Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 200 (2008) 51–70

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Separate Compilation of Polychronous Specifications

Julien Ouy Jean-Pierre Talpin Loïc Besnard Paul Le Guernic

INRIA - IRISA, Campus de Beaulieu, 35042 Rennes, France

Abstract

As code generation for synchronous programs requires strong safety properties to be satisfied, compositionality becomes a difficult goal to achieve. Most synchronous languages, such as Esterel, Lustre or Signal require a given module or compilation unit to be insensitive to latency that communication with its environment may incur. In Lustre or Signal, for instance, a compilation unit must satisfy the so-called property of endochrony. To preserve endochrony in an asynchronous environment, an ad-hoc protocol is synthesized to interface the module. However, endochrony is not preserved by composition. Consequently, the protocol has to be rebuilt every time a new module is added in the environment. We propose a methodology and code generation scheme which simplifies this concern. It consists of weakening the global objective of globally preserving endochrony. Instead, we aim at the preservation of a more liberal and compositional objective, weak endochrony [14], which is compositional and much closer from the expected requirement of insensitivity to communication latency. As a result, our code generation scheme supports true separate compilation: a locally compiled synchronous module does not require its synthesized interface with the environment to be rebuilt once composed with another module.

Keywords: Synchronous programming, mutil-clocked systems, code generation, separate compilation.

1 Introduction

To facilitate embedded system design, synchronous programming languages offer analysis, transformation and code generation services that guarantee the correct execution of a program by construction. To achieve this goal, most synchronous languages, such as Esterel, Lustre or Signal, require a given compilation unit or module to be insensitive to latency that may possibly be incurred when communicating with the environment.

This requires a safety property to be locally satisfied: endochrony. Endochrony is the property of a module that is able to deterministically define the pace of its output from that of its inputs. Endochrony is usually enforced by the synthesis of appropriate protocols that interface the specified module for its correct execution in an asynchronous environment.

Unfortunately, endochrony is not preserved by composition: the composition of two endochronous modules may not be endochronous. Should an endochronous

doi:10.1016/j.entcs.2008.02.006

module be first compiled and later be composed to another, then its interface may possibly need to be recompiled to meet the requirements of its new environment.

To circumvent this, synchronous programming language usually perform distributed code generation once the system is entirely specified and globally adheres to the property of isochrony (the equivalence of the synchronous and asynchronous composition of the modules).

To gain from the flexibility and ergonomy that separate compilation services would offer, we propose a simple analysis and code generation technique that addresses this limitation. Our approach consists in weakening the global safety objective usually targeted in related frameworks: isochrony. Instead, we consider a more liberal one of weak isochrony recently proposed in [14].

Starting from this objective, we design a simple code generation scheme, that merely reuses most of the existing code generation suite of Polychrony [13] and has the advantage of being compositional: a locally compiled synchronous module does not need its interface with the environment to be rebuilt once composed to another synchronous module.

Previous work

To our knowledge, work that relates to the issue addressed in the present article is essentially concerned with the distribution of synchronous data-flow programs. The issue of compositionally generating separately compiled code has not, to our knowledge, been addressed in isolation in the related work, although it implies results on distributed code generation.

Regarding the Signal language, related work has been introduced by Maffeis [4] and Aubry [1]. The approach proposed in Signal is to partition a synchronous module and synthesise sufficient inter-partition communications to ensure the preservation of the property that was initially secured for the synchronous module: endochrony.

Endochrony guarantees that the system responds to events incoming from an asynchronous environment by locally and deterministically choosing which of them needs to be synchronized at all times. Endochrony ensures the insensitivity of local computations and communications to global network latency. But, it is unfortunately not compositional.

A different approach is proposed by Girault [3] in the context of the Lustre and Esterel languages. It consists of the replication of the automaton obtained from the synchronous module and then on the optimized elimination of replicated transitions, replaced by inter-partition communications. Of course, distributed code generation is based on the objective of globally preserving the formal property secured locally: endochrony.

In [14], the so-called property of weak endochrony is proposed. Weak endochrony is a local property of synchronous programs that supports the compositional construction of globally asynchronous system that are insensitive to latency by adhering to the global objective of weak-isochrony. A weakly endochronous program is a deterministic synchronous program in which independent reaction support the diamond property. A weakly isochronous system is composed of non-blocking weakly endochronous programs: a synchronous transition initiated locally results in a globally asynchronous execution.

In [16], we proposed an analysis of Signal programs to check the property of weak endochrony in order to compositionally check the insensitivity of a synchronous module to latency. However, we observe that checking weak endochrony is far more costly to be usable for code generation purposes, as it requires the exploration of the state-space of the synchronous module being analyzed.

Our approach consists in maintaining a less costly yet compositional objective of weak-isochrony while considering the composition of endochronous modules. This appears to be a much more cost-effcient approach to code generation. Our contribution consists in an implementation of this methodology that efficiently reuses most of Polychronys compilation tool-chain to propose a simple synthesis scheme ensuring the aimed compilation objectives.

Outline

The article presents existing and contributed compilation techniques in the manner of a tutorial and through a series of illustrative examples. It does not address or prove the related formal aspects. Instead, it merely relies on existing analysis algorithms (and proofs) implemented in Polychrony.

The article starts, Section 2, as a tutorial on the Signal data-flow specification language and on its analysis of synchronization and scheduling relations. Section 3 continues this tutorial with a presentation of the code generation techniques currently implemented in Polychrony, the toolset supporting the Signal language. Our contribution, built upon these techniques, is presented in Section 4.

2 Position of the problem

To position the problem, we start with an informal yet thorough tutorial on the syntax, analysis and code generation techniques for Signal implemented in the Polychrony toolset.

Introduction to Signal

A Signal process, noted p, consists of the synchronous composition, noted $p \| q$, of equations on signals, noted $x = y\ f\ z$. A signal x is an infinite flow of values that is discretely sampled according to the pace of a symbolic clock, noted $\hat{x}$. Therefore, an equation partially relates in an abstract timing model, represented by clock relations, and a process, that defines the simultaneous solution of a system of equations in that timing domain.

$$p ::= x = y\ f\ z \mid p \| q \mid p/x$$

The process p/x restricts the lexical scope of the signal x to the process p. Signal defines the following primitive equations:

- A functional equation $x = y \, f \, z$ defines an arithmetic or boolean relation f between its operands y, z and the result x.
- A delay equation $x = y \, \mathsf{pre} \, v$ initially defines the signal x by the value v and then by the value of the signal y from the previous execution of the equation. In a delay equation, the signals x and y are assumed to be synchronous, i.e. either simultaneously present or simultaneously absent at all times.
- A sampling $x = y \, \mathsf{when} \, z$ defines x by y when z is true and both y and z are present. In a sampling equation, the output signal x is present iff both input signals y and z are present and z holds the value *true*.
- A merge $x = y \, \mathsf{default} \, z$ defines x by y when y is present and by z otherwise. In a merge equation, the output signal is present iff either of the input signals y or z is present.

A formal semantics of Signal in the polychronous model of computation is provided in [11] and quoted in appendix.

Clock and scheduling relations

The data-flow synchronous formalism Signal supports a representation of the control-flow and data-flow graphs of multi-clocked specifications for the purpose of analysis and transformation. In this structure, a clock c denotes a set of instants and defines a discrete sample of time. It is used as the condition upon which (or the time at which) a data-flow relation is executed.

The clock $\hat{x}$ of a signal x denotes the instants at which the signal x is present. The clocks $[x]$ (resp. $[\neg x]$) denotes the instants at which the signal x is present and holds the value true (resp. false).

$$c ::= \hat{x} \,|\, [x] \,|\, [\neg x]$$

A clock expression e is either the empty clock, noted 0 to mean the empty set of instants, a signal clock c, or the conjunction $e_1 \wedge e_2$, the disjunction $e_1 \vee e_2$, the complement $e_1 \setminus e_2$ of two clock expressions e_1 and e_2.

$$e ::= 0 \,|\, c \,|\, e_1 \wedge e_2 \,|\, e_1 \vee e_2 \,|\, e_1 \setminus e_2$$

Signals and clocks are nodes, noted a or b, in a labeled directed graph. Such a graph, noted g or h, describes synchronization and scheduling relations between nodes.

$$a, b ::= x \,|\, \hat{x} \quad \text{(node)}$$

A clock relation $c = e$ specifies that the clock c is present iff the clock expression e is true. A scheduling relation $a \to^c b$ specifies that the calculation of the node b, a signal or a clock, cannot be scheduled before that of a when the clock c is present. Since a graph g is the abstraction of a Signal process p, it is subject to the same composition $g \,|\, h$ and scoping g/x rules as processes.

$$g, h ::= c = e \,|\, a \to^c b \,|\, (g \,|\, h) \,|\, g/x$$

Any Signal process p corresponds to a system of implicit clock and scheduling relations g that denotes its timing and scheduling structure. It forms the interface of that process in the code generator process. All the analysis and transformation of the process p are based on g. We write $p : g$ to mean that p has graph g. The inference system $p : g$ is defined by structural induction on p. In a delay equation $x = y \,\mathsf{pre}\, v$, the input and output signals are synchronous, written $\hat{x} = \hat{y}$, and do not have any scheduling relation.

$$x = y \,\mathsf{pre}\, v : (\hat{x} = \hat{y})$$

In a sampling equation $x = y \,\mathsf{when}\, z$, the clock of the output signal x is defined by that of the input signal $\hat{y}$ at the sampling condition $[z]$. The input y is scheduled before the output when both $\hat{y}$ and $[z]$ hold, written $y \to^{\hat{x}} x$.

$$x = y \,\mathsf{when}\, z : (\hat{x} = \hat{y} \wedge [z] \,\|\, y \to^{\hat{x}} x)$$

In a merge equation $x = y \,\mathsf{default}\, z$ the output signal x is present if either of the input signals y, z are. The input signal y is scheduled before x when it is present, written $y \to^{\hat{y}} x$, and otherwise z is, written $z \to^{\hat{z}\setminus\hat{y}} x$.

$$x = y \,\mathsf{default}\, z : (\hat{x} = \hat{y} \vee \hat{z} \,\|\, y \to^{\hat{y}} x \,\|\, z \to^{\hat{z}\setminus\hat{y}} x)$$

A functional equation $x = y\, f\, z$ synchronizes and serializes its inputs and output signal.

$$x = y\, f\, z : (\hat{y} = \hat{z} \,\|\, \hat{x} = \hat{y} \,\|\, y \to^{\hat{x}} x \,\|\, z \to^{\hat{x}} x)$$

The rule for composition $p \| q$ is defined by induction on the deductions $p : g$ and $q : h$ made on the sub-terms of the expressions, it reads: "if $p : g$ and $q : h$ then $p \| q : g \| h$". Same with the rule for restriction p/x.

$$\frac{p : g \quad q : h}{p \| q : g \| h} \qquad \frac{p : g}{p/x : g/x}$$

In the remainder, we write $g \models h$ to mean that the clock and scheduling relations of g (a model) satisfies the clock and scheduling relations h (a property). For any process p of graph g and any boolean signal x in p, the assertions $g \models \hat{x} = [x] \vee [\neg x]$ and $g \models [x] \wedge [\neg x] = 0$ always hold.

3 Current practice illustrated

To illustrate the sequential code generation scheme implemented in Polychrony, we consider the specification and analysis of a one-place buffer. Process buffer implements two functionalities: $\mathsf{alternate}$ and $\mathsf{current}$.

$$x = \mathsf{buffer}(y) \stackrel{\text{def}}{=} (x = \mathsf{current}(y) \,\|\, \mathsf{alternate}(x, y))$$

Process $\mathsf{alternate}$ synchronizes the signals x and y to the true and false values of an alternating boolean signal t.

$$\mathsf{alternate}(x, y) \stackrel{\text{def}}{=} (s = t \,\mathsf{pre\ true} \,\|\, t = \,\mathsf{not}\, s \,\|\, \hat{x} = [t] \,\|\, \hat{y} = [\neg t]) \,/st$$

Process $\mathsf{current}$ stores the values of an input signal y and sends them along the output signal x upon request.

$$x = \mathsf{current}(y) \stackrel{\text{def}}{=} (r = y\,\mathsf{default}\,(r\,\mathsf{pre}\ \mathsf{false}\,)\,|\,x = r\,\mathsf{when}\,\hat{x}\,|\,\hat{r} = \hat{x} \vee \hat{y})\,/r$$

Synchronization analysis

The inference system $p : g$ infers the clock relations that denote the synchronization constraints implied by process buffer. There are four of them:

$$\hat{r} = \hat{s} \qquad \hat{t} = \hat{x} \vee \hat{y} \qquad \hat{x} = [t] \qquad \hat{y} = [\neg t]$$

From these equations, we observe that process buffer has three clock equivalence classes. The clocks $\hat{s}, \hat{t}, \hat{r}$ are synchronous and define the master clock synchronization class of buffer. Two other synchronization classes, $\hat{x}$ and $\hat{y}$, correspond to the true and false values of the boolean signal t.

$$\hat{r} = \hat{s} = \hat{t} \qquad \hat{x} = [t] \qquad \hat{y} = [\neg t]$$

Hierarchization

Hierarchization is the first source-to-source transformation step performed by the Signal compiler. It consists of syntactically restructuring a program in a way that reflects the ordering of its clock equivalence classes. This hierarchy or ordering defines a control-flow tree structure that will later become that of the transition function in the generated code.

The structure of a hierarchy is denoted by a partial order relation $\preceq$. It is defined by inductive application of the following rules :

(1) for all boolean signals x of g, define $\hat{x} \preceq [x]$ and $\hat{x} \preceq [\neg x]$. This means that, if we know that x is present, then we can determine whether x is true or false.

(2) if $b = c$ is deductible from g then define $b \preceq c$ and $c \preceq b$, written $b \sim c$. This means that if b and c are synchronous, and if either of the clocks b or c is known to be present, then the presence of the other can be determined.

(3) if $g \models b_1 = c_1\, f\, c_2$, $f \in \{\wedge, \vee, \setminus\}$, $b_2 \preceq c_1$, $b_2 \preceq c_2$ and b_2 is minimal [1] then $b_2 \preceq b_1$. This means that if b_1 is defined by $c_1\, f\, c_2$ in g and if both clocks c_1 and c_2 can be determined once their common upper bound b_2 is known, then b_1 can also be determined when b_2 is known.

For example, the hierarchy of the buffer is build in three steps by application of rules 1 to 3.

(i) From rule 1, one infers that $\hat{t}$ is above two branches $[t]$ and $[\neg t]$ as $[t] \succeq \hat{t} \preceq [\neg t]$.

$$\begin{tikzcd} & \hat{t} \arrow[dl, dash] \arrow[dr, dash] & \\ {[t]} & & {[\neg t]} \end{tikzcd}$$

[1] i.e. for any b such that $b \preceq c_1$ and $b \preceq c_2$, $b_2 \preceq b$

(ii) From rule 2, three equivalence classes $\hat{r} \sim \hat{s} \sim \hat{t}$, $\hat{x} \sim [t]$ and $\hat{y} \sim [\neg t]$ are constructed.

$$\begin{tikzcd}
& \hat{r} \sim \hat{s} \sim \hat{t} & \\
{[t]} \sim \hat{x} & & {[\neg t]} \sim \hat{y}
\end{tikzcd}$$

(iii) From rule 3, $\hat{t}$ is placed just above the least upper-bound of $\hat{x}$ and $\hat{y}$, that is to say $\hat{s}$.

$$\begin{tikzcd}
& \hat{r} \sim \hat{s} \sim \hat{t} \arrow[dl, no head] \arrow[dr, no head] & \\
{[t]} \sim \hat{x} & & {[\neg t]} \sim \hat{y}
\end{tikzcd}$$

Next, one has to define a proper scheduling of all computations to be performed within each clock equivalence class (e.g. to schedule s before t) and across them (e.g. to schedule x or y before r). This task is devoted to scheduling analysis, presented next.

Disjunctive form

Before that, Polychrony attempts to eliminate all clocks that are expressed using symmetric difference from the graph g of a process. This transformation consists in rewriting clock expressions of the form $e_1 \setminus e_2$ present in the synchronization and scheduling relations of g in a way that does no longer denote the absence of an event e_2, but that is instead computable from the presence or the value of signals.

In the case of process current, for instance, consider the alternative input r pre false in the first equation:

$$r = y \,\mathsf{default}\, (r \,\mathsf{pre\ false})$$

Its clock is $\hat{r} \setminus \hat{y}$, meaning that the previous value of r is assigned to r only if y is absent. To determine that y is absent, one needs to relate this absence to the presence or the value of another signal.

In the present case, there is an explicit clock relation in the alternate process: $\hat{y} = [\neg t]$. It says that y is absent iff t is present and true. Therefore, one can test the value of t instead of the presence or absence of y in order to deterministically assign either y or r pre false to r

$$y \rightarrow^{[\neg t]} r \;{}^{[t]}\!\leftarrow r \,\mathsf{pre\ false}$$

In [2], it is shown that the symmetric difference $c \setminus d$ between two clocks c and d has a disjunctive form only if c and d have a common minimum b in the hierarchy $\preceq$ of the process, i.e.,

$$c \succeq b \preceq d$$

We say that a graph g is in disjunctive form iff it has no clock expression defined by symmetric difference. The implicit reference to absence incurred by symmetric difference can be defined as $c \setminus d =^{\text{def}} c \wedge \overline{d}$ and can be isolated using logical decomposition rules :

- conjunction $\overline{c \wedge d} \overset{\text{def}}{=} \overline{c} \vee \overline{d}$ and disjunction $\overline{c \vee d} \overset{\text{def}}{=} \overline{c} \wedge \overline{d}$.

- positive $\overline{[x]} \overset{\text{def}}{=} \overline{\hat{x}} \vee [\neg x]$ and negative $\overline{[\neg x]} \overset{\text{def}}{=} \overline{\hat{x}} \vee [x]$ signal occurrences.

The reference to the absence of a signal x, noted $\overline{\hat{x}}$, is eliminated if (and only if) one of the possible elimination rules applies:

- The zero rule: $\hat{x} \wedge \overline{\hat{x}} \overset{\text{def}}{=} 0$, because a signal is either present or absent, exclusively.
- The "one" rule: $c \wedge (\hat{x} \vee \overline{\hat{x}}) \overset{\text{def}}{=} c$, because the presence or the absence of a signal is subsumed by any clock c.
- The synchrony rule: if $d \sim \hat{x}$ then $\overline{\hat{x}} \overset{\text{def}}{=} \overline{d}$, to mean that if $\overline{\hat{x}}$ cannot be eliminated but $\hat{x}$ is synchronous to the clock d, then $\overline{d}$ can possibly be eliminated possibly instead.

In the case of process current in the example of the buffer one has that

$$\hat{y} \sim [\neg t] \qquad \hat{x} \sim [t] \qquad \hat{r} \sim \hat{t}$$

Hence $\hat{x} \succeq \hat{t} \preceq \hat{y}$ and therefore $\hat{r} \setminus \hat{y}$ can be interpreted as $[t]$.

Scheduling analysis

Given the skeleton control-flow tree produced using the hierarchization algorithm and clock equations in disjunctive form, the compilation of a Signal program reduces to finding a proper way to schedule computations within and across clock equivalence classes. The inference system of the previous section defines the precise scheduling between the input and output signals of process buffer. Notice that t is needed to compute the clocks $\hat{x}$ and $\hat{y}$.

$$s \rightarrow^{\hat{s}} t \qquad y \rightarrow^{\hat{y}} r \qquad r \rightarrow^{\hat{x}} x$$

As seen in the previous section, however, the calculation of clocks in disjunctive form induces additional scheduling constraints, and, therefore, one has to take them into account at this stage. This is done by refining the graph g with a reinforced one, h, satisfying $h \models g$, and by ordered application of the following rules:

(i) $h \models \hat{x} \rightarrow^{\hat{x}} x$ for all $x \in \text{vars}(p)$. This means that the calculation of x cannot take place before its clock $\hat{x}$ is known.

(ii) if $g \models \hat{x} = [y]$ or $g \models \hat{x} = [\neg y]$ then $h \models y \rightarrow^{\hat{y}} \hat{x}$. This means that, if the clock of x is defined by a sample of y, then it cannot be computed before the value of y is known.

(iii) if $g \models \hat{x} = \hat{y}\, f\, \hat{z}$ with $f \in \{\vee, \wedge\}$ then $h \models \hat{y} \rightarrow^{\hat{y}} \hat{x} \,|\, \hat{z} \rightarrow^{\hat{z}} \hat{x}$. This means that, if the clock of x is defined by an operation on two clocks y and z, then it cannot be computed before these two clocks are known.

Reinforcing the graph of the buffer yields a refinement of its inferred graph with a structure implied by the calculation of clocks (we just ommitted clocks on arrows to lighten the depiction). Notice that t is now scheduled before the clocks $\hat{x}$ and $\hat{y}$.

$$\begin{tikzcd}
\hat{t} \arrow[r] & t \arrow[r] \arrow[dr] & \hat{x} \arrow[r] & x & r \arrow[l] & \hat{r} \arrow[l] \\
\hat{s} \arrow[r] & s \arrow[u] & \hat{y} \arrow[r] & y \arrow[ur] & &
\end{tikzcd}$$

Code can be generated starting from this refined structure only if the graph is acyclic. To check whether it is or not, we compute its transitive closure:

(i) if $g \models a \to^c b$ then $g \models a \twoheadrightarrow^c b$. This just tells that the construction of the transitive closure relation $\twoheadrightarrow$ starts from the scheduling graph $\to$ of the process.

(ii) if $g \models a \twoheadrightarrow^c b$ and $g \models a \twoheadrightarrow^d b$ then $g \models a \twoheadrightarrow^{c \vee d} b$. If b is scheduled after a at clock c and at clock d then so it is at clock $c \vee d$

(iii) if $g \models a \twoheadrightarrow^c b$ and $g \models b \twoheadrightarrow^d z$ then $g \models a \twoheadrightarrow^{c \wedge d} z$. If b is scheduled after a at clock c and z after b at clock d then z is necessarily scheduled after a at clock $c \wedge d$

The complete graph g of a process p is *acyclic* iff $g \models a \twoheadrightarrow^e a$ implies $g \models e = 0$ for all nodes a of g. The graph of our example is. Using this structure, together with the control-flow graph skeleton denoted by the hierarchy $\preceq$, Polychrony can generate sequential or distributed code.

Sequential code generation

To sequentially schedule the graph g, we need to further refine it in order to remove internal concurrency without affecting the composability of the graph with its environment. This is done by observing the following rule: a graph h reinforces g iff, for any graph f, if $f \| g$ is acyclic then $f \| g \| h$ is acyclic.

Starting from a sequential schedule and a hierarchy of process buffer, Polychrony generates simulation code split in several files.

```
int main() {
    bool code;
    buffer_OpenIO();
    code = buffer_initialize();
    while (code) code = buffer_iterate();
    buffer_CloseIO();
}
```

The main C file consists of opening the input-output streams of the program, of initializing the value of delayed signals and iteratively executing a transition function until no values are present along the input streams (return code 0). Simulation is finalized by closing the IO streams.

The most interesting part is the transition function. It translates the structure of the hierarchy and of the serialized scheduling graph in C code. It also makes a few optimizations along the way. For instance, r has disappeared from the generated code. Since the value stored in y from one iteration to another is the same as that of r, it is used in place of it for that purpose.

In the C code, the three clock equivalence classes of the hierarchy correspond to three blocks: line 2 (class $\hat{s} \sim \hat{t}$), lines $3-5$ (class $[t] \sim \hat{y}$) and lines $6-9$ (class $[\neg t] \sim \hat{x}$). The sequence of instructions between these blocks follows the sequence $t \to y \to x$ of the scheduling graph. Line 10 is the finalization of the transition function. It stores the value that s will hold next time.

```
01. bool buffer_iterate () {
02.     t = !s;
03.     if t {
04.         if !r_buffer_y (&y) return FALSE;
05.     }
06.     if !t {
07.         x = y;
08.         w_buffer_x (x);
09.     }
10.     s = t;
11.     return TRUE;
12. }
```

Also notice that the return code is true, line 11, when the transition function finalizes, but false if it fails to get the signal y from its input stream, line 4. This is fine for simulation code, as we expect the simulation to end when the input stream sample reaches the end. Embedded code does, of course, operate differently. It either waits for y or suspends execution of the transition function until it arrives.

A definition of endochrony

The buffer process satisfies the property of endochrony. Literally, this means that the buffer is locally timed. In the transition function of the buffer, this is easy to notice by observing that, at all times, the function synchronizes on either receiving y from its environment or sending x to its environment. Hence, the activity of the transition function is locally paced by the instants at which the signals x and y are present.

However, remember that the structure of control in the transition function is constructed using the hierarchy of process buffer. In the case of an internally timed process, this structure has the particular shape of a tree.

```
if t {
    if !r_buffer_y (&y) return FALSE;
} else {
    x = y;
    w_buffer_x (x);
}
```

At any time, one can always start reading the state s of the buffer, and calculate t. Then, if t is true, one emits x and, otherwise, one receives y. The presence of any signal in process buffer is determined from the value of a signal higher in the hierarchy or, at last, from its root.

$$\begin{tikzcd}
 & \hat{r} \sim \hat{s} \sim \hat{t} \arrow[dl, dash] \arrow[dr, dash] & \\
{[t] \sim \hat{x}} & & {[\neg t] \sim \hat{y}}
\end{tikzcd}$$

Formally, whatever the exact time samples t_1 and t_2 at which it receives an input signal y, or the time samples u_1 and u_2 at which it sends an output signal x, the buffer always behaves according to the same timing relations: t_i occurs strictly before u_i and s is always used at t_i and u_i.

$$\begin{array}{ccccccccc}
 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\,\cdot & \cdot\,\cdot & \cdot\,\cdot \\
y & t_1 & & t_2 & & t'_1 & & t'_2 & \\
s & t_1 & u_1 & t_2 & u_2 & t'_1 & u'_1 & t'_2 & u'_2 \\
x & & u_1 & & u_2 & & u'_1 & & u'_2
\end{array}$$

The timing relations between the signals x and y of the buffer are independent from latency incurred by communications with its environment: this is the formal definition of endochrony given in [11].

Current limitations illustrated

Both sequential, concurrent and distributed code generation schemes in Polychrony rely on the property of endochrony to generate the code. This observation also holds for code generation in related synchronous languages, Lustre and Esterel, without much salient difference. However, it is well-known that endochrony is not preserved by composition. To illustrate that, consider the following pair of processes, a producer and a consumer. The producer increments its output u when its input a is true and increments its output x otherwise.

$$(u, x) = \mathsf{producer}(a) \stackrel{\text{def}}{=} \left(\begin{array}{l} \quad \hat{u} = [a] \quad \mid u = 1 + (u\,\mathsf{pre}\,0) \\ \mid\ \hat{x} = [\neg a] \mid x = 1 + (x\,\mathsf{pre}\,0) \end{array} \right) \qquad \begin{tikzcd}[column sep=small] & \hat{a} \arrow[dl, dash] \arrow[dr, dash] & \\ {[a] \sim \hat{u}} & & {[\neg a] \sim \hat{x}} \end{tikzcd}$$

The consumer adds the value of x to the count v when b is true and 1 otherwise. Hierarchies are depicted on the right.

$$y = \mathsf{consumer}(b, x) \stackrel{\text{def}}{=} \left(\begin{array}{rcl} \hat{v} &=& \hat{b} \\ \mid\ \hat{x} &=& \mathsf{when}\, b \\ \mid\ v &=& (v\,\mathsf{pre}\,0) + (x\,\mathsf{default}\,1) \end{array} \right) \qquad \begin{tikzcd}[column sep=small] & \hat{b} \sim \hat{v} \arrow[dl, dash] \arrow[dr, dash] & \\ {[b] \sim \hat{x}} & & {[\neg b]} \end{tikzcd}$$

The signal $x\,\mathsf{default}\,1$ is implicitly created. It has the same clock as b, its value is that of x at the clock $[b]$ and 1 at the clock $[\neg b]$. Notice that the producer and the consumer are endochronous (their hierarchies are trees). Now, consider the composition of $\mathsf{producer}$ and $\mathsf{consumer}$ in the main process below.

$$(u, v) = \mathsf{main}(a, b) \stackrel{\text{def}}{=} \left(\begin{array}{rcl} (u, x) &=& \mathsf{producer}(a) \\ \mid \quad v &=& \mathsf{consumer}(b, x) \end{array} \right)$$

Polychrony produces a hierarchy in which two synchronized boolean signals C_a and C_b are added on top of the hierarchies of the producer and the consumer. This allows to (artificially) form an endochronous simulation process that relies on the environment to determine when to read a and/or b.

$$\begin{tikzcd}
 & & C_a \sim C_b \arrow[dl, no head] \arrow[dr, no head] & & \\
 & {[C_a] \sim \hat{a}} \arrow[dl, no head] \arrow[dr, no head] & & {[C_b] \sim \hat{b} \sim \hat{v}} \arrow[dl, no head] \arrow[dr, no head] & \\
{[a] \sim \hat{u}} & & {[\neg a] \sim \hat{x} \sim [b]} & & {[\neg b]}
\end{tikzcd}$$

This structure yields the generation of code that differs from what we have seen so far in that the transition function now expects the clocks C_a and C_b to be synchronously delivered by the environment, instead of being computed internally.

In the C code, the functions `r_main_C_a`, `r_main_C_b`, `r_main_a` and `r_main_b` read the input signals C_a, C_b, a, b and the functions `w_main_u` and `w_main_v` write the outputs u and v.

The compiler places the clocks $[\neg a]$, $\hat{x}$ and $[b]$ in the same equivalence class. However, one easily notices that the equation $[\neg a] = [b]$, incurred by the composition of the producer and the consumer, is non-trivial. At present, it is bailed out as a so-called “clock constraint” by Polychrony.

To handle this clock constraint, Polychrony can either generate a proof obligation, which will have to be checked by the user, or generate defensive code to raise an exception if the clock constraint is violated during execution. In the generated code below, if `!a != b`, an exception is reported by the simulation loop.

```
bool main_iterate() {
    if (!r_main_C_a(&C_a)) return FALSE;
    if (!r_main_C_b(&C_b)) return FALSE;
    if (C_b) {
        if (!r_main_b(&b)) return FALSE;
    }
    if (C_a) {
        if (!r_main_a(&a)) return FALSE;
        C_ = !a;
        if (a) {
            u = 1 + u;
            w_main_u(u);
        }
        if (C_) {
            x = 1 + x;
        }
    }
    C__63 = (C_a ? C_ : FALSE);
```

```
          if ((C_) != b)
              polychrony_exception
                  ( "Exception for: (C_, b) " );
          if (C_b) {
              if (C__63) XZX_36 = x; else XZX_36 = 1;
              v = v + XZX_36;
              w_main_v(v);
          }
          C_ = FALSE;
          return TRUE;
      }
```

The functionality of Polychrony to detect and report such a constraint is central in the code generation scheme that will be presented next. Let us have a second look at the present situation:

- the producer and the consumer are endochronous
- the signal x is defined in one process, the producer
- its clock $\hat{x}$ is used in both processes

In the composition of the consumer and the producer, one can hence define x by a shared variable and use its clock constraint to define when it can deterministically be defined and/or used by either the processes.

4 Our contribution illustrated

Our contribution builds upon this simple idea, that is suitable for the simulation of otherwise deterministic specifications, and uses the facility of Polychrony to report clock constraints (such as $[b] = [\neg a]$) and to export independent clocks (such as C_a and C_b) to build a scheduler that satisfies the expected safety properties.

In this aim, and first of all, we would like to avoid increasing the interface of the program (with C_a or C_b) in order to have an efficient (sequential or concurrent) execution scheme. In the present code generation scheme of Polychrony, C_a and C_b are added to rebuild an endochronous simulation loop.

In the present case, however, the composition of the producer and the consumer is weakly endochronous: the very interleaving of a and b during execution is not relevant to the correct propagation of input and output values. The transitions involving only a or only b may be executed in any order. However, transitions involving both a and b need to be synchronized. This is precisely where the clock constraint $[b] = [\neg a]$ comes into play.

A definition of weak endochrony

A weakly endochronous system is a deterministic process in which independent actions, such as sending or receiving values through distinct signals, both

- can be triggered from the clock and value of higher signals, and,
- can be timely performed in any order (the state is unaffected)

For instance, the composition of the producer and of the consumer is weakly endochronous. The presence of all signals in this composition is locally determined from the value or presence of signals in either the producer or the consumer (it is deterministic) and the very order of execution: producer first, consumer first, or both, does not matter (they satisfy the diamond property). This is the very definition of weak endochrony in [14].

$$\begin{tikzcd}
 & \circ \arrow[dr, "\neg b\hat{v}"] & \\
\circ \arrow[ur, "a\hat{u}"] \arrow[rr, "a\hat{u}\neg b\hat{v}"] \arrow[dr, "\neg b\hat{v}"'] & & \circ \\
 & \circ \arrow[ur, "a\hat{u}"'] &
\end{tikzcd}$$

Building a controller

Using the information provided by Polychrony, namely, the exportation of non-hierarchized clocks C_a and C_b, the report of a clock constraint on shared signals such as $[b] = [\neg a]$, we can easily build a process for controlling the execution of the composition of the producer and the consumer so as to keep it within a suitable safety objective.

To allow for a correct resynchronization on the values of x, the controller needs to obey the requirement expressed by the clock constraint $[\neg a] = [b]$ while imposing no additional synchronization constraint (on a or b).

Nicely, this controller can be expressed and synthesized in Signal. It uses the clock constraint of the shared variable ($\hat{x} = \mathsf{when\ not}\, a = \mathsf{when}\, b$) to synchronize instants that need to be.

The controller accepts the input signals a and b and feeds the producer and the consumer with copies c and d until one of the constraints is met, $[\mathsf{when\ not}\, a]$ or $[\mathsf{when}\, b]$. As soon as this occurs, it stops reading input from the signal (a or b), suspending the corresponding process, until the other meets the constraint.

$$(c, d) = \mathsf{controller}(a, b) \stackrel{\text{def}}{=} \left(\begin{array}{l} \quad c = \mathsf{scheduler}(a, r_a, r) \\ \mid\ d = \mathsf{scheduler}(b, r_b, r) \\ \\ \mid r_a = \ \mathsf{not}\, a\, \mathsf{default}\, (r_a\ \mathsf{pre\ false}\,) \\ \mid r_b = b \qquad \mathsf{default}\, (r_b\ \mathsf{pre\ false}\,) \\ \mid\ \ r = r_a\, \mathsf{and}\, r_b \end{array}\right) / r_a r_b r$$

The controller contains two schedulers that are responsible for suspending and resuming the input signals a and b (hence the producer and the consumer) in order to correctly schedule the operations in the sequential implementation of the rendez-vous.

$$y = \mathsf{scheduler}(x, r_x, r) \overset{\text{def}}{=} \left(\begin{array}{rl} \hat{x} = & \mathsf{true\ when}\, c_x \\ | \, r_x = & \mathsf{not}\, a\, \mathsf{default}\, r'_x \\ | \, r'_x = & r_x\, \mathsf{pre\ false} \\ | \, c_x = & \quad\quad\quad (\mathsf{true\ when}\, (r\, \mathsf{pre\ false})) \\ & \mathsf{default}\ \ (\mathsf{false\ when}\, r'_x) \\ & \mathsf{default}\ \ \mathsf{true} \\ | \, c_y = & (c_x\, \mathsf{and\ not}\, r_x)\, \mathsf{or}\, r \\ | \ \ y = & (x\, \mathsf{cell}\, c_y)\, \mathsf{when}\, c_y \end{array} \right) / c_x c_y$$

Last, we need to patch the main program with the controller to correctly feed the producer and the consumer with the values of a and b that satisfy the clock constraint.

$$(u, v) = \mathsf{main}(a, b) \overset{\text{def}}{=} \left(\begin{array}{rl} (u, x) & = \mathsf{producer}(c) \\ | \quad\quad v & = \mathsf{consumer}(d, x) \\ | \ (c, d) & = \mathsf{controller}(a, b) \end{array} \right) / cdx$$

Notice that each of the producer and the consumer is able to independently react when either $[\neg a]$ or $[b]$ holds, as no synchronization needs to take place in those cases.

Sequential code generation scheme

The controller is build upon the clocks exported and the constraints reported by Polychrony. This provides sufficient information to generate the necessary code to control the execution of the composition of endochronous processes.

In the controlled main program, variables prefixed with `pre_` register the values of signal (of corresponding suffix) until the next cycle. The generated `r` variables translate the synchronization obligation implied by the reported clock constraint as `r = ra && rb`. Functions named `{r|w}_main_`x read and write the signal x.

As opposed to the generated code presented Section 3, the present program does not need its master clocks to be synchronized: C_a and C_b are local variables, not input signals. As a result, the interface of the composition of the producer and the consumer is the union of interfaces.

We observe that, since the producer and the consumer are endochronous, and since their composition is such that all clocks which can be computed (all have a disjunctive form), the main program is weakly isochronous in the sense of [14]: any synchronous reaction, initiated from one side, yields a globally isochronous execution. This yields to a generic methodological principle, presented next.

```
bool main_iterate() {

    /* c = scheduler (a, ra, r) */

    if (pre_r) C_a = TRUE;
       else if (pre_ra) C_a = FALSE;
       else C_a = TRUE;
    if (C_a) {
        if (!r_main_a(&a)) return FALSE;
    }
    if (C_a) ra = !a;
       else ra = pre_ra;

    /* d = scheduler (b, rb, r) */

    if (pre_r) C_b = TRUE;
       else if (pre_rb) C_b = FALSE;
       else C_b = TRUE;
    if (C_b) {
        if (!r_main_b(&b)) return FALSE;
    }
    if (C_b) rb = b;
       else rb = pre_rb;

    /* main */

    r = ra && rb;
    C_c = (C_a && !ra) || r;
    C_d = (C_b && !rb) || r;         }
```

```
/*  (x,u) = producer (c) */

C_1 = FALSE;
if (C_c) {
    C_1 = !a;
    if (a) {
        u = 1 + u;
        w_main_u(u);
    }
    if (C_1) x = 1 + x;
}

/*  y = consumer (d,x) */

C_2 = (C_c ? C_1 : FALSE);
if (C_d) {
    if (C_2) X_1 = x;
       else X_1 = 1;
    v = v + X_1;
    w_main_v(v);
}

/* finalisation */

pre_ra = ra;
pre_rb = rb;
pre_r = r;
return TRUE;
```

Contributed methodology

This simple observation translates into a global objective of preserving weak isochrony. It is defined by the following design methodology :

(1) if a process p is endochronous then p is weakly endochronous

(2) if p and q are weakly-endochronous and if $p \| q$ is both cycle-free and has clocks in disjunctive form, then $p \| q$ is weakly endochronous (and weakly isochronous as well).

Condition 1 allows us to define our methodology starting from the existing code generation tool-chain of Polychrony. It also implies that the same approach could be adapted to Lustre.

Condition 2 on the composition of p and q translates the condition that is imposed to a pair of weakly endochronous processes in [14] for their composition to be isochronous: the condition is, exactly, that any synchronous reaction initiated by p or q should yield an execution of $p \| q$.

A necessary condition is that the graph of $p \| q$ must be acyclic and a sufficient condition is that the $p \| q$ should not incur a reaction to the absence of a signal i.e. that all clocks in the graph of $p \| q$ should have a disjunctive form.

In our example, we observe that, should the main process be composed with an additional endochronous process (or weakly endochronous network), then we would only need to build an additional controller between those two, based on the same principle as previously mentionned: to capture the clocks exported by Polychrony and to implement rendez-vous between toplevel clock constraints (here: $\hat{b} = [c]$) in the hierarchy.

$$(u, w) = \mathsf{main2}(a, b, c) \overset{\text{def}}{=} \left(\begin{array}{rl} (u, v) &= \mathsf{main}(a, d) \\ \| \quad w &= \mathsf{consumer}(e, v) \\ \| \ (d, e) &= \mathsf{controller2}(b, c) \end{array} \right) / de$$

Concurrent code generation scheme

The generation of code for concurrent execution differs from sequential code generation by the construction of clusters that match the physical partition of signals on the target execution architecture. In the present case, these clusters are the composed endochronous processes, the producer and the consumer.

Our compilation technique for sequential code generation can easily be adapted for concurrent execution. It allows to define an interface or controller that performs minimum arbitration with its environment. As a result, producer and consumer are compiled separately and the global safety guarantee of weak isochrony is relied on assess the safety of the concurrent composition.

In the example, we have separately compiled the producer and consumer to ready them for concurrent execution. They use the local read/write functions of the producer and the consumer: `{r|w}_{consumer|producer}_`x). The clock constraint $[\neg a] = b$ is again used to synchronize the threads with a barrier: a mutex zone `RDV` is created to protect the shared variable x.

```
pthread_barrier_t *begin_RDV, *end_RDV ;
pthread_barrier_init(begin_RDV, 2);
pthread_barrier_init(end_RDV, 2);
```

```
bool consumer() {
  if (!r_consumer_b(&b)) return FALSE;
  if (b) {
     pthread_barrier_wait(begin_RDV);
     X_1 = x;
     pthread_barrier_wait(end_RDV);
  } else X_1 = 1;
  v = v + X_1;
  w_consumer_v(v);
  return TRUE;
}
bool producer() {
  if (!r_producer_a(&a)) return FALSE;
  if (a) {
     u = 1 + u;
     w_producer_u(u);
  }
  if (!a) {
     pthread_barrier_wait(begin_RDV);
     x = 1 + x;
     pthread_barrier_wait(end_RDV);
  }
  return TRUE;
}
```

The generated code is otherwise unchanged. We obtain a concurrent code generation scheme that modularly and compositionally supports separate compilation. It efficiently uses existing report functionalities of the present implementation of Polychrony to effectively support the synthesis of a controller that is able to assemble endochronous processes so as to maintain a global objective of weak isochrony.

5 Conclusions

We have introduced a sequential and concurrent code generation technique built upon existing functionalities of the Polychrony compiler to provide a modular and compositional way of assembling endochronous processes while maintaining a global safety objective of insensitivity to latency.

Our code generation scheme supports true separate compilation: a locally compiled synchronous module does not require its interface to the global environment be rebuilt once composed to another module since a compositional property of weak isochrony needs only be checked and locally translated into an equivalent controller or interface.

References

[1] P. Aubry. Mises en oeuvre distribuées de programmes synchrones. Thèse de l'Université de Rennes, October 1997.

[2] L. Besnard. Compilation de Signal: horloges, dépendances, environnements. Thèse de l'Université de Rennes, September 1992.

[3] A. Girault and X. Nicollin. Clock-driven automatic distribution of Lustre programs. International Conference on Embedded Software. Lectures notes in computer science, volume 2855. Springer Verlag, October 2003.

[4] O. Maffeïs. Ordonnancements de graphes de flots synchrones ; application à la mise en oeuvre de SIGNAL. Thèse de l'Universit? de Rennes, January 1993.

[5] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, v. 163. Academic Press 2000.

[6] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 2003.

[7] A. Benveniste, P. Caspi, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Embedded Software Conference*. Springer, 2003.

[8] C. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. The theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 20(9). IEEE Press, 2001.

[9] P. Caspi, A. Girault, D. Pilaud. Distributing Reactive Systems. International Conference on Parallel and Distributed Computing Systems. ISCA, 1994.

[10] P. Caspi, C. Mazuet, and N. Reynaud. About the design of distributed control systems: the quasi synchronous approach. In *International Conference on Computer Safety, Reliability and Security.* Springer, 2003.

[11] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*. World Scientific, 2003.

[12] N. Lynch and E. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, v. 82(1). Academic Press, 1989.

[13] Polychrony is available from http://www.irisa.fr/espresso/Polychrony.

[14] D. Potop-Butucaru and B. Caillaud and A. Benveniste. Concurrency in Synchronous Systems. In *Formal Methods in System Design*, v. 28(2). Springer, March 2006.

[15] Schneider, K., Brandt, J., Vecchié, E. Efficient code generation from synchronous programs. In *Methods and Models for Codesign*. IEEE Press, 2006.

[16] J.-P. Talpin, D. Potop-Butucaru, J. Ouy, B. Caillaud. From multi-clocked synchronous specifications to latency-insensitive systems. In *Embedded Software Conference*. ACM, 2005.

A Polychronous model of computation

We describe the semantics of Signal in the polychronous model of computation (see [11] for more detail). In this model, symbolic tags t or u denote periods in time during which execution takes place. Time is defined by a partial order relation $\leq$ on tags: $t \leq u$ stipulates that t occurs before u. A chain is a totally ordered set of tags. It corresponds to the clock of a signal: it samples its values over a series of totally related tags. The domains for events, signals, behaviors and processes are defined as follows:

- an *event* is a pair consisting of a tag $t \in \mathbb{T}$ and a value $v \in \mathbb{V}$,
- a *signal* is a function from a *chain* of tags to a set of values,
- a *behavior* b is a function from a set of signal names to signals,
- a *process* p is a set of behaviors that have the same domain.

We write $\mathcal{T}(s)$ for the chain of tags of a signal s and $\min s$ and $\max s$ for its minimal and maximal tag. We write $\mathcal{V}(b)$ for the domain of a behavior b (a set of signal names). The restriction of a behavior b to X is noted $b|_X$ (i.e. s.t. $\mathcal{V}(b|_X) = X$). Its complementary $b_{/X}$ (i.e. s.t. $\mathcal{V}(b_{/X}) = \mathcal{V}(b) \setminus X$) satisfies $b = b|_X \uplus b_{/X}$. We overload the use of $\mathcal{T}$ and $\mathcal{V}$ to talk about the tags of a behavior b and the set of signal names of a process p.
The synchronization of a behavior b with a behavior c is noted $b \leq c$ and is defined as the effect of "stretching" its timing structure. A behavior c is a *stretching* of a behavior b, written $b \leq c$, iff $\mathcal{V}(b) = \mathcal{V}(c)$ and there exists a bijection f on tags s.t.

$\forall tu \in \mathcal{T}(b), t \leq f(t) \wedge (t < u \Leftrightarrow f(t) < f(u))$

$\forall x \in \mathcal{V}(b), \mathcal{T}(c(x)) = f(\mathcal{T}(b(x))) \wedge \forall t \in \mathcal{T}(b(x)), b(x)(t) = c(x)(f(t))$

b and c are *clock-equivalent*, written $b \sim c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$. The synchronous composition $p \,|\, q$ of two processes p and q is defined by combining behaviors $b \in p$ and $c \in q$ that are identical on $I = \mathcal{V}(p) \cap \mathcal{V}(q)$, the interface between p and q.

$$p \,|\, q = \{b \cup c \,|\, (b, c) \in p \times q \;\wedge\; b|_I = c|_I \;\wedge\; I = \mathcal{V}(p) \cap \mathcal{V}(q)\}$$

The semantics $[\![P]\!]$ of a Signal process P is a set of behaviors that are inductively defined by the concatenation of reactions. A reaction r is a behavior with (at most) one time tag t. We write $\mathcal{T}(r)$ for the tag of a non empty reaction r. An empty reaction of the signals X is noted $\emptyset|_X$. The empty signal is noted $\emptyset$. A reaction r is concatenable to a behavior b iff $\mathcal{V}(b) = \mathcal{V}(r)$, and, for all $x \in \mathcal{V}(b)$, $\max(b(x)) < \mathcal{T}(r(x))$. If so, concatenating r to b is defined by

$$\begin{aligned} &\forall x \in \mathcal{V}(b), \forall u \in \mathcal{T}(b) \cup \mathcal{T}(r), \\ &\qquad (b \cdot r)(x)(u) = \text{if } u \in \mathcal{T}(r(x)) \text{ then } r(x)(u) \text{ else } b(x)(u) \end{aligned}$$

Initially, we assume that $\emptyset|_{\mathcal{V}(p)} \in [\![P]\!]$. The semantics of a delay $x = y \,\mathsf{pre}\, v$ is defined by appending a reaction r, of tag t, to a behavior b of $x = y \,\mathsf{pre}\, w$, for any value w (the previous-previous value of y). In the reaction r, y is present (i.e. $r(y) \neq \emptyset$) iff x is present and its value of is v (i.e. $r(x)(t) = v$). The previous value of y in behavior b, at tag u, is v.

$$[\![x = y \,\mathsf{pre}\, v]\!] = \left\{ b \cdot r \;\middle|\; \begin{aligned} \forall w \in \mathbb{V}, b \in [\![x = y \,\mathsf{pre}\, w]\!], t = \mathcal{T}(r), u = \max(\mathcal{T}(b(y))), \\ (r(x)(t) = v \wedge r(y) \neq \emptyset) \wedge (b(y)(u) = v \vee b = \emptyset_{xy}) \end{aligned} \right\}$$

Similarly, the semantics of a sampling $x = y \,\mathsf{when}\, z$ defines x by y when z is true.

$$[\![x = y \,\mathsf{when}\, z]\!] = \left\{ b \cdot r \;\middle|\; \begin{aligned} b \in [\![x = y \,\mathsf{when}\, z]\!], t = \mathcal{T}(r), (r(z)(t) = \mathsf{true} \wedge r(x) \\ = r(y)) \vee ((r(z)(t) \neq \mathsf{true} \vee r(z) = \emptyset) \wedge r(x) = \emptyset) \end{aligned} \right\}$$

Finally, $x = y \,\mathsf{default}\, z$ defines x by y when y is present and by z otherwise.

$$[\![x = y \,\mathsf{default}\, z]\!] = \left\{ b \cdot r \;\middle|\; b \in [\![x = y \,\mathsf{default}\, z]\!], r(x) = \left| \begin{matrix} r(y), \; r(y) \neq \emptyset \\ r(z), \; r(y) = \emptyset \end{matrix} \right. \right\}$$

The meaning of the synchronous composition $P \,|\, Q$ is the synchronous composition $[\![P \,|\, Q]\!] = [\![P]\!] \,|\, [\![Q]\!]$ of the meaning of P and Q. The meaning of restriction is defined by $[\![P/x]\!] = \{c \,|\, b \in [\![P]\!] \wedge c \leq (b/x)\}$.