]\!]
\end{array}$$

We discuss the structure of the linear process term $\widetilde{p}$ in the following sections:

3.1 Variable scope operator and program counters

1. The set of program counters is defined as follows:

$$\mathcal{I} = \{i_k \mid k \in \mathbb{N}^{>0}\},\ \text{such that } \mathcal{I} \cap \mathcal{V} = \emptyset$$

2. The valuation $\sigma_\mathrm{pc} : \mathcal{I} \mapsto \{\bot\}$ is a partial function which is syntactically denoted as $\{i_1 \mapsto \bot, \ldots i_k \mapsto \bot\}$, k being the number of program counters used in the definition of a linear form. The valuation σ_pc declares the program counters used to describe a linear form as local discrete variables. These are distinct from all other local discrete, algebraic or continuous model variables.

3. The valuation $\sigma : \mathcal{V} \mapsto \{\bot\}$ is syntactically denoted as $\{x \mapsto \bot, y \mapsto \bot, \ldots, z \mapsto \bot\}$, where x, y, z are local discrete or continuous model variables other than program counters.

4. C is the set of local continuous variables and L is the set of local algebraic variables.

3.2 Recursion scope operator

The recursion scope operator $[\![_{\mathrm{R}}\, \{X \mapsto \overline{p}\} :: u \wedge u_{\mathrm{pc}} \curvearrowright X\,]\!]$, where u_{pc} is a predicate over program counters and u is a predicate over model variables, defines a single recursion definition. The right hand side of the recursive definition is a linear process term.

The BNF definition of the set of process terms $\overline{\mathcal{P}}$, with $\overline{p} \in \overline{\mathcal{P}}$ is as follows:

$$\begin{array}{rcl} \overline{p} & ::= & b_{\mathrm{pc}} \rightarrow p_{\mathrm{u}} \\ & | & b_{\mathrm{pc}} \rightarrow p_{\mathrm{act}}, \mathrm{update}(X_i);\ X \\ & | & b_{\mathrm{pc}} \rightarrow p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}};\ X \\ & | & b_{\mathrm{pc}} \rightarrow p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}} \\ & | & \overline{p} \,[\!]\, \overline{p} \end{array}$$

where X_i for any natural number i denotes a recursion variable. The notation $\mathrm{update}(X_i)$ occurs only in intermediate output forms as an intermediate result of linearizing a recursion scope operator.

The set of action process terms $\mathcal{P}_{\mathrm{act}}$, with $p_{\mathrm{act}} \in \mathcal{P}_{\mathrm{act}}$, and the set of action predicates $\mathcal{AP}$, with $\mathrm{ap} \in \mathcal{AP}$, are defined as follows:

$$\begin{array}{rcl} p_{\mathrm{act}} & ::= & p_{\mathrm{atom}} \\ & | & p_{\mathrm{atom}}, \mathrm{ap} \\ \mathrm{ap} & ::= & W : r \\ & | & \mathrm{ap}, \mathrm{ap} \end{array}$$

where p_{atom} is an atomic action that has been defined in Section 2, W is a subset of model variables and $r \in \mathcal{R}$ is a jump predicate containing model variables.

In the BNF definition of a process term $\overline{p}$, the alternatives consisting of an urgency condition or an invariant are never followed by the recursion variable X, because an urgency condition and an invariant do not terminate. Some alternatives with action process terms are also not followed by a recursive call to X. These are the terminating actions. The process term $\overline{p}$ can terminate by executing one of the terminating actions. When $\overline{p}$ terminates, all program counters are set to zero.

The set of action predicates $\mathcal{AP}_{\mathrm{pc}}$ that update program counters, with $\mathrm{ap}_{\mathrm{pc}} \in \mathcal{AP}_{\mathrm{pc}}$ and the set of predicates on program counters $\mathcal{R}_{\mathrm{pc}}$, with $r_{\mathrm{pc}} \in \mathcal{R}_{pc}$, are defined as follows:

$$\begin{array}{rcl} \mathrm{ap}_{\mathrm{pc}} & ::= & W_{\mathrm{pc}} : r_{\mathrm{pc}} \\ & | & \mathrm{ap}_{\mathrm{pc}}, \mathrm{ap}_{\mathrm{pc}} \\ W_{\mathrm{pc}} & \subseteq & \mathcal{I} \\ r_{\mathrm{pc}} & ::= & \bigwedge_{i_k \in W_{\mathrm{pc}}} i_k = c_k \end{array}$$

where $c_k \in \mathbb{N}$. We use the convention that $\bigwedge_{i_k \in \emptyset} i_k = c_k$ evaluates to true.

3.3 Even and Odd values of program counters

Program counters in a linear form either have an odd or an even value. Even values are reserved for the so called "active" program counters. Odd values are reserved for the so called inactive program counters. This distinction between active and inactive program counters is needed to be able to properly deal with (partial) termination in parallel composition. In parallel composition, we need two concepts of termination: local termination of a component of parallel composition, and global termination. Local termination refers to termination of a component of a parallel composition, when the other components of the composition have not yet terminated. On local termination, the program counters of the terminating component are set to an odd value. Global termination refers to the final termination of the parallel composition, that takes place when the last component of the parallel composition terminates. When performing a terminating action of a component of a parallel composition, in order to determine whether local or global termination should follow, we check the parity of program counters of the other components. We do this by checking the parity of the product of all program counters of other components.

The set of guards, $\mathcal{B}_{\mathrm{pc}}$, where $b_{\mathrm{pc}} \in \mathcal{B}_{\mathrm{pc}}$, is defined as follows:

$$\begin{array}{rcl} b_{\mathrm{pc}} & ::= & b_{\mathrm{even}} \\ & | & b_{\mathrm{even}} \wedge \mathrm{Odd}(I) \\ & | & b_{\mathrm{even}} \wedge \neg \mathrm{Odd}(I) \end{array}$$

where,

$$\begin{array}{rcl} b_{\mathrm{even}} & ::= & i_k = e \\ & | & b_{\mathrm{even}} \wedge b_{\mathrm{even}} \end{array}$$

where e is an even number, $i_k \in \mathcal{I}$, $I \subseteq \mathcal{I}$ and $\mathrm{Odd}(I)$ denotes $\Pi_I \mod 2 = 0$ where Π_I denotes the product of the elements from set I.

The initialization predicate u is used to initialize model variables other than program counters; u is any predicate including true.

The initialization predicate u_{pc} initializes the local program counters. The initial value of a program counter is a natural number. An even value indicates a program counter is active and an odd value indicates a program counter that is inactive.

The set of initialization predicates $\mathcal{U}_{pc}$, with $u_{\mathrm{pc}} \in \mathcal{U}_{\mathrm{pc}}$, is defined as follows:

$$\begin{array}{rcl} u_{\mathrm{pc}} & ::= & i_k = n \\ & | & u_{\mathrm{pc}} \wedge u_{\mathrm{pc}} \end{array}$$

$i_k \in \mathcal{I}$ and $n \in \mathbb{N}$.

Initially inactive program counters in a process term indicate that the process term consists of a sequential composition such that the number of program counters of the second sequent is greater than the number of program counters

in the first sequent. The program counters that are only needed in the second sequent are declared at the start, but they remain inactive (evaluate to an odd value) while the process is in the first sequent.

Initial options or initial alternatives in a linear process equation of a given process term are the alternatives of the LPE consisting of the first possible actions or first possible urgency or invariant conditions of the given process term. By convention, the highest values of program counters guard the initial options of an LPE. In a linear process term $\widetilde{p}$, with an initialization predicate u_{pc}, an alternative guarded by a guard b_{pc} is an initial option of $\widetilde{p}$ only if $u_{\mathrm{pc}} \implies b_{\mathrm{pc}}$.

In the definition of a linear form, we observe the following:

1. An option of the LPE is never activated by odd values of program counters alone. The guard predicate b_{pc} always contains an atom that compares the value of at least one program counter against an even value.

2. An odd value of a program counter i_k ($k \in \mathbb{N}$) in a linear form represents one of the two cases:

   (a) The linear form under consideration originates from a parallel composition and the parallel component that uses program counter i_k in the definition of its linear form has terminated;

   (b) Or, the given linear form originates from a sequential composition. The first sequent has less number of parallel components than k whereas the second sequent has at least k parallel components. The odd value of program counter i_k indicates that the first sequent has not terminated yet and therefore i_k is inactive.

   Instead of odd values, we could also have used a specific value (for example $\bot$) to fulfill the purpose mentioned above. A program counter with that value would indicate both local termination of a parallel component and inactive program counter in a sequential composition. However, odd numbers possess some desirable properties. We are able to use these properties to our advantage at several places in the linearization algorithm such as:

   (a) We reuse program counters in the linearization of sequential, alternative and recursion scope operators. When reusing, we increment all the used values of a given program counter by an even number to allow for more values for representing all required guard conditions. Incrementing an odd value by an even number gives an odd number. Therefore an inactive program counter remains inactive when we reuse program counters. Hence we do not have to do any book keeping for inactive program counters.

   (b) For detecting termination of a component in a parallel composition, we check the parity of all program counters used in the linear form of that component. This is easily done by checking the parity of the product of its program counters as the product of odd values is an odd value.

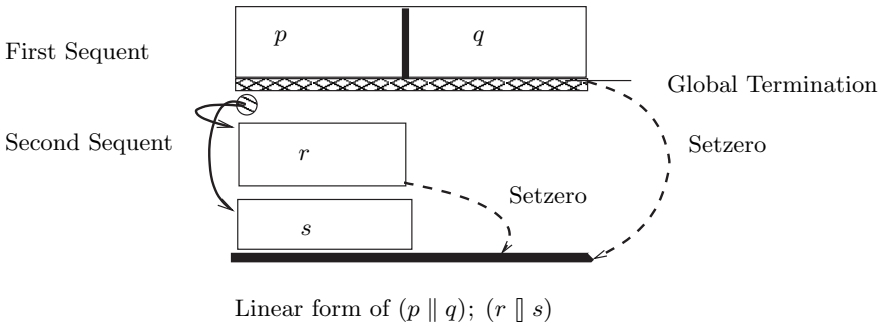


Figure 1: A graphical representation of a linear form

If a specific value is decided to be used instead of odd values, then the properties of that specific value should be exploited in the linearization algorithm.

4 Visualization of a linear form

The linearization algorithm turns out to be complicated and involves many steps for linearization of each operator. To help understanding the basic steps of the algorithm, we devise a visualization of the process term obtained after linearization. As can be seen in the previous section, the linear form contains many features. It is only possible to illustrate a subset of these features in a two dimensional diagram. We focus on the number of program counters used in a linear form, the program counters that are active in a particular segment of a linear form, and the changes in the values of program counters as different actions of a process term are executed.

Figure 1 shows a graphical representation of a linear process term. Features such as lines and arrows are incorporated in the graphical representation diagram to indicate jumps in the values of program counters or to indicate the end of a parallel composition.

We discuss them below:

1. The width of the block (length along horizontal axis) indicates the total number of program counters in a linear process term. The indices of program counters increase as we move from left to right in a block.

2. The height of a block represents the highest of the maximum values of all program counters in a process term. In our algorithm, we keep the convention that program counter i_1 has the highest maximum value. Therefore, in a parallel composition (see Section 5.5) we shift the program counters of the process term that has the lower value for program counter i_1. The values of program counters decrease as we move down in a block.

3. A black line at the bottom of a block stretching the entire length of a block indicates that all program counters have been set to zero.

4. A patterned horizontal line indicates the termination of a parallel composition. The value of a program counter can go below a patterned line only if all the program counters have reached it.

5. A dot represents the root of an alternative composition. The number of arrows originating from the dot indicates the total number of initially available options in the operands of the alternative composition. The arrows should end at appropriate places in alternatives. The exact place where an arrow ends in a block is kept abstract in our diagrams to avoid cluttering. In finding the linear form of an alternative composition, the values of program counters that are common in the operands of the alternative composition should be made distinct. To do this, we increment the values of program counters in one of the process terms. The zero values of the program counters are not incremented since all program counters must be set to zero at termination. In terms of our visualization, the block of one of the operands of alternative composition is placed on top of the other. A dashed arrow originates from the end of the block placed on the top of the other and ends at the bottom of the alternative composition.

6. A vertical black line in a block divides two operands of a parallel composition.

7. In a recursion scope operator, the blocks representing the linear forms of the process definitions of the recursion variables are placed on top of one another. Each block contains the name of the recursion variable it represents. The blocks may have arrows (mimicking a call to a recursion variable) originating from their bottom surfaces and ending in the top of other blocks. The pointer update(X_i) can be viewed as ports at the bottom of a block from which arrows originate.

We do not give a visualization for the urgent action operator, the variable scope operator, the encapsulation and the channel scope operator.

5 Linearization Algorithm

5.1 Notations

The following functions and notations are used in the linearization algorithm:

1. The notation $x[a'_k/a_k]_{k\in S,P(a_k)}$ represents x with every occurrence of a_k that satisfies a certain property P replaced by a'_k, for all k in a set S.

2. The notation $x[a'/a]_{a\in S}$ denotes x with every occurrence of a in x replaced by a', for all a in a set S.

3. The function Normalize : $\mathcal{P} \to \widetilde{\mathcal{P}}$ returns the linear form of a given process term. In case a process term of the form $p_{\rm s}$ is input to the algorithm, then the linear process term returned does not contain pointers of the form update(X_i).

4. The function rhs : $\widetilde{\mathcal{P}} \to \overline{\mathcal{P}}$ takes a linear process term and returns the righthand side of its single recursion definition.

$$\text{rhs}(\|[_{\rm V}\, \sigma_{\rm pc} \cup \sigma, C, L :: \|[_{\rm R}\, \{X \mapsto \overline{p}\} :: u \wedge u_{\rm pc} \curvearrowright X\,]]\|) \quad = \quad \overline{p}$$

5. The function $U : \widetilde{\mathcal{P}} \to$ Predicate returns the predicate initializing the model variables of the given linear form.

$$U(\|[_{\rm V}\, \sigma_{\rm pc} \cup \sigma, C, L :: \|[_{\rm R}\, \{X \mapsto \overline{p}\} :: u \wedge u_{\rm pc} \curvearrowright X\,]]\|) \quad = \quad u$$

6. The function $U_{\rm pc} : \widetilde{\mathcal{P}} \to$ Predicate returns the predicate initializing the program counters of the given linear form.

$$U_{\rm pc}(\|[_{\rm V}\, \sigma_{\rm pc} \cup \sigma, C, L :: \|[_{\rm R}\, \{X \mapsto \overline{p}\} :: u \wedge u_{\rm pc} \curvearrowright X\,]]\|) \quad = \quad u_{\rm pc}$$

7. The function Sigma : $\widetilde{\mathcal{P}} \to \mathcal{V} \mapsto \Lambda$ returns the valuation of local model variables in a linear process term. The symbol Λ, denotes the set of all possible values for model variables other than program counters.

$$\text{Sigma}(\|[_{\rm V}\, \sigma_{\rm pc} \cup \sigma, C, L :: \|[_{\rm R}\, \{X \mapsto \overline{p}\} :: u \wedge u_{\rm pc} \curvearrowright X\,]]\|) \quad = \quad \sigma$$

8. The function pcs : $(\overline{\mathcal{P}} \cup \mathcal{B}_{\rm pc} \cup \mathcal{R}_{\rm pc} \cup \mathcal{AP}_{\rm pc}) \to \mathcal{I}$ returns the set of program counters used in its argument. The set of program counters in a linear process term is the same as the set of program counters in the right hand side defining its recursion variable.

$$\begin{array}{lcl}
\text{pcs}(\overline{p} \,[\!]\, \overline{q}) & = & \text{pcs}(\overline{p}) \cup \text{pcs}(\overline{q}), \\
\text{pcs}(b_{\rm pc} \to p_{\rm u}) & = & \text{pcs}(b_{\rm pc}) \\
\text{pcs}(b_{\rm pc} \to p_{\rm act}, \text{update}(X_i);\ X) & = & \text{pcs}(b_{\rm pc}) \\
\text{pcs}(b_{\rm pc} \to p_{\rm act}, \text{ap}_{\rm pc}) & = & \text{pcs}(b_{\rm pc}) \cup \text{pcs}(\text{ap}_{\rm pc}) \\
\text{pcs}(b_{\rm pc} \to p_{\rm act}, \text{ap}_{\rm pc};\ X) & = & \text{pcs}(b_{\rm pc}) \cup \text{pcs}(\text{ap}_{\rm pc}) \\
\text{pcs}(i_k = n) & = & \{i_k\} \\
\text{pcs}(\text{Odd}(S)) & = & S \\
\text{pcs}(\neg\text{Odd}(S)) & = & S \\
\text{pcs}(b_{\rm pc} \wedge b'_{pc}) & = & \text{pcs}(b_{\rm pc}) \cup \text{pcs}(b'_{\rm pc}) \\
\text{pcs}(W_{\rm pc}) & = & W_{\rm pc} \\
\text{pcs}(r_{\rm pc} \wedge r'_{\rm pc}) & = & \text{pcs}(r_{\rm pc}) \cup \text{pcs}(r'_{\rm pcs}) \\
\text{pcs}(W_{\rm pc} : r_{\rm pc}) & = & \text{pcs}(W_{\rm pc}) \cup \text{pcs}(r_{\rm pcs}) \\
\text{pcs}(W_{\rm pc} : r_{\rm pc}, \text{ap}_{\rm pc}) & = & \text{pcs}(W_{\rm pc} : r_{\rm pc}) \cup \text{pcs}(\text{ap}_{\rm pc})
\end{array}$$

where $b_{\rm pc}, b'_{\rm pc}$ are guard predicates in set $\mathcal{B}_{\rm pc}$, $r_{\rm pc}, r'_{\rm pc}$ are action predicates on program counters in set $\mathcal{R}_{\rm pc}$, S and $W_{\rm pc}$ are subsets of program counters and $k \in \mathbb{N}^{>0}, n \in \mathbb{N}$,

9. The function $\mathrm{Count} : \overline{P} \to \mathbb{N}$ takes the linear process equation of a linear form and returns number of program counters used in it.

$$\mathrm{Count}(\overline{p}) = |\mathrm{pcs}(\overline{p})|$$

10. The function $\mathrm{value} : \mathcal{U}_{\mathrm{pc}} \times \mathbb{N} \to \mathbb{N}$ takes a predicate of the form $\mathcal{U}_{\mathrm{pc}}$ and the index of a program counter. It returns the value assigned to the program counter with the given index in the given predicate. In our algorithm we ensure that $\mathrm{value}(u_{\mathrm{pc}}, k)$ is only called when the program counter with index k is present in the predicate u_{pc}.

    The predicate u_{pc} is a conjunction. The conjunctions are such that a program counter with a particular index is used only in one atom of the conjunction. The function value is a recursive function. It checks the atoms of u_{pc} looking for the required program counter. When the program counter with the given index is not in the current atom, it returns 0. The function value applied to a conjunction of predicates is the maximum of the values returned when applied to each predicate individually.

$$\begin{array}{lcl} \mathrm{value}(i_k = n, k) & = & n \\ \mathrm{value}(i_k = n, l)_{l \neq k} & = & 0 \\ \mathrm{value}(u_{\mathrm{pc}} \wedge u'_{\mathrm{pc}}, k) & = & \max(\mathrm{value}(u_{\mathrm{pc}}, k), \mathrm{value}(u'_{\mathrm{pc}}, k)) \end{array}$$

    where, $k, l \in \mathbb{N}^{>0}, n \in \mathbb{N}$.

11. The function $\mathrm{Alt} : \overline{\mathcal{P}} \to 2^{\overline{\mathcal{P}}}$ returns the set of alternatives of a process term of the form $\overline{\mathcal{P}}$.

$$\begin{array}{lcl} \mathrm{Alt}(b_{\mathrm{pc}} \to p_{\mathrm{u}}) & = & \{b_{\mathrm{pc}} \to p_{\mathrm{u}}\} \\ \mathrm{Alt}(b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{update}(X_i);\ X) & = & \{b_{\mathrm{pc}} \to p_{\mathrm{act}} \mathrm{update}(X_i);\ X\} \\ \mathrm{Alt}(b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}}) & = & \{b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}}\} \\ \mathrm{Alt}(b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}};\ X) & = & \{b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}};\ X\} \\ \mathrm{Alt}(\overline{p} \mathbin{[\!]} \overline{q}) & = & \mathrm{Alt}(\overline{p}) \cup \mathrm{Alt}(\overline{q}) \end{array}$$

12. The function $\Big[\!\Big] : 2^{\overline{\mathcal{P}}} \to \overline{\mathcal{P}}$ takes a set of process terms of the form $\overline{\mathcal{P}}$ as its argument and returns the term obtained after alternatively composing all the elements of the set. A special case is the empty set which returns inv true, i.e. $\Big[\!\Big] \emptyset = \mathsf{inv}$ true.

13. The function $\mathrm{Nonterm} : \overline{\mathcal{P}} \to \overline{\mathcal{P}}$: takes a process term $\overline{p}$ and returns a process term consisting of only the non-terminating alternatives of $\overline{p}$. A non-terminating alternative consists of either an action followed by a re-

cursive call to X, or an invariant or an urgency condition.

$$\begin{aligned}\mathrm{Nonterm}(\overline{p}) \;=\; & \Big[\!\Big] \{b_{pc} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}};\ X \mid b_{pc} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}};\ X \in \mathrm{Alt}(\overline{p})\} \\ & \Big[\!\Big] \{b_{pc} \to p_{\mathrm{u}} \mid b_{pc} \to p_{\mathrm{u}} \in \mathrm{Alt}(\overline{p})\} \\ & \Big[\!\Big] \{b_{pc} \to p_{\mathrm{act}}, \mathrm{update}(X_i);\ X \\ & \quad \mid b_{pc} \to p_{\mathrm{act}}, \mathrm{update}(X_i);\ X \in \mathrm{Alt}(\overline{p}) \\ & \quad \}\end{aligned}$$

14. The function $\mathrm{Term} : \overline{\mathcal{P}} \to \overline{\mathcal{P}}$: takes a process term $\overline{p}$ and returns a process term consisting of only the terminating alternatives of $\overline{p}$. A terminating alternative consists of an action without a trailing call to the recursion variable X.

$$\mathrm{Term}(\overline{p}) \;=\; \Big[\!\Big] \{b_{pc} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}} \mid b_{pc} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}} \in \mathrm{Alt}(\overline{p})\}$$

In the following sections, we linearize different forms of the input process term p_{s} one by one.

5.2 Atomic actions

The linear form of an atomic action is defined by the Normalize function as follows:

$$\begin{aligned}\mathrm{Normalize}(p_{\mathrm{atom}}) = \; & [\![_{\mathrm{V}}\, \{i_1 \mapsto \bot\}, \emptyset, \emptyset \\ & ::\ [\![_{\mathrm{R}}\, \{X \mapsto (i_1 = 2) \to p_{\mathrm{atom}}, \{i_1\} : i_1 = 0\} \\ & \quad :: \mathrm{true} \wedge (i_1 = 2) \curvearrowright X \\ & \quad]\!] \\ &]\!]\end{aligned}$$

It is represented by a square with a black bottom in Figure 2.

Figure 2: An atomic action

5.3 Invariants or Urgency Conditions

An invariant or urgency condition, denoted by p_{u}, is defined by the Normalize function as follows:

$$\begin{aligned}\mathrm{Normalize}(p_{\mathrm{u}}) = \; & [\![_{\mathrm{V}}\, \{i_1 \mapsto \bot\}, \emptyset, \emptyset \\ & ::\ [\![_{\mathrm{R}}\, \{X \mapsto (i_1 = 2) \to p_{\mathrm{u}}\} \\ & \quad :: \mathrm{true} \wedge (i_1 = 2) \curvearrowright X \\ & \quad]\!] \\ &]\!]\end{aligned}$$

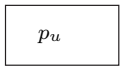


Figure 3: An invariant or urgency condition

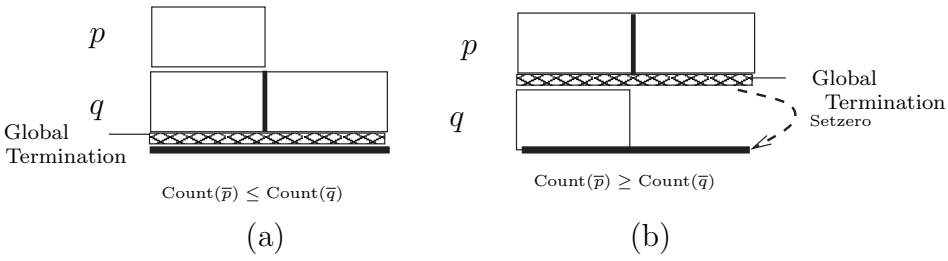


Figure 4: Sequential Composition

It is represented by a square without a black bottom in Figure 3.

5.4 Sequential Composition

A process p; q first behaves as p. After p has terminated, p; q continues behaving as q.

Assume

$$\begin{aligned}\text{Normalize}(p) = \widetilde{p} = \; & [\![_{\text{V}}\, \sigma^p_{\text{pc}} \cup \sigma_p, C_p, L_p \\ & :: \; [\![_{\text{R}}\, \{X \mapsto \overline{p}\} \; :: u_p \wedge u^p_{\text{pc}} \curvearrowright X \,]\!] \\ &]\!]\end{aligned}$$

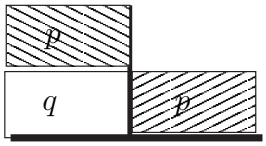
and

$$\begin{aligned}\text{Normalize}(q) = \widetilde{q} = \; & [\![_{\text{V}}\, \sigma^q_{\text{pc}} \cup \sigma_q, C_q, L_q \\ & :: \; [\![_{\text{R}}\, \{X \mapsto \overline{q}\} \; :: u_q \wedge u^q_{\text{pc}} \curvearrowright X \,]\!] \\ &]\!]\end{aligned}$$

The set of input process terms to the Normalize function is $\mathcal{P}$. The case where the second sequent in a given sequential composition is a recursion variable (i.e. a process term p_s; X_i, is given as input to the algorithm) is dealt in the linearization of the recursion scope operator (See Section 5.7).

In the linear form of a sequential composition, the values of program counters that are common in the linear forms of the two sequents are made distinct by incrementing the values of all program counters in the first sequent by the maximum value of the program counter i_1 in the second. Recall that in a linear process term, i_1 always has the greatest or one of the greatest values among all program counters. In this way, no two guards where one is in the first sequent and the other is in the second sequent can get activated by the same values of program counters. Let p be the first sequent and q be the second sequent.

Instead of incrementing all program counters' values in p by the maximum value of i_1 in q, we could also increment a program counter i_k in p, by the maximum value of i_k in q or zero incase the total number of program counters in q is less than k. This approach will also make the value of i_k distinct in the two operands of sequential composition. Adopting this approach would result in a linear form best explained by modifying figure 4(b) as follows:



$\text{Count}(\overline{p}) \geq \text{Count}(\overline{q})$

Figure 5: A different way of incrementing program counter values

The shaded area in the figure 5 represents the linear form of the first sequent p. We adopt the first approach of incrementing all program counter values in the first sequent by the maximum value of i_1 in the second sequent as it is simpler.

The total number of program counters used in the linear form is the maximum of the numbers of program counters in the two sequents. Initially only the initial options of the first sequent are enabled. Only when the first sequent terminates, the options of the second sequent are activated.

If $\text{Term}(\overline{p}) = \mathsf{inv}\ \text{true}$, i.e. the first sequent does not terminate, then

$$\text{Normalize}(p;\ q) = \widetilde{p}$$

else,

$$\begin{array}{l}
\text{Normalize}(p;\ q) = \widetilde{r} = \\
\quad [\![_{\text{V}}\ \sigma^r_{\text{pc}} \cup \sigma_p \cup \sigma_q, C_p \cup C_q, L_p \cup L_q \\
\quad ::\ \ [\![_{\text{R}}\ \{X \mapsto \text{Setzero}(\text{FSequent}(\text{Incrpcs}(\overline{p}, \text{value}(u^q_{\text{pc}}, 1)), \widetilde{q})\ [\!]\ \overline{q})\} \\
\qquad :: u_p \wedge u^r_{\text{pc}} \curvearrowright X \\
\qquad]\!] \\
\quad]\!]
\end{array}$$

where,

1. The valuation σ^r_{pc} defining program counters is given below:

$$\sigma^r_{\text{pc}} \quad = \quad \{i_1 \mapsto \bot, \ldots, i_{\max(\text{Count}(\overline{p}), \text{Count}(\overline{q}))} \mapsto \bot\}$$

   The total number of program counters used in the linear form is thus the maximum of the numbers of program counters in the two sequents.

2. The initialization predicate initializes the program counters according to their initial values in the first sequent incremented by the initial value of

i_1 in the second sequent.

$$\begin{array}{lcl} u^r_{\mathrm{pc}} & = & \mathrm{Incrpcs}(u^p_{\mathrm{pc}}, \mathrm{value}(u^q_{\mathrm{pc}}, 1)) \wedge \\ & & \bigwedge_{i_j \in \mathrm{pcs}(\overline{q}) \setminus \mathrm{pcs}(\overline{p})} i_j = \mathrm{value}(u^p_{\mathrm{pc}}, 1) + \mathrm{value}(u^q_{\mathrm{pc}}, 1) + 1 \end{array}$$

When the number of program counters in the linear form of the second sequent is greater than the number of program counters in that of the first, the program counters that are not used initially are set to an odd value.

3. The function $\mathrm{Incrpcs} : (\mathcal{U}_{\mathrm{pc}} \times \mathbb{N} \to \mathcal{U}_{\mathrm{pc}}) \cup (\overline{\mathcal{P}} \times \mathbb{N} \to \overline{\mathcal{P}})$ takes a predicate on program counters or a process term as the first argument and a natural number as the second argument. It increments the values assigned to the program counters in the given predicate or in the given process term by the given number.

$$\mathrm{Incrpcs}(x, n) \quad = \quad x[i_k = c_k + n / i_k = c_k]_{k \in \mathbb{N}}$$

where c_k is a natural number for all values of k .

4. The function $\mathrm{FSequent} : \overline{\mathcal{P}} \times \widetilde{P} \to \overline{\mathcal{P}}$ takes two process terms that are to be joined in a sequential composition. The function FSequent removes the action predicate $\mathrm{ap}_{\mathrm{pc}}$ from the terminating alternatives of the first argument (which is the first sequent of the sequential composition) and appends the following:

   (a) An action predicate initializing the local model variables of the second sequent according to its initialization predicate, u_q. The jump set of this action predicate consists of the local discrete and continuous variables of the second sequent;

   (b) An action predicate initializing the program counters according to the initialization predicate u^q_{pc} of the second sequent; and

   (c) A deactivation of the program counters of $\overline{p}$ that are not used in $\overline{q}$, in case the first sequent has more program counters than the second sequent; and

   (d) Finally, a recursive call to X;

$$\begin{array}{l} \mathrm{FSequent}(\overline{p}, \widetilde{q}) = \\ \quad \mathrm{Term}(\overline{p}) \ [\quad b_{\mathrm{pc}} \to p_{\mathrm{act}}, \\ \qquad\qquad\quad \{v \mid v \in \mathrm{dom}(\mathrm{Sigma}(\widetilde{q}))\} : U(\widetilde{q}), \\ \qquad\qquad\quad \mathrm{pcs}(\mathrm{rhs}(\widetilde{q})) : U_{\mathrm{pc}}(\widetilde{q}), \\ \qquad\qquad\quad \mathrm{pcs}(\overline{q}) \setminus \mathrm{pcs}(\mathrm{rhs}(\widetilde{q})) : \\ \qquad\qquad\quad \bigwedge_{i_d \in \mathrm{pcs}(\overline{q}) \setminus \mathrm{pcs}(\mathrm{rhs}(\widetilde{q}))} i_d = \mathrm{value}(U_{\mathrm{pc}}(\widetilde{q}), 1) + 1; \ X \\ \qquad\qquad / \quad b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}} \\ \qquad\qquad] \\ \quad [] \ \mathrm{Nonterm}(\overline{p}) \end{array}$$

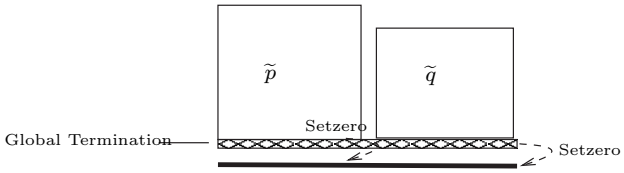


Figure 6: Parallel Composition

5. The function $\text{Setzero} : \overline{\mathcal{P}} \to \overline{\mathcal{P}}$ takes a process term of the form $\overline{p}$. It sets all the program counters of $\overline{p}$ to zero, in the terminating options of $\overline{p}$.

$$\begin{array}{l} \text{Setzero}(\overline{p}) = \\ \quad \text{Term}(\overline{p}) \;[\quad b_{\text{pc}} \to p_{\text{act}}, \\ \qquad\qquad\quad \text{pcs}(\overline{p}) : \bigwedge_{i_d \in \text{pcs}(\overline{p})} i_d = 0 \\ \qquad\qquad /\quad b_{\text{pc}} \to p_{\text{act}}, \text{ap}_{\text{pc}} \\ \qquad\qquad] \\ \quad [] \; \text{Nonterm}(\overline{p}) \end{array}$$

5.5 Parallel Composition

Assume

$$\begin{array}{rl} \text{Normalize}(p_{\text{s}}) = \widetilde{p} = & [\![_{\text{V}}\, \sigma^p_{\text{pc}} \cup \sigma_p, C_p, L_p \\ & :: \; [\![_{\text{R}}\, \{X \mapsto \overline{p}\} :: u_p \wedge u^p_{\text{pc}} \curvearrowright X \,]\!] \\ &]\!] \end{array}$$

and

$$\begin{array}{rl} \text{Normalize}(q_{\text{s}}) = \widetilde{q} = & [\![_{\text{V}}\, \sigma^q_{\text{pc}} \cup \sigma_q, C_q, L_q \\ & :: \; [\![_{\text{R}}\, \{X \mapsto \overline{q}\} :: u_q \wedge u^q_{\text{pc}} \curvearrowright X \,]\!] \\ &]\!] \; . \end{array}$$

Only the linear form of a parallel composition of the form $p_{\text{s}} \parallel q_{\text{s}}$ is given, as parallel composition between process terms of the form $p, q \in \mathcal{P}$ that cannot both be written as a term in $\mathcal{P}_{\text{s}}$ is not allowed in the input language of the algorithm.

We do not reuse the program counters when joining the linear forms $\widetilde{p}$ and $\widetilde{q}$ in parallel composition. We differentiate between the program counters of $\widetilde{p}$ and $\widetilde{q}$ by shifting the subscripts of all program counters in one of the process terms by the number of program counters in the other. We shift the program counters of that process term that has the smallest maximum value for i_1. In this way, in our visualization the first column of blocks has the maximum height. The total number of program counters in a linear form of a parallel composition is the sum of the program counters in the linear forms of the two operands.

Assume $\mathrm{value}(u^q_{\mathrm{pc}}, 1) \leq \mathrm{value}(u^p_{\mathrm{pc}}, 1)$, then,

$$
\begin{aligned}
&\mathrm{Normalize}(p_{\mathrm{s}} \parallel q_{\mathrm{s}}) = \\
&\quad \llbracket_{\mathrm{V}}\, \sigma^p_{\mathrm{pc}} \cup \mathrm{Shiftpcs}(\sigma^q_{\mathrm{pc}}, \mathrm{Count}(\overline{p})) \cup \sigma_p \cup \sigma_q, C_p \cup C_q, L_p \cup L_q \\
&\quad :: \llbracket_{\mathrm{R}}\ \{X \mapsto \mathrm{Setzero} \quad (\quad \mathrm{Extend}(\overline{p}, \mathrm{Shiftpcs}(\overline{q}, \mathrm{Count}(\overline{p})) \\
&\qquad\qquad\qquad\qquad\qquad\qquad [\!] \ \mathrm{Extend}(\mathrm{Shiftpcs}(\overline{q}, \mathrm{Count}(\overline{p})), \overline{p}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad [\!] \ \{\mathrm{COM}(\mathrm{alt}_p, \mathrm{alt}_q) \mid \mathrm{alt}_p \in \mathrm{Alt}(\overline{p}), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathrm{alt}_q \in \mathrm{Alt}(\mathrm{Shiftpcs}(\overline{q}, \mathrm{Count}(\overline{p}))), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathrm{match}(\mathrm{alt}_p, \mathrm{alt}_q) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \} \\
&\qquad\qquad\qquad\qquad) \\
&\qquad :: (u_p \wedge u_q) \wedge (u^p_{\mathrm{pc}} \wedge \mathrm{Shiftpcs}(u^q_{\mathrm{pc}}, \mathrm{Count}(\overline{p}))) \curvearrowright X \\
&\qquad \rrbracket \\
&\quad \rrbracket,
\end{aligned}
$$

where,

1. The function $\mathrm{Shiftpcs} : ((\mathcal{V} \mapsto \Lambda) \times \mathbb{N} \to (\mathcal{V} \mapsto \Lambda)) \cup (\mathcal{U}_{\mathrm{pc}} \times \mathbb{N} \to \mathcal{U}_{\mathrm{pc}}) \cup (\overline{\mathcal{P}} \times \mathbb{N} \to \overline{\mathcal{P}})$ takes as the first parameter a valuation (a set of mappings of variables to values in some value set Λ), or a predicate, or a process term, and as the second parameter a natural number. It shifts the subscripts of all the program counters in the given valuation, predicate or the process term by the given number.

$$\mathrm{Shiftpcs}(x, d) \quad = \quad x[i_{k+d}/i_k]_{k \in \mathbb{N}}$$

2. The symmetric function $\mathtt{match} : \overline{\mathcal{P}} \times \overline{\mathcal{P}} \to \mathcal{B}\mathrm{ool}$, takes as its parameters two alternatives, one from the linear process equation of the linear form of each operand of parallel composition. In case its parameters contain matching send and receive actions, the function match returns true else it returns false. The rules for the function match strip off the unnecessary details from an alternative: the guards, calls to the recursive variable X and all action predicates are removed. The following rules define the function match:

   (a) $\mathrm{match}((b_{\mathrm{pc}} \rightarrow p_{\mathrm{act}}, \mathrm{update}(X_i);\ X), \overline{p}) = \mathrm{match}(p_{\mathrm{act}}, \overline{p})$

   (b) $\mathrm{match}((b_{\mathrm{pc}} \rightarrow p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}};\ X), \overline{p}) = \mathrm{match}(p_{\mathrm{act}}, \overline{p})$

   (c) $\mathrm{match}((b_{\mathrm{pc}} \rightarrow p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}}), \overline{p}) = \mathrm{match}(p_{\mathrm{act}}, \overline{p})$

   (d) $\mathrm{match}(p_{\mathrm{act}}, (p'_{\mathrm{atom}}, \mathrm{ap})) = \mathrm{match}(p_{\mathrm{act}},\ p'_{\mathrm{atom}})$

   (e) $\mathrm{match}(h\,!\,\mathbf{e}_n, W : r,\ h\,?\,\mathbf{x}_n, W : r) = \mathrm{true}$, where W and W' are environment variable sets, r and r' are predicates and h is a communication channel.

   In situations, where none of the rules from above (possibly preceded by interchanging of the arguments) can be applied , the function match returns false.

3. The symmetric function $\mathrm{COM} : \overline{P} \times \overline{P} \to \overline{P} \cup \{\bot\}$ takes as its parameters two alternatives, one from the linear process equation of the linear form of each operand of parallel composition. In case, its parameters contain matching send and receive actions, the function COM returns the result of communication between its parameters. It is a symmetric function. Parallel composition is only defined for process terms of the form p_{s}. The linear form of a p_{s} process term does not consist of any alternative with a pointer (See Section 3 and Section 5.7). Therefore for a parameter containing a pointer, of the form $\mathrm{update}(X_i)$, COM returns $\bot$. The function COM is defined below:

   (a) $\mathrm{COM}((b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{update}(X_i);\ X), \overline{p}) = \bot$, where X_i is a recursion variable.

   (b) In case $\mathrm{match}(\overline{p}, \overline{q})$

$$\begin{aligned}
&\mathrm{COM}((b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}}), (b'_{\mathrm{pc}} \to q_{\mathrm{act}}, \mathrm{ap}'_{\mathrm{pc}})) = \\
&\qquad b_{\mathrm{pc}} \wedge b'_{\mathrm{pc}} \to \mathrm{com}(p_{\mathrm{act}}, q_{\mathrm{act}}), \mathrm{ap}_{\mathrm{pc}}, \mathrm{ap}'_{\mathrm{pc}} \\
&\mathrm{COM}((b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}};\ X), (b'_{\mathrm{pc}} \to q_{\mathrm{act}}, \mathrm{ap}'_{\mathrm{pc}})) = \\
&\qquad b_{\mathrm{pc}} \wedge b'_{\mathrm{pc}} \to \mathrm{com}(p_{\mathrm{act}}, q_{\mathrm{act}}), \mathrm{ap}_{\mathrm{pc}}, \mathrm{ap}'_{\mathrm{pc}}[1/0];\ X \\
&\mathrm{COM}((b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}};\ X), (b'_{\mathrm{pc}} \to q_{\mathrm{act}}, \mathrm{ap}'_{\mathrm{pc}};\ X)) = \\
&\qquad b_{\mathrm{pc}} \wedge b'_{\mathrm{pc}} \to \mathrm{com}(p_{\mathrm{act}}, q_{\mathrm{act}}), \mathrm{ap}_{\mathrm{pc}}, \mathrm{ap}'_{\mathrm{pc}};\ X,
\end{aligned}$$

   (c) In case $\neg\mathrm{match}(\overline{p}, \overline{q})$

$$\mathrm{COM}(\overline{p}, \overline{q}) = \mathsf{inv}\ \mathrm{true},$$

   where the notation $\mathrm{ap}_{\mathrm{pc}}[1/0]$ represents an action predicate $\mathrm{ap}_{\mathrm{pc}}$ with all the predicates setting a program counter to value zero replaced by predicates setting them to 1.

$$\mathrm{ap}_{\mathrm{pc}}[1/0] \quad = \quad \mathrm{ap}_{\mathrm{pc}}[i_k = 1/i_k = 0]_{k \in \mathbb{N}}$$

   In a communication between two actions, where one of the actions is terminating and the other is non-terminating, the resulting communication cannot be a terminating action. A communication action may be a terminating action of $p \parallel q$, only if it is a communication between terminating actions of $\widetilde{p}$ and $\widetilde{q}$. In the communication between a terminating and a non-terminating action, the program counters that are being set to zero in the terminating action must be set to 1 instead. This is done through $\mathrm{ap}_{\mathrm{pc}}[1/0]$.

4. The function $\mathrm{com} : \mathcal{P}_{\mathrm{act}} \times \mathcal{P}_{\mathrm{act}} \to \mathcal{P}_{\mathrm{act}}$ takes two action process terms and returns their communication.

$$\mathrm{com}((h\,!\,e_{\mathbf{n}}, \mathrm{ap}), (h\,?\,x_{\mathbf{n}}, \mathrm{ap}')) = h\,!?\,x_{\mathbf{n}} := e_{\mathbf{n}}, \mathrm{ap}, \mathrm{ap}'$$

   com is a partial function. In our linearization algorithm, the function com is only called with its parameters match according to the function match.

5. The semantics of parallel operator includes details of communication, synchronization and interleaving of actions of parallel components. The communication between parallel components was dealt in the function COM. The interleaving of actions and synchronization of delays is dealt in the function Extend. The function Extend : $\overline{P} \times \overline{P} \rightarrow \overline{P}$ takes the linear process equations of the parallel components. It returns the first argument with some modifications in its alternatives containing terminating actions. Alternatives containing non-final actions and invariant or urgency conditions of are not modified by the function Extend. Termination of a component needs to be specially handled as explained in the paragraph below.

   We recall from section 3 the concepts of local and global termination. A parallel composition terminates when all its components terminate. When one component of a parallel composition terminates while the other components have not terminated yet, then it is called *local* termination of the terminated component. When the last component of a parallel composition terminates, it is called *global* termination. When a process term terminates locally, its program counters are set to 1 (i.e. an odd value) instead of a zero. On global termination, the program counters of all process terms in parallel are set to zero.

   In the function Extend, when a component of parallel composition performs a terminating action, we check whether the process terms in parallel have already terminated. This is done by checking the parity of the product of all program counters of the process terms in parallel. If a program counter of the process terms in parallel still has an even value, i.e. parity of the calculated product is even, then one of the parallel components still has to perform an action. In this case, all the program counters of the terminating component are set to 1 and a recursive call to X is added. Else, if the parity of the calculated product is odd, then the components in parallel have already locally terminated and the action under consideration is indeed the terminating action of the parallel composition.

   The linear form of a parallel composition or the linear form of a process term with parallel composition in its last sequent can be identified by its terminating options. The terminating actions of a linear form of such a process term are guarded by predicates of the form $b_{\text{even}} \wedge \text{Odd}(S)$, where S is a set of program counters. In hybrid χ as is in other process algebras, the parallel operator is defined as a binary operator. A process term $p \parallel q \parallel r$ is defined as $(p \parallel q) \parallel r$ or $p \parallel (q \parallel r)$. (The parallel operator is associative.) While linearizing a process term $(p \parallel q) \parallel r$, in the function Extend the terminating options of the linear form of $(p \parallel q)$ are modified to include the parity checking for the program counters of the linear form of r. This restricts the number of new alternatives that will be added to the linear form of $(p \parallel q)$ to obtain the linear form of $(p \parallel q) \parallel r$ to the size of r. Otherwise not distinguishing that one component of a parallel composition is itself a parallel composition or contains a parallel composition in its last sequent results in an increase in size which is equal to the size of r plus

the number of parallel components in the last sequent.

In case one of the process terms does not terminate, then the terminating alternatives of the process term that terminates are appended with a recursive call to X, because in that case, $p \parallel q$ is non-terminating.

The function $\text{Extend} : \overline{\mathcal{P}} \times \overline{\mathcal{P}} \to (\overline{\mathcal{P}} \cup \{\bot\})$ is defined as follows:

(1) $\text{Extend}(\overline{p} \mathbin{[\!]} \overline{p}', \overline{q}) = \text{Extend}(\overline{p}, \overline{q}) \mathbin{[\!]} \text{Extend}(\overline{p}', \overline{q})$

(2) $\text{Extend}(b_{\text{pc}} \to p_{\text{u}}, \overline{q}) = b_{\text{pc}} \to p_{\text{u}}$

(3) $\text{Extend}((b_{\text{pc}} \to p_{\text{act}}, \text{update}(X_i);\ X), \overline{q}) = \bot$

For the remaining alternatives, two cases for the second parameter are distinguished:

If $\text{Term}(\overline{q}) = \mathsf{inv}\ \text{true}$,

(1) $\text{Extend}((b_{\text{pc}} \to p_{\text{act}}, \text{ap}_{\text{pc}};\ X), \overline{q}) = b_{\text{pc}} \to p_{\text{act}}, \text{ap}_{\text{pc}};\ X$

(2) $\text{Extend}((b_{\text{pc}} \to p_{\text{act}}, \text{ap}_{\text{pc}}), \overline{q}) = b_{\text{pc}} \to p_{\text{act}}, \text{ap}_{\text{pc}}[1/0];\ X$

Else if $\text{Term}(\overline{q}) \neq \mathsf{inv}\ \text{true}$

(1) $\text{Extend}((b_{\text{even}} \to p_{\text{act}}, \text{ap}_{\text{pc}};\ X), \overline{q})$
$= b_{\text{even}} \to p_{\text{act}}, \text{ap}_{\text{pc}};\ X$

(2) $\text{Extend}((b_{\text{even}} \wedge \neg\text{Odd}(S) \to p_{\text{act}}, \text{ap}_{\text{pc}};\ X), \overline{q})$
$= b_{\text{even}} \wedge \neg\text{Odd}(S) \to p_{\text{act}}, \text{ap}_{\text{pc}};\ X$
where $\text{allone}(\text{ap}_{\text{pc}}) \neq \text{true}$.
The function $\text{allone} : \mathcal{AP}_{\text{pc}} \to \mathcal{B}\text{ool}$ returns true if all the program counters in ap_{pc} are being set to 1. The function allone is defined below:

$$\begin{array}{lcl} \text{allone}(W_{\text{pc}} : \bigwedge_{i_d \in W_{\text{pc}}} i_d = 1) & = & \text{true} \\ \text{allone}(W_{\text{pc}} : r_{\text{pc}}, \text{ap}_{\text{pc}}) & = & \text{allone}(W_{\text{pc}} : r_{\text{pc}}) \wedge \text{allone}(\text{ap}_{\text{pc}}) \end{array}$$

(3) $\text{Extend}((b_{\text{even}} \wedge \text{Odd}(S) \to p_{\text{act}}, \text{ap}_{\text{pc}}), \overline{q})$
$= b_{\text{even}} \wedge \text{Odd}(S \cup \text{pcs}(\overline{q})) \to p_{\text{act}}, \text{ap}_{\text{pc}}$

(4) $\text{Extend}((b_{\text{even}} \wedge \neg\text{Odd}(S) \to p_{\text{act}}, \text{ap}_{\text{pc}};\ X), \overline{q})$
$= b_{\text{even}} \wedge \neg\text{Odd}(S \cup \text{pcs}(\overline{q})) \to p_{\text{act}}, \text{ap}_{\text{pc}};\ X,$
where $\text{allone}(\text{ap}_{\text{pc}}) = \text{true}$.

(5) $\text{Extend}((b_{\text{even}} \to p_{\text{act}}, \text{ap}_{\text{pc}}), \overline{q})$
$= \text{Extend}((b_{\text{even}} \wedge \text{Odd}(\text{pcs}(\overline{q})) \to p_{\text{act}}, \text{ap}_{\text{pc}}), \overline{q})$
$\mathbin{[\!]} \text{Extend}((b_{\text{even}} \wedge \neg\text{Odd}(\text{pcs}(\overline{q})) \to p_{\text{act}}, \text{ap}_{\text{pc}}[1/0];\ X), \overline{q})$

In the last three items, parity checking of the program counters of $\overline{q}$ is added to the terminating options of $\overline{p}$. The last item is applicable when $\overline{p}$ is a linear form of a process term whose last sequent does not contain

a parallel composition. Examples of process terms with their last sequent without a parallel composition are:

$$a;\ b;\ c \quad a \mathbin{[\!]} b \quad (p_1 \parallel p_2);\ a \mathbin{[\!]} b$$

When these process terms appear as a component in parallel composition, the last item is applicable.

The two cases former to the last item are applicable when $\overline{p}$ contains a parallel composition in its last sequent.

$$p_1 \parallel p_2 \quad a;\ (p_1 \parallel p_2) \quad (p_1 \parallel p_2) \mathbin{[\!]} b$$

Only when $\overline{p}$ contains a parallel composition in its last sequent, an alternative guarded by a predicate $b_{\text{even}} \wedge \neg\text{Odd}(S)$ sets program counters to values 1. If $\overline{p}$ is linear form of a term $(p_1 \parallel q_1);\ p_2$, then an alternative guarded by $b_{\text{even}} \wedge \neg\text{Odd}(S)$ sets the program counters to some value higher than 1.

5.6 Alternative Composition

The alternative composition of process terms provides a choice between them. The choice is resolved as soon as an action is performed, in favor of the process term the action of which has been executed. A graphical representation of two process terms and their alternative composition is given in Figure 7. As shown in the figure, there are two ways in which two process terms, p and q, can be alternatively composed:

1. In Figure 7(b), the roots of the two process terms are merged to obtain a root for their alternative composition. Transitions emerging from this root are the same as the transitions emerging from the roots of p and q.

2. In Figure 7(c), a new root for the resulting alternative composition is created, which is distinct from the roots of the given alternatives. Transitions emerging from the new root end at proper places within the transition trees of p and q. Note that the original roots of the alternatives are retained in this way of alternative composition but these roots are no longer initial states.

Merging two roots to obtain a new root for the alternative composition works only if the operand process terms do not have self recursion and none of the process terms have initial parallelism. To explain further, we present scenarios of self recursion and initial parallelism in operands of an alternative composition below:

In terms of transition systems, self recursion means that there is a transition emerging from within the tree of a process term and ending at its root. When roots of operands are merged to form the root of the alternative composition, then a transition ending at the root of alternative composition activates both operands which is not intended.

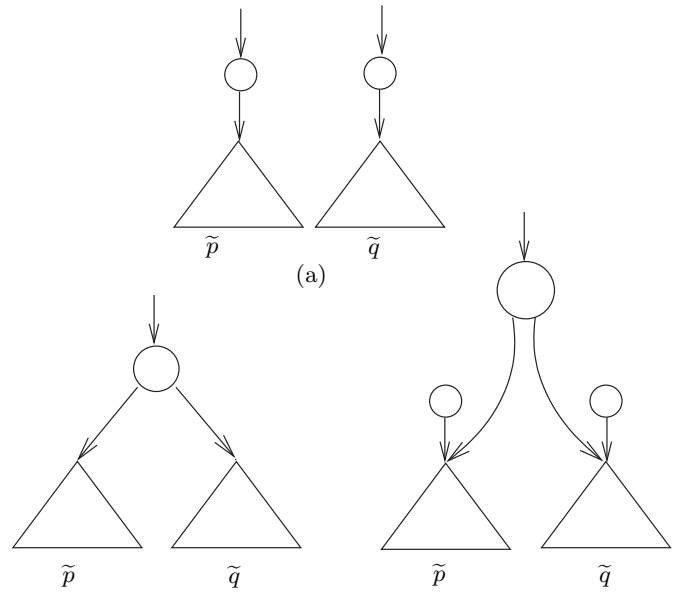


Figure 7: Two Techniques for alternative composition

In a linear form of $p \,[\!]\, q$, the initial values of program counters activate the initial options of both p and q. Consider the case where one of the operands, has self recursion, for example let

$$q = \|_{\mathrm{R}} \{X_1 \mapsto a; b; X_1\} :: X_1 \,\|$$

In the linear form of $\widetilde{q}$, after actions a and b, a program counter of q will be set back to its initial value. In $p \,[\!]\, q$ obtained by merging the roots of operands, if q is chosen, then resetting a program counter to its initial value activates the initial options of p also. This problem does not arise when a new root is created for the alternative composition. In alternative composition with a new root, the initial values of program counters in $p \,[\!]\, q$ are distinct from their initial values in p and q. The value to which a program counter is reset in case of self recursion is not its initial value in the alternative composition, but its initial value in the operand with self recursion.

To observe the initial parallelism in $p \,[\!]\, q$, let $q = a;\ b \parallel c;\ d$. The term q can start with either performing action a or b. In terms of our linear form, there are two program counters i_1 and i_2 that are active initially in $\widetilde{q}$. Therefore also in the linear form of alternative composition $p \,[\!]\, q$ at least two program counters are initially active. If process q is chosen from the alternative composition $p \,[\!]\, q$, then when the first action of $\widetilde{q}$ is performed, one of its program counters is decremented, whereas the other program counter is still at its initial value. If

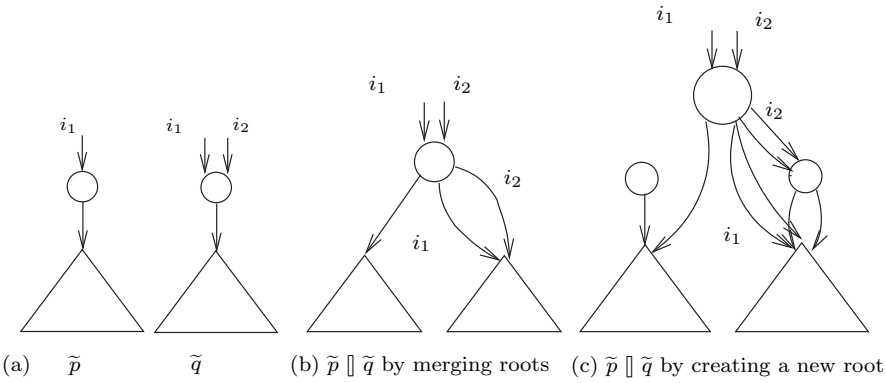


Figure 8: Alternative Composition with initial parallelism in q

the roots of p and q are merged to form the root of the alternative composition, then in $p \,[\!]\, q$, after doing an action of q, an action of $\widetilde{p}$ is still possible. See Figure 8(b). If first the action of $\widetilde{q}$ governed by program counter i_2 is performed, then after the action, i_1 is still at its initial value and can activate an option of $\widetilde{p}$.

When we create a new root for $p \,[\!]\, q$, then after executing an action of q, all program counters except the one governing the action executed, are reset according to the root of $\widetilde{q}$, i.e. according to the initial values of program counters in the linear form of q. This shown by in the figure 8 (c).

The algorithm that creates a new root is a general purpose algorithm but it yields unreachable states, incase there is no self-recursion or initial parallelism. Therefore, we give in this section two algorithms for alternative composition, one that merges the roots of operands to obtain the root of alternative composition and the other that creates a new root for alternative composition. Depending on the scenario at hand, different algorithm for alternative composition can be adopted.

Assume

$$\begin{aligned}\text{Normalize}(p) = \widetilde{p} = \;& [\![_{\mathrm{V}}\, \sigma^p_{\mathrm{pc}} \cup \sigma_p, C_p, L_p \\ & :: [\![_{\mathrm{R}}\, \{X \mapsto \overline{p}\} :: u_p \wedge u^p_{\mathrm{pc}} \curvearrowright X \,]\!] \\ &]\!]\end{aligned}$$

and

$$\begin{aligned}\text{Normalize}(q) = \widetilde{q} \;& [\![_{\mathrm{V}}\, \sigma^q_{\mathrm{pc}} \cup \sigma_q, C_q, L_q \\ & :: [\![_{\mathrm{R}}\, \{X \mapsto \overline{q}\} :: u_q \wedge u^q_{\mathrm{pc}} \curvearrowright X \,]\!] \\ &]\!],\end{aligned}$$

5.6.1 Alternative Composition without a new root

This algorithm is applicable if the linear forms of both p and q have one program counter initially active and none of the operands have self recursion. The linear forms $\widetilde{p}$ and $\widetilde{q}$ are tested for initial parallelism and self recursion as follows:

1. If only the value of i_1 is even in the initialization predicate of a linear form, then the linear form does not have initial parallelism. Thus for absence of initial parallelism in $\widetilde{p}$ and $\widetilde{q}$, the following predicates should be true. :

$$\text{value}(u^p_{\text{pc}}, 1) \mod 2 = 0 \wedge \text{Odd}(\text{pcs}(\overline{p}) \backslash \{i_1\})$$

and

$$\text{value}(u^q_{\text{pc}}, 1) \mod 2 = 0 \wedge \text{Odd}(\text{pcs}(\overline{q}) \backslash \{i_1\})$$

2. We look at the allowed syntax of the input language $\mathcal{P}$ to the algorithm. Let $p \in \mathcal{P}$, then:

$$\begin{array}{rcl} p & ::= & p_{\text{s}} \\ & | & p_{\text{s}}; X_i \\ & | & p_{\text{s}}; p \\ & | & p \mathbin{[]} p \end{array}$$

The occurrence of a recursion variable is always guarded in an input process term. An operand of alternative composition can have self recursion, only if it is a recursion scope operator.

Consider the following example of a recursion scope operator:

$$\begin{array}{l} [\![_{\text{R}} \ \{ \quad X_1 \mapsto a;\ X_1, \\ \qquad\quad X_2 \mapsto b;\ X_2, \\ \qquad\quad X_3 \mapsto a;\ X_1 \mathbin{[]} b;\ X_2 \\ \quad \} \\ :: X_3 \\ ]\!] \end{array}$$

Despite recursion in the process definition of X_3, we use the algorithm without a new root to linearize its process definition. Although semantically, $X_3 \leftrightarrows X_1 \mathbin{[]} X_2$, but due to the syntactic difference, there is no loop to the initial states of X_1 and X_2. The initial options of X_3 are activated by different values of program counters than those for the initial options of X_1 and X_2. Thus a new root is always automatically created for the alternative composition due to guarded occurrence of X_2 and X_3.

To test for self recursion in a linear form, we look at all the alternatives of the linear process equation of a given linear form. In case one of the alternatives sets a program counter to the value given in the initialization predicate of the linear form, then the process term has self recursion.

Below we define a function TestforRec : $\widetilde{P} \rightarrow \mathcal{B}ool$ that checks for self recursion in a linear form:

$$\text{TestforRec}(\widetilde{p}) = \begin{cases} \text{true} & \exists b_{\text{pc}} \rightarrow p_{\text{act}}, \text{ap}_{\text{pc}};\ X \in \text{Alt}(\text{rhs}(\widetilde{p})) \\ & \wedge \text{matchvalue}(\text{ap}_{\text{pc}}, U_{\text{pc}}(\widetilde{p})) \\ \text{false} & \text{otherwise} \end{cases}$$

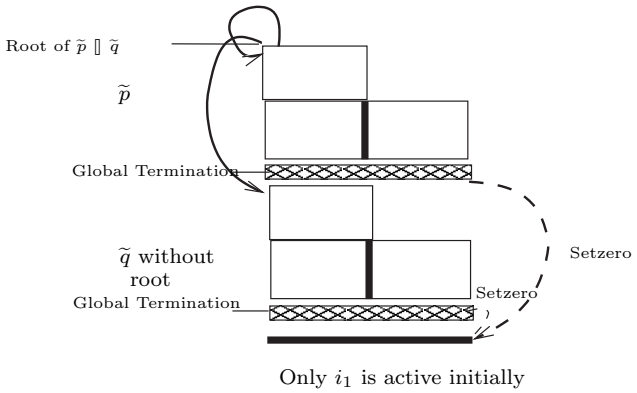


Figure 9: Alternative Composition without a new root

where the function matchvalue : $\mathcal{AP}_{\rm pc} \times \mathcal{U}_{\rm pc} \rightarrow \mathcal{B}ool$ takes an action predicate and an initialization predicate of a linear form. It returns true if the given action predicate is setting a program counter according to the value of the program counter in the given initialization predicate.

$$\begin{aligned} \text{matchvalue}(W_{\rm pc} : r_{\rm pc}, u_{\rm pc}) &= \begin{cases} \text{true} & \exists i_d \in W_{\rm pc} \wedge \\ & r_{\rm pc} \implies \\ & (i_d = \text{value}(u_{\rm pc}, d)) \\ \text{false} & \text{otherwise} \end{cases} \\ \text{matchvalue}((W_{\rm pc} : r_{\rm pc}, \text{ap}_{\rm pc}), u_{\rm pc}) &= \text{matchvalue}(W_{\rm pc} : r_{\rm pc}, u_{\rm pc}) \\ & \vee \text{matchvalue}(\text{ap}_{\rm pc}, u_{\rm pc}) \end{aligned}$$

where $W_{\rm pc}$ is a set of program counters, $r_{\rm pc}$ is a predicates on program counters and $\text{ap}_{\rm pc}$ is an action predicate on program counters.

Note that in the definition of TestforRec, we do not check the terminating alternatives of a process term nor the alternatives with pointers update(X_i). A terminating option does not have recursion. We know that an operand of alternative composition has self recursion only when it is a recursion scope operator. An alternative with a pointer update(X_i) appears in an intermediate form during the linearization of a process term of the form $p_{\rm s}; X_i$ (see Section 5.7). The pointers of the form *update*(X_i) are not present in the final linear form of a recursion scope operator.

While joining two process terms in alternative composition, as is done in sequential composition, see Section 5.4, the values of the program counters common in the linear forms of operands are made distinct from each other by incrementing the values of program counters in one of the operands. For linearizing $p \parallel q$, we can without loss of generality, decide to increment the values of program counters in $\widetilde{p}$. In the algorithm for alternative composition without a new root,

we increment the values of all program counters in one operand by the maximum value of the program counter i_1 in the other operand minus 2. The reason for doing this is that in this algorithm, the root (i.e. initial options) of $\widetilde{q}$ is moved (i.e. incremented) to the same level (i.e. value of program counter i_1) as the root of $\widetilde{p}$ after incrementing. This leaves behind a gap in the value of the program counter i_1 at the border of $\widetilde{p}$ and $\widetilde{q}$. (This is different from sequential composition, where there is no such gap in the values of program counter i_1). Incrementing the program counter i_1 in $\widetilde{p}$ by maximum value of i_1 in $\widetilde{q}$ minus 2, brings the alternative guarded by predicate $i_1 = 2$ in $\widetilde{p}$ to the same level as the initial options of $\widetilde{q}$. i.e. after incrementing, the predicate $i_2 = 2$ in $\widetilde{p}$ is replaced by $i_1 = \text{value}(u^q_{\text{pc}}, 1)$. But the value $\text{value}(u^q_{\text{pc}}, 1)$, will not be used to guard the initial option of q in the linear form of $p \mathbin{[\!]} q$, because the root of q has to be moved to the same level as that of p. Therefore, no overlap of program counter values guarding the options of $\widetilde{p}$ and $\widetilde{q}$ occurs.

$$
\begin{array}{l}
\text{Normalize}(p \mathbin{[\!]} q) = \widetilde{r} = \\
\quad [\![_{\text{V}}\, \sigma^r_{\text{pc}} \cup \sigma_p \cup \sigma_q, C_p \cup C_q, L_p \cup L_q \\
\quad :: \ [\![_{\text{R}}\ \{X \mapsto \text{Setzero} \quad (\quad \text{Incrpcs}^{>1}(\overline{p}, \text{value}(u^q_{\text{pc}}, 1) - 2) \\
\qquad\qquad\qquad\qquad\qquad\quad [\!] \ \text{IncrInitialpcs}(\widetilde{q}, \text{value}(u^p_{\text{pc}}, 1) - 2) \\
\qquad\qquad\qquad\qquad\quad) \\
\qquad\quad \} \\
\qquad :: u_p \wedge u_q \wedge u^r_{\text{pc}} \curvearrowright X \\
\qquad]\!] \\
\quad]\!],
\end{array}
$$

where,

1. The valuation σ^r_{pc} defining program counters is given below:

$$\sigma^r_{\text{pc}} \quad = \quad \{i_1 \mapsto \bot, \ldots, i_{\max(\text{Count}(\overline{p}), \text{Count}(\overline{q}))} \mapsto \bot\}$$

   The total number of program counters in the alternative composition is the maximum of the numbers of program counters in the two operands.

2. The initialization predicate u^r_{pc} initializing the program counters is as follows:

$$
\begin{array}{l}
u^r_{\text{pc}} = (i_1 = \text{value}(u^p_{\text{pc}}, 1) + \text{value}(u^q_{\text{pc}}, 1) - 2) \wedge \\
\quad \bigwedge_{1 < k \leq \max(\text{Count}(\overline{p}), \text{Count}(\overline{q}))} i_k = \text{value}(u^p_{\text{pc}}, 1) + \text{value}(u^q_{\text{pc}}, 1) - 1
\end{array}
$$

   Initially only program counter i_1 is active.

3. The function $\text{Incrpcs}^{>1} : (\text{predicate} \times \mathbb{N} \to \text{predicate}) \cup (\overline{\mathcal{P}} \times \mathbb{N} \to \overline{\mathcal{P}}) \cup (\widetilde{P} \times \mathbb{N} \to \widetilde{P})$ takes a predicate or a process term as the first parameter, and a natural number as the second parameter. It increments the values (greater than 1) assigned to the program counters in the given predicate or the given process term by the given number.

$$\text{Incrpcs}^{>1}(x, n) \quad = \quad x[i_k = c_k + n / i_k = c_k]_{k \in \mathbb{N}, c_k > 1}$$

where $c_k > 1$ for all values of k. The zero and 1 values of program counters are not incremented, as they indicate the (final) terminating actions of an operand.

In a linear form $\widetilde{p}$, program counters are assigned values in the initialization predicate $U_{\mathrm{pc}}(\widetilde{p})$ and in the right hand side of the recursion definition of $\widetilde{p}$, i.e. $\mathrm{rhs}(\widetilde{p})$. (See Section 5.1 for the definition of rhs). $\mathrm{Incrpcs}^{>1}(\widetilde{p}, n)$ increments the non zero values assigned to program counters in both these constructs of $\widetilde{p}$.

4. The function $\mathrm{IncrInitialpcs} : \widetilde{P} \times \mathbb{N} \to \overline{\mathcal{P}}$ takes a process term of the form $\widetilde{p}$ and a natural number. It increments the initial value of i_1 in the right hand side of the recursion definition of $\widetilde{p}$ by the number given.

$$\mathrm{IncrInitialpcs}(\widetilde{p}, n) = \mathrm{rhs}(\widetilde{p}) \quad \begin{array}{ll} [& i_1 = \mathrm{value}(U_{\mathrm{pc}}(\widetilde{p}), 1) + n \\ / & i_1 = \mathrm{value}(U_{\mathrm{pc}}(\widetilde{p}), 1) \\ ] & \end{array}$$

(See Section 5.1 for the definition of rhs).

This algorithm is used for linearizing alternative compositions with operands lacking self recursion. As self recursion is excluded, therefore an initially active program counter, particularly i_1 (when there is no initial parallelism), will never be reset back to its initial value. This means that the initial value of i_1 only occurs in the guards of the operand process terms. We make use of this fact in the definition of the function IncrIntialpcs and increment all occurrences of the initial value of i_1.

The function IncrIntialpcs makes available the initial options of $\widetilde{q}$ by setting the value of i_1 in $\widetilde{q}$ to the initial value of i_1 in $\mathrm{Normalize}(p \between q)$.

5.6.2 Alternative Composition with a new root

This algorithm can be used to alternatively compose linear process terms with initial parallelism and self recursion. We create a new root for the alternative composition. Initially we only activate program counter i_1 in the alternative composition. The initial parallelism in the operands, if present, is captured by options guarded by i_1. In case of parallelism, more than one option is initially available. For each initial option in the given operands p and q, an option guarded by $i_1 = \mathrm{value}(u^p_{\mathrm{pc}}, 1) + \mathrm{value}(u^q_{\mathrm{pc}}, 1) + 2$, is added in the alternative composition. Hence a new root is created by a new value for the program

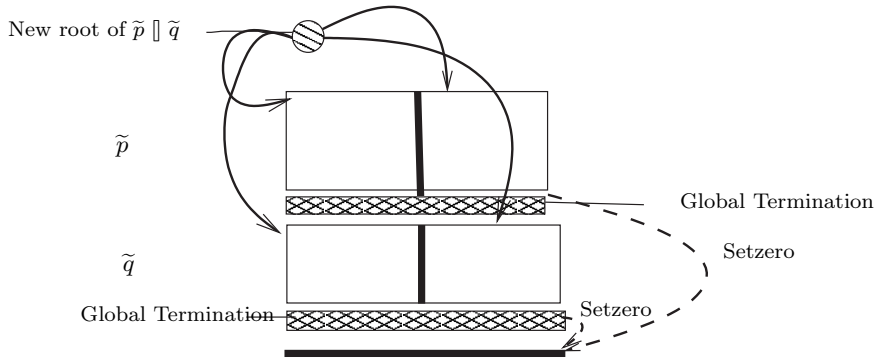


Figure 10: Alternative Composition with a new root

counter i_1 which is equal to the sum of its maximum values in $\widetilde{p}$ and $\widetilde{q}$ plus 2.

$$
\begin{array}{l}
\text{Normalize}(p \,[\!]\, q) = \widetilde{r} = \\
\quad [\![_{\text{V}}\ \sigma^r_{\text{pc}} \cup \sigma_p \cup \sigma_q, C_p \cup C_q, L_p \cup L_q \\
\quad :: \ [\![_{\text{R}}\ \{X \mapsto \text{Setzero} \quad (\quad \text{Incrpcs}^{>1}(\overline{p}, \text{value}(u^q_{\text{pc}}, 1)) \\
\qquad\qquad\qquad\qquad\qquad\quad [\!]\ \overline{q} \\
\qquad\qquad\qquad\qquad\qquad\quad [\!]\ \text{Createnewroot}(m_{u_{pq}}, \text{Incrpcs}^{>1}(\widetilde{p}, \text{value}(u^q_{\text{pc}}, 1))) \\
\qquad\qquad\qquad\qquad\qquad\quad [\!]\ \text{Createnewroot}(m_{u_{pq}}, \widetilde{q}) \\
\qquad\qquad\qquad\qquad\quad) \\
\qquad\quad \} \\
\qquad :: u_p \wedge u_q \wedge u^r_{\text{pc}} \curvearrowright X \\
\qquad]\!] \\
\quad]\!],
\end{array}
$$

where,

1. The notation $m_{u_{pq}}$ is an abbreviation for $\text{value}(u^p_{\text{pc}}, 1) + \text{value}(u^q_{\text{pc}}, 1) + 2$.

2. The valuation σ^r_{pc} defining program counters is given below:

$$\sigma^r_{\text{pc}} \quad = \quad \{i_1 \mapsto \bot, \ldots, i_{\max(\text{Count}(\overline{p}), \text{Count}(\overline{q}))} \mapsto \bot\}$$

   The total number of program counters in the alternative composition is the maximum of the numbers of program counters in the two operands.

3. The initialization predicate u^r_{pc} initializing the program counters is as follows:

$$
\begin{array}{rcl}
u^r_{\text{pc}} & = & (i_1 = \text{value}(u^p_{\text{pc}}, 1) + \text{value}(u^q_{\text{pc}}, 1) + 2) \wedge \\
& & \bigwedge_{1 < k \leq \max(\text{Count}(\overline{p}), \text{Count}(\overline{q}))} \\
& & \qquad\qquad (i_k = \text{value}(u^p_{\text{pc}}, 1) + \text{value}(u^q_{\text{pc}}, 1) + 1)
\end{array}
$$

   Except for i_1, all program counters are set to odd values.

4. The function Createnewroot : $\mathbb{N} \times \widetilde{P} \to \overline{\mathcal{P}}$ returns part of the new root that is created for the alternative composition. The first parameter of Createnewroot is $m_{u_{pq}}$, the sum of the initial values of i_1 in $\widetilde{p}$ and in $\widetilde{q}$, +2. The second parameter is linear form of one of the operands p or q. This function makes available the initial options of the given operand in the linear form of $p \mathbin{[\!]} q$. This is done by taking all the initially active options in the given linear process term and replacing their guard predicates by guards setting i_1 to $m_{u_{pq}}$. After performing the first action, the program counters are initialized according to the initialization predicate of the given operand.

$$
\begin{array}{l}
\text{Createnewroot}(m, \widetilde{p}) = \\
\quad \Big[\!\Big] \quad \{ \quad i_1 = m \to p_{\mathrm{u}} \mid b_{\mathrm{pc}} \to p_{\mathrm{u}} \in \mathrm{Alt}(\overline{p}) \wedge U_{\mathrm{pc}}(\widetilde{p}) \implies b_{\mathrm{pc}} \} \\
\quad \Big[\!\Big] \quad \{ \quad i_1 = m \to p_{\mathrm{act}}, \mathrm{update}(X_i);\ X \\
\qquad \mid \quad b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{update}(X_i);\ X \in \mathrm{Alt}(\overline{p}) \wedge U_{\mathrm{pc}}(\widetilde{p}) \implies b_{\mathrm{pc}} \\
\qquad \} \\
\quad \Big[\!\Big] \quad \{ \quad i_1 = m \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}}, (\mathrm{pcs}(\mathrm{rhs}(\widetilde{p})) \backslash \mathrm{pcs}(\mathrm{ap}_{\mathrm{pc}})) : \\
\qquad\quad \bigwedge_{i_d \in (\mathrm{pcs}(\mathrm{rhs}(\widetilde{p})) \backslash \mathrm{pcs}(\mathrm{ap}_{\mathrm{pc}}))} i_d = \mathrm{value}(U_{\mathrm{pc}}(\widetilde{p}, d));\ X \\
\qquad \mid \quad b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}};\ X \in \mathrm{Alt}(\overline{p}) \wedge U_{\mathrm{pc}}(\widetilde{p}) \implies b_{\mathrm{pc}} \\
\qquad \} \\
\quad \Big[\!\Big] \quad \{ \quad i_1 = m \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}} \mid b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}} \in \mathrm{Alt}(\overline{p}) \wedge \\
\qquad\quad U_{\mathrm{pc}}(\widetilde{p}) \implies b_{\mathrm{pc}} \\
\qquad \}
\end{array}
$$

In an alternative composition, after doing an action of one alternative, it is not possible to do an action of the other alternative. Therefore, after performing the first action, program counters are reset according to the initialization predicate of the given operand.

5.7 Recursion Scope operator

In the input language to the algorithm, recursion variables are only allowed in a recursion scope operator. Only complete recursion definitions are allowed. i.e. the top-level process term as well as any process definition of a recursion variable may not mention any recursion variable not defined within the same scope.

We mentioned in the section Output form of the algorithm (Section 3) that the pointer update(X_i) appears only in the intermediate linear form during linearization of a recursion scope operator. Recall from Section 2 the allowed syntax for process definitions of recursion variables. The set of process defini-

tions P for recursion variables, with $p \in P$ is defined as,

$$\begin{array}{rcl} p & ::= & p_{\mathrm{s}} \\ & | & p_{\mathrm{s}}; X_i \\ & | & p_{\mathrm{s}}; p \\ & | & p \mathbin{[\!]} p \end{array}$$

In sections 5.2 to 5.6, we have defined how to linearize different process terms from the set $\mathcal{P}$. The linearization of a process term of the form $p_{\mathrm{s}};\ X_i$ was not defined. It is defined later in this section. The pointer update(X_i) is introduced during the linearization of a process definition of the form $p_{\mathrm{s}};\ X_i$. When linearizing such a process definition, we come across recursion variables names whose linear forms we may not know yet. For example consider the following process term:

$$\begin{array}{l} [\![_{\mathrm{R}}\, \{X_1 \mapsto p_{\mathrm{s}};\ X_2, X_2 \mapsto q_{\mathrm{s}};\ X_3, X_3 \mapsto r_{\mathrm{s}};\ X_1\} \\ :: X_1 \\ ]\!] \end{array}$$

In linearization of such a recursion scope, as we may not yet know the linear form of the recursion variable being referred to, we place a pointer update(X_i) in the terminating options of the linear form of the process term referring to the recursion variable X_i. The pointer update(X_i) is later replaced by some action predicates according to the linear form of the process definition of variable X_i, after all recursion variables in a given recursion scope operator have been partially linearized.

Recall that the restricted form of the recursion scope operator process term is defined by:

$$\begin{array}{rcll} p_{\mathrm{R}} & ::= & [\![_{\mathrm{R}}\, R :: X_i\,]\!] & \mathrm{Complete}(R) \wedge X_i \in \mathrm{dom}\, R \\ & | & [\![_{\mathrm{R}}\, R :: p\,]\!] & \mathrm{Complete}(R) \wedge \mathrm{Recvars}(p) \in \mathrm{dom}\, R \end{array}$$

In this section we give a linearization algorithm for a recursion scope of the form $[\![_{\mathrm{R}}\, R :: X_i\,]\!]$ only. A recursion scope operator of the form $[\![_{\mathrm{R}}\, R :: p\,]\!]$ can always be transformed into the recursion scope of the form $[\![_{\mathrm{R}}\, R :: X_i\,]\!]$ as follows:

- Introduce a new recursion variable in R and define its right hand side to be equal to p;
- rewrite the recursion scope by adding the new definition to R and replacing p by the new recursion variable

If a recursion scope operator of the form $[\![_{\mathrm{R}}\, R :: p\,]\!]$ is given in the input to the algorithm, we first transform it and then linearize it. Therefore in the linearization algorithm,

$$\mathrm{Normalize}([\![_{\mathrm{R}}\, R :: p\,]\!]) = \mathrm{Normalize}([\![_{\mathrm{R}}\, R \cup \{X_{|\mathrm{dom}\, R|+1} \mapsto p\} :: X_{|\mathrm{dom}\, R|+1}\,]\!]),$$

where $|\ \mathrm{dom}\, R\ |$ denotes the number of recursion variables in $\mathrm{dom}\, R$.

5.7.1 A restriction on process definitions of recursion variables

Consider the following recursion definition.

$$\{X_1 \mapsto [\![_{\mathrm{V}}\, \sigma, C, L :: p\,]\!]\ ;\ X_1\}$$

We can view X_1 as an infinite sequence of variable scopes $[\![_{\mathrm{V}}\, \sigma, C, L :: p\,]\!]$, with each scope having a new instance of local variables $\mathrm{dom}(\sigma), C$ and L. In the linear form, (see section 3), a variable scope is only present at the top-level. All local variable definitions and valuations of any variable scopes present in the input to the linearization algorithm are moved to the top-level. It is not possible to linearize a process definition as that of X_1 in the current linearization algorithm since it requires infinite instances of local variables. See Section 5.12 for a detailed discussion of the problem. We disallow in the input to the algorithm, recursion variables with definitions consisting of a variable scope operator followed by self recursion.

5.7.2 Linearization of $p_{\mathrm{s}}; X_i$

Assume

$$\begin{array}{rcl} \mathrm{Normalize}(p_{\mathrm{s}}) = \widetilde{p} & = & [\![_{\mathrm{V}}\, \sigma_{\mathrm{pc}} \cup \sigma_{\perp}, C, L \\ & & ::\ \ [\![_{\mathrm{R}}\, \{X \mapsto \overline{p}\} :: u_p \wedge u_{\mathrm{pc}} \curvearrowright X\,]\!] \\ & &]\!] \end{array}$$

The number of program counters and the initialization predicates of the linear form of $p_s;\ X_i$, for some recursion variable X_i, are the same as for the linear form of p_{s}. The terminating options of p_{s} are modified to include a pointer to the recursion variable X_i.

$$\begin{array}{rcl} \mathrm{Normalize}(p_{\mathrm{s}};\ X_i) & = & [\![_{\mathrm{V}}\, \sigma_{\mathrm{pc}} \cup \sigma, C, L \\ & & ::\ \ [\![_{\mathrm{R}}\, \{X \mapsto \mathrm{RFSequent}(\overline{p}, X_i)\} :: u_p \wedge u_{\mathrm{pc}} \curvearrowright X\,]\!] \\ & &]\!], \end{array}$$

where the function $\mathrm{RFSequent} : \overline{\mathcal{P}} \times \mathcal{X} \to \overline{\mathcal{P}}_{\mathrm{R}}$ takes a process term of the form $\overline{p}$ and a recursion variable. It removes the action predicate updating program counters from the terminating options of $\overline{p}$ and appends them by a pointer update(X_i) and a recursive call to X.

$$\begin{array}{rcl} \mathrm{RFSequent}(\overline{p}, X_i) & = & \mathrm{Term}(\overline{p})[b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{update}(X_i);\ X \\ & & \quad /b_{\mathrm{pc}} \to p_{\mathrm{act}}, \mathrm{ap}_{\mathrm{pc}} \\ & & \quad] \\ & & [\!]\ \mathrm{Nonterm}(\overline{p}) \end{array}$$

5.7.3 Linearization of $[\![_{\mathrm{R}}\, \{X_1 \mapsto p_1, \ldots, X_n \mapsto p_n\} :: X_m\,]\!]$

In the linearization of a recursion scope operator, we reuse program counters in the linear form of process definitions of recursion variables. The total number

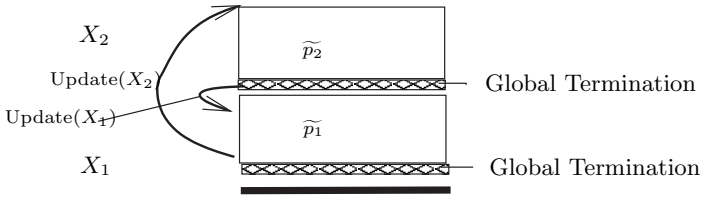


Figure 11: A set of recursion definitions

of program counters is equal to the highest number of program counters used in any process definition of a recursion variable. Since recursion variables can only appear at the end of a process definition, therefore updating counters is easy.

We follow the following steps in the linearization of a recursion scope operator:

1. Linearize the righthand sides of the definitions of all recursion variables;
2. Make the values of program counters that are common among the linear forms of recursion variables distinct from each other. This is done by incrementing the values of all program counters in the linear form of a recursion variable X_j, with $j > 1$, by the sum of maximum values of i_1 in linear forms of X_1 to X_{j-1}. The linear form of X_1 is not incremented;
3. Replace the expressions update(X_i) by action predicates setting program counters and model variables according to the initialization predicates of the linear form of X_i;
4. Alternatively compose the linear forms of the recursion definitions of all recursion variables;
5. Finally set all the program counters to zero in the terminating options of the alternative composition thus obtained.

Assume

$$\begin{aligned}\text{Normalize}(p_1) = \widetilde{p}_1 \quad = \quad & [\![_{\mathrm{V}}\ \sigma^1_{\mathrm{pc}} \cup \sigma_1, C_1, L_1 \\ & ::\ \ [\![_{\mathrm{R}}\ \{X \mapsto \overline{p}_1\} :: u_1 \wedge u^1_{\mathrm{pc}} \curvearrowright X\ ]\!] \\ &]\!] \\ & \quad \ldots \end{aligned}$$

$$\begin{aligned}\text{Normalize}(p_n) = \widetilde{p}_n \quad = \quad & [\![_{\mathrm{V}}\ \sigma^n_{\mathrm{pc}} \cup \sigma_n, C_n, L_n \\ & ::\ \ [\![_{\mathrm{R}}\ \{X \mapsto \overline{p}_n\} :: u_n \wedge u^n_{\mathrm{pc}} \curvearrowright X\ ]\!] \\ &]\!] \end{aligned}$$

Then,

$$\begin{array}{l}\text{Normalize}(\llbracket_{\text{R}}\,\{X_1 \mapsto p_1, \ldots X_n \mapsto p_n\} :: X_m\,\rrbracket) = \\ \llbracket_{\text{V}}\,\sigma_{\text{pc}} \cup \bigcup_{1\le j\le n}\sigma_j, \bigcup_{1\le j\le n} C_j, \bigcup_{1\le j\le n} L_j \\ :: \quad \llbracket_{\text{R}}\,\{X \mapsto \text{Setzero}(\bigsqcup_{1\le j\le n} \text{Update}(j, [\widetilde{p}_1, \ldots, \widetilde{p}_n]))\} \\ \quad :: u_m \wedge u_{\text{pc}} \curvearrowright X \\ \quad \rrbracket \\ \rrbracket,\end{array}$$

where

1. $1 \le m \le n$

2. Let maxpc denote the highest number of program counters used in any of the linear forms, $\widetilde{p}_1 \ldots \widetilde{p}_n$. Then,

$$\text{maxpc} = \max(\bigcup_{1\le j\le n} \{\text{Count}(\overline{\text{p}}_{\text{j}})\})$$

   The valuation σ_{pc} defining the program counters is as follows:

$$\sigma_{\text{pc}} \quad = \quad \{i_1 \mapsto \bot, \ldots, i_{\text{maxpc}} \mapsto \bot\}$$

3. The initialization predicate u_{pc} initializing the program counters is as follows:

$$\begin{array}{lcl} u_{\text{pc}} & = & \text{Incrpcs}^{>1}(u^m_{\text{pc}}, \text{Incrvalue}(m, [\widetilde{p}_1, \ldots, \widetilde{p}_n]))\wedge \\ & & \bigwedge_{d\in[1,\ldots,\text{maxpc}]\setminus[1,\ldots,\text{Count}(p_m)]} \\ & & i_d = \text{value}(\text{Incrpcs}^{>1}(u^n_{\text{pc}}, \text{Incrvalue}(n, [\widetilde{p}_1, \ldots, \widetilde{p}_n])), 1) + 1 \end{array}$$

   where the function $\text{Incrvalue} : \mathbb{N} \times \widetilde{P}^* \to \mathbb{N}$ takes a natural number and a list of linear forms. The given natural number must be an index of the given list. The function Incrvalue returns the sum of maximum values of the program counter i_1 in the linear forms appearing before the given index in the given list. If the given index points to the first element of the list, then the function Incrvalue returns 0.

$$\begin{array}{lcl} \text{Incrvalue}(1, L) & = & 0 \\ \text{Incrvalue}(j, L)_{j>1} & = & \Sigma_{k=1}^{j-1}\text{value}(U_{\text{pc}}(L.k), 1), \end{array}$$

   where the notation $L.k$ denotes the k^{th} element of the list L.

   We give a mathematical definition of the function Incrvalue. For implementation purpose, a recursive definition of the function can be adopted.

   The initialization predicate of a recursion scope operator is the initialization predicate of the linear form of the initial recursion variable, after incrementing the program counters in the predicate by the sum of maximum values of i_1 in linear forms of X_1 until X_{m-1}.

The program counters that are not used in the linear form $\widetilde{p}_m$, are set to an odd value which is equal the highest value of program counter i_1 +1. The highest value of program counter i_1 is used in the linear form of the recursion variable X_i with the highest index i. It is given by the expression, value(Incrpcs$^{>1}(u^n_{\mathrm{pc}}, \mathrm{Incrvalue}(n, [\widetilde{p}_1, \ldots, \widetilde{p}_n])), 1)$.

4. The function Update : $\mathbb{N} \times \widetilde{P}^* \rightarrow \overline{\mathcal{P}}$ takes a natural number and a list of linear process terms. The natural number must point to an element of the list, which consists of linear forms of recursion variables. The function Update(j, L) increments the non zero values of all program counters in the j^{th} element of L, by the increment value Incrvalue(j, L) and replaces any pointers update(X_i) by appropriate action predicates, where $i, j \in [1, \mid L \mid]$. Thus the function Update covers two steps i.e. incrementing process definitions of recursion variables and replacing a pointer update(X_i) by the required action predicates in the linearization procedure.

   A pointer update(X_i) in an alternative of rhs$(L.j)$ is replaced by the following actin predicates:

   (a) An action predicate setting the local environment variables of $L.i$ according to its initialization predicate u_i. The jump set of this action predicate consists of the local discrete and continuous variables of $L.i$;

   (b) An action predicate initializing the program counters according to their initial values in $L.i$, after incrementing the initial values by a factor Incrvalue(i, L) ; and

   (c) In case the program counters of $L.j$ are more than the program counters in $L.i$, then the unused program counters are deactivated.

$$
\begin{array}{l}
\mathrm{Update}(j, L) = \\
\mathrm{Incrpcs}^{>1}(\mathrm{rhs}(L.j), \mathrm{Incrvalue}(j, L)) \\
\quad [\quad \{v \mid v \in \mathrm{dom}(\mathrm{Sigma}(L.i))\} : U(L.i), \\
\qquad \mathrm{pcs}(\mathrm{rhs}(L.i)) : \mathrm{Incrpcs}^{>1}(U_{\mathrm{pc}}(L.i), \mathrm{Incrvalue}(i, L)), \\
\qquad \mathrm{pcs}(\mathrm{rhs}(L.j)) \backslash \mathrm{pcs}(\mathrm{rhs}(L.i)) : \\
\qquad \bigwedge_{i_d \in \mathrm{pcs}(\mathrm{rhs}(L.j)) \backslash \mathrm{pcs}(\mathrm{rhs}(L.i))} \\
\qquad i_d = \mathrm{value}(\mathrm{Incrpcs}^{>1}(U_{\mathrm{pc}}(L.\mathrm{len}(L)), \mathrm{Incrvalue}(\mathrm{len}(L), L)), 1) + 1 \\
\quad / \quad \mathrm{update}(X_i) \\
\quad],
\end{array}
$$

where len(L) denotes the length of the list L. Inactive program counters are set to an odd value equal to the highest value of i_1 in any of the linear forms in the list L +1. The linear form with the highest value of i_1 is the last element of the list. Therefore extra program counters are set to: value(Incrpcs$^{>1}(U_{\mathrm{pc}}(L.\mathrm{len}(L)), \mathrm{Incrvalue}(\mathrm{len}(L), L)), 1) + 1$.

5.8 Initialization

An initialization predicate u is a predicate on the model variables. A process term p with an initialization predicate u, denoted by $u \curvearrowright p$, behaves as p whenever the initialization predicate u holds. The linear form of $u \curvearrowright p$ is calculated by concatenating u with the initialization predicate initializing the model variables in the linear form of p.

Assume

$$\begin{array}{rcl} \text{Normalize}(p_{\text{s}}) = \widetilde{p} & = & [\![_{\text{V}}\, \sigma_{\text{pc}} \cup \sigma, C, L \\ & & :: \ [\![_{\text{R}}\, \{X \mapsto \overline{p}\} :: u_p \wedge u_{\text{pc}} \curvearrowright X \,]\!] \\ & &]\!] \end{array}$$

Then,

$$\begin{array}{rcl} \text{Normalize}(u \curvearrowright p_{\text{s}}) & = & [\![_{\text{V}}\, \sigma_{\text{pc}} \cup \sigma, C, L \\ & & :: \ [\![_{\text{R}}\, \{X \mapsto \overline{p}\} :: u \wedge u_p \wedge u_{\text{pc}} \curvearrowright X \,]\!] \\ & &]\!] \end{array}$$

The linear form of $u \curvearrowright p_{\text{s}}$ has been derived from the following property of the semantics of hybrid χ.

$$u \curvearrowright (u' \curvearrowright p) \leftrightarrows u \wedge u' \curvearrowright (p)$$

5.9 Encapsulation

Two kinds of encapsulation operators are allowed in the input language:

1. $\partial_A(p)$ denotes the encapsulation of non communicating actions in the set A. Actions from the set A are blocked.

2. $\partial_H(p)$ denotes the encapsulation of send and receive actions on channels in the set H. $\partial_H(p)$ is defined as:

   $$\partial_H(p) \triangleq \partial_{\{h\,!\,\text{cs}, h\,?\,\text{cs} | h \in H, \text{cs} \in \Lambda^*\}}(p)$$

   In a channel encapsulation operator, $\partial_H(p)$ the blocking of send and receive actions is done on the basis of communication channels and not on the basis of values sent or received.

Assume

$$\begin{array}{rcl} \text{Normalize}(p_{\text{s}}) = \widetilde{p} & = & [\![_{\text{V}}\, \sigma_{\text{pc}} \cup \sigma, C, L \\ & & :: [\![_{\text{R}}\, \{X \mapsto \overline{p}\} :: u_p \wedge u_{\text{pc}} \curvearrowright X \,]\!] \\ & &]\!] \end{array}$$

Then,

$$\begin{array}{rcl} \text{Normalize}(\partial_L(p_{\text{s}})) & = & [\![_{\text{V}}\, \sigma_{\text{pc}} \cup \sigma, C, L \\ & & :: [\![_{\text{R}}\, \{X \mapsto \text{Encaps}(\overline{p}, L)\} :: u_p \wedge u_{\text{pc}} \curvearrowright X \,]\!] \\ & &]\!] \end{array}$$

where, the function Encaps : $\overline{\mathcal{P}} \times (2^{\mathcal{A}} \cup 2^{\mathcal{H}}) \to \overline{\mathcal{P}}$ takes as the first argument a process term of the form $\overline{p}$, and as the second argument, a set of action labels from the set A_{label} or a set of channels . It scans the given process term for any actions from the given set of action labels or send or receive actions on a channel in the given set of channels. If such an action is present in any alternative of the given process term $\overline{p}$, then that action is replaced by inv true. Any action predicates, pointers or recursive call to X following such an action are removed.

The function Encaps is defined below.

1. $\text{Encaps}(b_{\text{pc}} \to p_{\text{u}}, L) = b_{\text{pc}} \to p_{\text{u}}$
2. In case $\text{label}(p_{\text{atom}}) \in L \vee \text{Ch}(p_{\text{atom}}) \in L$, then

$$\begin{array}{lcl} \text{Encaps}((b_{\text{pc}} \to p_{\text{atom}}, \text{ap}, \text{update}(X_i);\ X), L) & = & b_{\text{pc}} \to \mathsf{inv}\ \text{true} \\ \text{Encaps}((b_{\text{pc}} \to p_{\text{atom}}, \text{ap}, \text{ap}_{\text{pc}}), L) & = & b_{\text{pc}} \to \mathsf{inv}\ \text{true} \\ \text{Encaps}((b_{\text{pc}} \to p_{\text{atom}}, \text{ap}, \text{ap}_{\text{pc}};\ X), L) & = & b_{\text{pc}} \to \mathsf{inv}\ \text{true} \end{array}$$

3. In case $\text{label}(p_{\text{atom}}) \notin L \wedge \text{Ch}(p_{\text{atom}}) \notin L$, then

$$\begin{array}{l} \text{Encaps}((b_{\text{pc}} \to p_{\text{atom}}, \text{ap}, \text{update}(X_i);\ X), L) \\ \qquad = b_{\text{pc}} \to p_{\text{atom}}, \text{ap}, \text{update}(X_i);\ X \\ \text{Encaps}((b_{\text{pc}} \to p_{\text{atom}}, \text{ap}, \text{ap}_{\text{pc}}), L) \\ \qquad = b_{\text{pc}} \to p_{\text{atom}}, \text{ap}, \text{ap}_{\text{pc}} \\ \text{Encaps}((b_{\text{pc}} \to p_{\text{atom}}, \text{ap}, \text{ap}_{\text{pc}};\ X), L) \\ \qquad = b_{\text{pc}} \to p_{\text{atom}}, \text{ap}, \text{ap}_{\text{pc}};\ X \end{array}$$

4. $\text{Encaps}(\overline{p} \,[\!]\, \overline{q}, L) = \text{Encaps}(\overline{p}, L) \,[\!]\, \text{Encaps}(\overline{q}, L)$

where

1. The function $\text{label} : \mathcal{P}_{\text{atom}} \to \text{A}_{\text{label}} \cup A_{\text{com}}$ takes an atomic action and returns its label.

$$\begin{array}{lcl} \text{label}(l_{\text{a}}, W : r) & = & l_{\text{a}} \\ \text{label}(h\,!\,\mathbf{e}_n, W : r) & = & h\,!\,\text{cs} \\ \text{label}(h\,?\,\mathbf{x}_n, W : r) & = & h\,?\,\text{cs} \\ \text{label}(h\,!?\,\mathbf{x}_n := \mathbf{e}_n, W : r) & = & h\,!?\,\text{cs} \end{array}$$

   cs denotes a list of values.

2. The function $\text{Ch} : \mathcal{P}_{\text{atom}} \to \mathcal{H} \cup \{\bot\}$ takes an atomic action. If the given action is a send, receive or communication action, it returns the channel of communication else it returns $\bot$.

$$\begin{array}{lcl} \text{Ch}(l_{\text{a}}, W : r) & = & \bot \\ \text{Ch}(h\,!\,\mathbf{e}_n, W : r) & = & h \\ \text{Ch}(h\,?\,\mathbf{x}_n, W : r) & = & h \\ \text{Ch}(h\,!?\,\mathbf{x}_n := \mathbf{e}_n, W : r) & = & h \end{array}$$

5.10 Channel Scope Operator

Localizing a channel in $[\![_{\mathrm{H}}\, H :: p_{\mathrm{s}}\,]\!]$ has the following effects on the process term p_{s}:

1. It makes communication on a local channel invisible to outside observers. An external observer only observes the silent action τ when communication on a local channel takes place.

2. Send and receive actions on local channels are no longer possible. Only the synchronous execution of a send and receive action resulting in communication is allowed on a local channel.

Assume

$$\begin{array}{rcl} \mathrm{Normalize}(p_{\mathrm{s}}) = \widetilde{p} & = & [\![_{\mathrm{V}}\, \sigma_{\mathrm{pc}} \cup \sigma, C, L \\ & & ::\ \ [\![_{\mathrm{R}}\, \{X \mapsto \overline{p}\} :: u_p \wedge u_{\mathrm{pc}} \curvearrowright X\,]\!] \\ & &]\!] \end{array}$$

Then

$$\begin{array}{rcl} \mathrm{Normalize}([\![_{\mathrm{H}}\, H :: p_{\mathrm{s}}\,]\!]) & = & [\![_{\mathrm{V}}\, \sigma_{\mathrm{pc}} \cup \sigma, C, L \\ & & ::\ \ [\![_{\mathrm{R}}\, \{X \mapsto \mathrm{localCh}(\overline{p}, H)\} :: u_p \wedge u_{\mathrm{pc}} \curvearrowright X\,]\!] \\ & &]\!], \end{array}$$

where the function $\mathrm{localCh} : \overline{\mathcal{P}} \times 2^{\mathcal{H}} \to \overline{\mathcal{P}}$ takes a process term of the form $\overline{p}$ and a set of channels that are to made local to $\overline{p}$. It does the following:

1. It replaces the communication action $h\,!?\,\mathbf{x}_n := \mathbf{e}_n$ with h in the given set by the silent action τ.

2. It replaces the send and receive actions on a cannel in the given set by inv true and removes any action predicates and recursive call to X following such a send or receive action.

It is defined below:

1. $\mathrm{localCh}(b_{\mathrm{pc}} \to p_{\mathrm{u}}, H) = b_{\mathrm{pc}} \to p_{\mathrm{u}}$

2. In case $\mathrm{Ch}(p_{\mathrm{atom}}) \notin H$, then,

$$\begin{array}{l} \mathrm{localCh}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{update}(X_i);\ X), H) \\ \qquad\qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{update}(X_i);\ X \\ \mathrm{localCh}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap}_{\mathrm{pc}}), H) \\ \qquad\qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap}_{\mathrm{pc}} \\ \mathrm{localCh}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap}_{\mathrm{pc}};\ X), H) \\ \qquad\qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap}_{\mathrm{pc}};\ X \end{array}$$

3. In case $\mathrm{Ch}(p_{\mathrm{atom}}) \in H$ and $\mathrm{label}(p_{\mathrm{atom}}) = h\,!?\,\mathrm{cs}$, where $cs \in \Lambda^*$, then

$$\begin{array}{l}\mathrm{localCh}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{update}(X_i);\ X), H) \\ \qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}[\tau/h\,!?\,\mathbf{x}_n := \mathbf{e}_n], \mathrm{ap}, \mathrm{update}(X_i);\ X \\ \mathrm{localCh}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap}_{\mathrm{pc}}), H) \\ \qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}[\tau/h\,!?\,\mathbf{x}_n := \mathbf{e}_n], \mathrm{ap}, \mathrm{ap}_{\mathrm{pc}} \\ \mathrm{localCh}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap}_{\mathrm{pc}};\ X), H) \\ \qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}[\tau/h\,!?\,\mathbf{x}_n := \mathbf{e}_n], \mathrm{ap}, \mathrm{ap}_{\mathrm{pc}};\ X\end{array}$$

4. In case, $\mathrm{Ch}(p_{\mathrm{atom}}) \in H$ and $\mathrm{label}(p_{\mathrm{atom}}) \neq h\,!?\,\mathrm{cs}$, where $cs \in \Lambda^*$, then

$$\begin{array}{lcl}\mathrm{localCh}(b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{update}(X_i);\ X, H) & = & b_{\mathrm{pc}} \to \mathsf{inv}\ \mathrm{true} \\ \mathrm{localCh}(b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap}_{\mathrm{pc}}, H) & = & b_{\mathrm{pc}} \to \mathsf{inv}\ \mathrm{true} \\ \mathrm{localCh}(b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap}_{\mathrm{pc}};\ X, H) & = & b_{\mathrm{pc}} \to \mathsf{inv}\ \mathrm{true}\end{array}$$

5. $\mathrm{localCh}(\overline{p} \,[\!]\, \overline{q}, H) = \mathrm{localCh}(\overline{p}, H) \,[\!]\, \mathrm{localCh}(\overline{q}, H)$

5.11 Urgent Channel Operator

Linearizing the urgent communication operator is allowed only for "well-posed systems". Definitions of well-posed dynamical systems found in literature [20, 21] imply existence and uniqueness of a solution for a given dynamical system. We give a different meaning to well-posed systems. By well-posed systems we mean systems that can be represented by *reactive automata* as defined in [19]. I.e., if a guard condition for a set of actions is true, then the invariants of the target locations (subprocesses following the guard conditions) to the given actions also hold. Thus invariants of target locations do not prevent actions from taking place immediately.

The urgent communication operator makes communication on a given set of channels urgent. In case communication on a channel $h \in H$ is possible for a process p, then the process $\upsilon_H(p)$ cannot do any time transitions.

Assume

$$\begin{array}{lcl}\mathrm{Normalize}(p_{\mathrm{s}}) = \widetilde{p} & = & [\![_{\mathrm{V}}\ \sigma_{\mathrm{pc}} \cup \sigma, C, L \\ & & ::\ \ [\![_{\mathrm{R}}\ \{X \mapsto \overline{p}\} :: u_p \wedge u_{\mathrm{pc}} \curvearrowright X\ ]\!] \\ & &]\!]\end{array}$$

Then

$$\begin{array}{lcl}\mathrm{Normalize}(\upsilon_H(p_{\mathrm{s}})) & = & [\![_{\mathrm{V}}\ \sigma_{\mathrm{pc}} \cup \sigma, C, L \\ & & ::\ \ [\![_{\mathrm{R}}\ \{X \mapsto \mathrm{Urgent}(\overline{p}, H)\} :: u_p \wedge u_{\mathrm{pc}} \curvearrowright X\ ]\!] \\ & &]\!]\end{array}$$

where the function $\mathrm{Urgent} : \overline{\mathcal{P}} \times \mathcal{H} \to \overline{\mathcal{P}}$ takes a process term of the form $\overline{p}$ and a set of channels. It scans the process term $\overline{p}$ searching for alternatives containing communication actions $h\,!?\,cs$ with h in the given channel set and cs a list of

values. If such an alternative is found, then the function Urgent adds another alternative with an urgency condition on the "guard" of the action predicate accompanying the communication action $h\,!?\,cs$. The new alternative is guarded by the same predicate b_{pc} as the found alternative. By "guard" of an action predicate $W : r$, we mean the part of predicate r that imposes conditions on values of model variables before an action takes place.

The function $\mathrm{Precond} : (\mathcal{AP} \cup \mathcal{R}) \to \mathcal{R}$ takes an action predicate or predicate on model variables and returns the part of predicate imposing restrictions on previous values of model variables. For the syntax of allowed predicates see Section 2.

$$
\begin{array}{lcl}
\mathrm{Precond}(\mathrm{true}) & = & \mathrm{true} \\
\mathrm{Precond}(\mathrm{false}) & = & \mathrm{false} \\
\mathrm{Precond}(x^{-}\ \mathrm{op_r}\ c) & = & x\ \mathrm{op_r}\ c \\
\mathrm{Precond}(x^{+}\ \mathrm{op_r}\ c) & = & \mathrm{true} \\
\mathrm{Precond}(r \wedge r') & = & \mathrm{Precond}(r) \wedge \mathrm{Precond}(r') \\
\mathrm{Precond}(W : r) & = & \mathrm{Precond}(r) \\
\mathrm{Precond}(W : r, \mathrm{ap}) & = & \mathrm{Precond}(r) \wedge \mathrm{Precond}(\mathrm{ap})
\end{array}
$$

where W is a subset of model variables, x is a model variable, c is a value and $\mathrm{op_r}$ denotes the set of relational operators.

The function Urgent is defined below:

1. $\mathrm{Urgent}(b_{\mathrm{pc}} \to p_{\mathrm{u}}, H) = b_{\mathrm{pc}} \to p_{\mathrm{u}}$

2. In case $\mathrm{Ch}(p_{\mathrm{atom}}) \notin H$, then

$$
\begin{array}{l}
\mathrm{Urgent}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{update}(X_i);\ X), H) \\
\qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{update}(X_i);\ X \\
\mathrm{Urgent}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap_{pc}}), H) \\
\qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap_{pc}} \\
\mathrm{Urgent}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap_{pc}};\ X), H) \\
\qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap_{pc}};\ X
\end{array}
$$

3. In case $\mathrm{Ch}(p_{\mathrm{atom}}) \in H$ and $\mathrm{label}(p_{\mathrm{atom}}) = h\,!?\,\mathrm{cs}$, where $\mathrm{cs} \in \Lambda^*$, then

$$
\begin{array}{l}
\mathrm{Urgent}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{update}(X_i);\ X), H) \\
\qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{update}(X_i);\ X \\
\qquad\quad [\!]\ b_{\mathrm{pc}} \to \mathsf{urg}\ \mathrm{Precond}(\mathrm{ap}) \\
\mathrm{Urgent}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap_{pc}}), H) \\
\qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap_{pc}} \\
\qquad\quad [\!]\ b_{\mathrm{pc}} \to \mathsf{urg}\ \mathrm{Precond}(\mathrm{ap}) \\
\mathrm{Urgent}((b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap_{pc}};\ X), H) \\
\qquad = b_{\mathrm{pc}} \to p_{\mathrm{atom}}, \mathrm{ap}, \mathrm{ap_{pc}};\ X \\
\qquad\quad [\!]\ b_{\mathrm{pc}} \to \mathsf{urg}\ \mathrm{Precond}(\mathrm{ap})
\end{array}
$$

4. $\mathrm{Urgent}(\overline{p}\ [\!]\ \overline{q}, H) = \mathrm{Urgent}(\overline{p}, H)\ [\!]\ \mathrm{Urgent}(\overline{q}, H)$

5.12 Variable Scope Operator

Assume

$$\begin{array}{rcl}\text{Normalize}(p_{\text{s}}) = \widetilde{p} & = & [\![_{\text{V}}\, \sigma^p_{\text{pc}} \cup \sigma_p, C_p, L_p \\ & & :: \ [\![_{\text{R}}\, \{X \mapsto \overline{p}\} :: u_p \wedge u^p_{\text{pc}} \curvearrowright X \,]\!] \\ & &]\!]\end{array}$$

Then assuming that $\widetilde{p}$ and the variable scope operator $[\![_{\text{V}}\, \sigma_\perp, C, L :: p_{\text{s}} \,]\!]$ do not share any local environment variable names, i.e.

$$(L \cup \text{dom}(\sigma_\perp)) \cap (\text{dom}(\sigma_p) \cup L_p) \quad = \emptyset,$$

we define the linear form of the variable scope operator as follows:

$$\begin{array}{rcl}\text{Normalize}([\![_{\text{V}}\, \sigma_\perp, C, L :: p_{\text{s}} \,]\!]) & = & [\![_{\text{V}}\, \sigma^p_{\text{pc}} \cup \sigma_p \cup \sigma'_\perp, C_p \cup C, L_p \cup L \\ & & :: \ [\![_{\text{R}}\, \{X \mapsto \overline{p}\} :: u \wedge u^p_{\text{pc}} \curvearrowright X \,]\!] \\ & &]\!]\end{array}$$

where,

1. The valuation $\sigma'_\perp$ is defined as follows:

$$\begin{array}{l}\text{dom}(\sigma'_\perp) = \text{dom}(\sigma_\perp) \\ \forall_{x \in \text{dom}(\sigma'_\perp)} \sigma'_\perp(x) = \perp\end{array}$$

2. The initialization predicate u is defined as follows:

$$u = \left(\bigwedge_{x \in \text{dom}(\sigma_\perp) \text{ s.t. } \sigma_\perp(x) \neq \perp} x = \sigma_\perp(x) \right) \wedge u_p$$

   All model variables defined in the variable scope operator are initialized in the initialization predicate u. u_p includes the initialization of those variables that are defined in σ_p and are initialized to $\perp$.

In the output form of our linearization algorithm, we allow a variable scope only at the top most level. Hence during linearization, the local valuations and local variables of all variable scope operators present in a given input process term are moved to the top level. We assume that all variable scope operators of an input process term p_{s} have distinct local variables.

In a variable scope operator, the local variables can be initialized in two ways. They are initialized either in the local valuation or in an initialization predicate appearing at the start of the variable scope operator process term. In the output form of our linearization algorithm, see Section 3, only initialization predicates are allowed to initialize local variables. The top level valuation σ is undefined for all local variables. In the linear form of an input process term p containing a variable scope, the variables local to the variable scope must be set to their initial values at the same time as when they are initialized in p. To

achieve this, we add an action predicate to the action preceding the variable scope in p. This action predicate allows the local variables of the variable scope operator to jump to their required initial values. This has been mentioned in the linearization of a sequential composition (Section 5.4) and recursion scope (Section 5.7.)

A problem may occur when linearizing a recursion variable with a process definition consisting of a variable scope operator and self recursion. This problem was pointed out in [8].

Consider the following recursion definition:

$$X_1 \mapsto [\![_{\mathrm{V}} \{n \mapsto \bot\}, \emptyset, \emptyset :: u \curvearrowright p_{\mathrm{s}}]\!] \; ; \; X_1$$

Without linearization, we can view X_1 as an infinite sequence of variable scopes $[\![_{\mathrm{V}} \{n \mapsto \bot\}, \emptyset, \emptyset :: u \curvearrowright p_{\mathrm{s}}]\!]$ with each member of the sequence having its own instance of the local variable n. In our linearization algorithm, where a variable scope is only allowed at the top level, this is not possible.

We would like to linearize the process definition of X_1 by adding an action predicate $\{n\} : u$ to the last action of p_{s}, where u is the initialization predicate of the variable scope. But situations may arise where appending the last action of p_{s} with $\{n\} : u$ results in a deadlock. For example,

$$X_1 \mapsto [\![_{\mathrm{V}} \{n \mapsto \bot\}, \emptyset, \emptyset :: (n = 0) \curvearrowright \tau, \{n\} : n = n^- + 1]\!] \; ; \; X_1$$

The process definition of X_1 is capable of performing infinite number of τ actions. When we linearize X_1, then an action predicate $\{n\} : (n = 0)$ is appended to the last action of the variable scope. In this case, the last action of the variable scope, also updates n in its predicate. This leads to the predicate $n = n^- + 1 \wedge n = 0$ which equals false and therefore in the linear form of X_1, the action τ cannot take place. For this linearization algorithm, we simply disallow recursion variables with process definitions consisting of variable scopes and self recursion. Further research on this topic is left as a future work.

6 Conclusive Remarks

In this report a linearization of hybrid χ specifications using program counters is presented. Linearization is the procedure of rewriting a process specification into a linear form. A linear form consists of only basic operators of a process language such as atomic actions,sequential composition and choice. Program counters are fresh discrete variables introduced in a process specification that keep track of next possible actions of the system. In the linear form of a parallel composition, a separate program counter is introduced for each parallel component. Action interleaving of components is modelled by updating of program counters which greatly reduces the size of the resulting linear form.

A linearization algorithm and tool for the previous version of hybrid χ [7] is already available in [8]. In [8] the linearization algorithm does not use any special data structures to model interleaving in parallel composition or sequential

composition of processes. The linear form of a hybrid χ process P_{N} as defined in [8] appears to be similar to the linear form obtained in this report. The linear form obtained by the algorithm in [8] is given below:

$$
\begin{aligned}
P_{\mathrm{N}} = \; & [\!|_{\mathrm{V}}\, \sigma, C, L \\
& :: \; [\!|_{\mathrm{H}}\, H \\
& \qquad [\!|_{\mathrm{R}}\, R_{\mathrm{n}} :: X \,]\!| \\
& \quad]\!| \\
&]\!|
\end{aligned}
$$

where R_{n} is a recursive definition and $X \in \mathrm{dom}(R_{\mathrm{n}})$.

Main differences in the linear forms of the two linearization algorithms are as follows: In [8] the channel scope operator is not eliminated from the normal form and the recursion definition R_{n} may contain more than one recursion variables. The channel scope operator can easily be removed from the linear form as already mentioned in [8]. The righthand sides of all recursion variables defined in R_{n} are linear process terms, i.e. they consist of only basic operators (such as sequential and alternative composition) and tail recursion. The algorithm given in [8] is for the previous syntax and semantics, therefore the recursive process definitions also contain unary operators (not present in the new syntax) like signal emissions and jump-enabling operators. It is not possible to eliminate signal emission from these process terms.

We linearized the train gate controller specification (taken from [12]) modelled in hybrid χ using the linearization tool of [8]. The linear term obtained was very long with over a hundred and seventy recursion variables defined in the recursion definition. Many of these recursion variables were not reachable from the initial term. Based on the new linearization algorithm given in this report, a tool in ASF+SDF environment [22] is being developed. The previous linearization algorithm for hybrid χ [8] was also developed in the same language. ASF+SDF is a term rewriting language that is suitable for defining domain-specific languages, source code analysis, source code transformations, semantics of programming languages, many-sorted algebras etc. It is supported by the Meta-Environment [23] that provides an Integrated development environment for application development in ASF+SDF.

Some restrictions have been imposed on the input process terms (more specifically on the recursion scope operator) to the new linearization algorithm. These restrictions are the same as were in [8]. They are as follows: only complete recursion definitions, i.e. recursion definitions that do not refer to recursion variables defined outside the recursion scope are allowed; an occurrence of a recursion variable must be guarded; only tail recursion is permitted and a recursion variable cannot occur in a parallel composition. Completeness of recursion definitions makes the task of linearization easier and it does not pose any limitations on the expressiveness of specifications. The restriction of guardedness is required for uniqueness of solutions for recursion variables. Recursion over parallel composition is disallowed. This restriction prevents a possible infinite parallelism, e.g. in $X_1 \mapsto a;\, X_1 \parallel b;\, Y_2$. But parallel composition of recursion scopes is allowed which can model parallelism among different components of a system.

As mentioned in Section 5.12, recursion variables with process definitions consisting of variable scopes and self recursion are disallowed. The problem arising in such self calling recursion variables could perhaps be avoided by introducing a new identical recursion variable in the specification. However, in retrospect, the usefulness of setting a local variable to some value in the last action of a variable scope after which the variable ceases to exist is questionable. Corresponding to the variable scope operator in hybrid χ is the abstraction operator in HyPA [13]. In the linearization of HyPA process terms [18], an abstraction operator is only allowed at the top most level in the input to the linearization algorithm. The reason being the same as that for restricting variable scope operator in recursion scopes that an abstraction operator cannot be eliminated from recursive equations.

In comparison to the linearization algorithms for HyPA and μCRL, our linearization algorithm is much simpler. No complex data structures (such as stacks in HyPA and lists, multi-sets and stacks in μCRL) are used during linearization. This makes the intermediate and final linear forms of our linearization algorithm more readable than the linear forms obtained from other linearization algorithms. On the other hand, the use of these data structures would allow more flexibility in input process terms. Stacks are needed in the linearization of a sequential composition of parameterized recursion variables. In μCRL [14], recursive occurrences of parallel composition and of renaming operators are linearized using lists of multi-sets. In our linearization algorithm, the reuse of program counters turns out to be natural in linearizing a sequential composition and recursion scope operator. For alternative composition with new root (Section 5.6.2), the reuse of program counters gives a rather complex algorithm. The algorithm for alternative composition with a new root is only needed in cases where one of the alternative is a recursion scope with self recursion. For modeling of most dynamical systems, the alternative composition without a new root will be used which is simpler.

References

[1] Van der Mortel-Fronczak, J.M. and J.E. Rooda, "Application of Concurrent Programming to Specification of Industrial Systems", Proceedings of the 1995 IFAC Symposium on Information Control Problems in Manufacturing, pp. 421-426.

[2] Van der Mortel-Fronczak, J.M., J.E. Rooda and N.J.M. van den Nieuwelaar, " Specification of a Flexible Manufacturing System Using Concurrent Programming", The International Journal of Concurrent Engineering: Research and Applications, Vol. 3, No. 3, 187-194, 1995.

[3] Fabian, G., "A Language and Simulator for Hybrid Systems", IPA dissertation series 1999-11, Faculty of Mechanical Engineering, TU/e, 1999.

[4] D.A. van Beek and J.E. Rooda, "Languages and applications in hybrid modelling and simulation: Positioning of χ", Control Engineering Practice, vol 8(1), 81-91, 2000.

[5] D.A. van Beek and K.L. Man and M.A. Reniers and J.E. Rooda and R.R.H. Schiffelers, " Syntax and Consistent Equation Semantics of Hybrid Chi", Journal of Logic and Algebraic Programming, 2006, vol 68, 1-2,129-210.

[6] D.A. van Beek M.A. Reniers R.R.H. Schiffelers J.E. Rooda, "Syntax and Operational Semantics of Chi", (To be published soon).

[7] K. L. Man, R.R.H. Schiffelers , "Formal specification and analysis of hybrid systems", IPA dissertation series 2006-04, Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e, 2006.

[8] R.J.M Theunissen, "Process algebraic linearization of hybrid Chi", Master's thesis, Department of Mechanical Engineering, SE 420479, Systems Engineering Group, TU/e, June, 2006.

[9] J.C.M. Baeten, W.P. Weijland, "Process Algebra", Cambridge Tracts n Theoretical Computer Science, Cambridge University Press.

[10] J.C.M. Baeten, T. Basten, M.A. Reniers, "Algebra of Communiating Processes", Faculty of Mathematics and Computer Science, TU/e.

[11] J.C.M. Baeten, C.A.Middelburg, "Process Algebra with Timing", Springer, 2002.

[12] J.A.Bergstra, C.A.Middelburg, "Process Algebra for Hybrid Systems", Theoretical Computer Science 335, (2005) 215-280.

[13] P. Cuijpers, M. Reniers, " Hybrid process algebra", Journal of Logic and Algebraic Programming, 62(2):191-245, Februari 2005.

[14] Y. Usenko, "Linearization in μCRL", PhD thesis TU/e, IPA dissertation series 2002-16, 2002

[15] Arno Wouters ,"Manual for the μcrl Tool Set (version 2.8.2)",Report Sen-R0130 December 31,2001, CWI, Amsterdam.

[16] R. Alur. Marrying words and trees, 26th ACM Symposium on Principles of Database Systems, 2007.

[17] Andrzej Wasowski, "Flattening statecharts without explosion ", Language, Compiler and Tool Support for Embedded Systems, Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, 2004.

[18] P.C.W. van den Brand, M.A. Reniers and P.J.L. Cuijpers, Linearization of Hybrid Processes, Journal of Logic and Algebraic Programming, 68(1-2):54-104, June-July 2006. Special issue on Process Theory for Hybrid Systems.

[19] D.A. van Beek, M.A. Reniers, R.R.H. Schiffelers, J.E. Rooda, "Foundations of a compositional interchange format for hybrid systems", 19, SE Report 2006-05, Internal Report (2006)

[20] J.-I. Imura and A.J. van der Schaft, "Characterization of well-posedness of piecewise linear systems", IEEE Transactions on Automatic Control, 45(9):1600-1619, 2000.

[21] A.Y. Pogromsky, W.P.M.H. Heemels, H. Nijmeijer, "On solution concepts and well-posedness of linear relay systems", Automatica, 39(12), 2139 - 2147, (2003)

[22] Mark van den Brand, Paul Klint, Jurgen Vinju, "The ASF+SDF Meta-environment: A Component-Based Language Development Environment ", in Compiler Construction : 10th International Conference, CC 2001: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings , LNCS, Volume 2027/2001, 2001.

[23] The Meta Environment,

http://www.cwi.nl/htbin/sen1/twiki/bin/view/Meta-Environment