On the Combination of Synchronous Languages ⋆

Axel Poigné, Leszek Holenderski

GMD-SET, Schloss Birlinghoven, D-53754 Sankt Augustin, Germany

1 Introduction

Synchronous languages [1, 4, 7, 9] address the specification and programming of *reactive* processes, i.e. processes which continuously respond to stimuli at a rate determined by the environment. The *synchrony hypothesis* [1] states that a process is fully responsible for the synchronization with its environment, that is:

- *event synchronization*: the process is always able to react to events of the environment at a rate determined by the environment;
- *response synchronization*: the response synchronizes properly with the environment, i.e., the time elapsed between a stimulus and the response of the process is short enough (relatively to the dynamics of the environment) so that the environment is still receptive to the response.

Furthermore, the behaviour of a process should be *reproducible* with regard to input events, or, in more technical terms, *deterministic*. Both these requirements are prerequisite for the dependable service of a process, for instance as a controller in a safety-critical environment such as an automobile, an aircraft, or a power station.

Available synchronous formalisms are quite different in focus and style:

- Data flow languages such as *Lustre* [4] or *Signal* [7] are particularly suited for representing periodic behaviour as is typical for "continuous" computation of sensor/actuator data,
- state-based languages such as *Esterel* [1] are better suited for "spontaneous" control behaviour, e.g. that of a mouse or a track pad.
- Graphical languages such as *Statecharts* [5] or *Argos* [9] are useful for structuring as, e.g., representing "change of mode" of a continuous system.

Existence of several synchronous formalisms is rather an advantage than a drawback: the formalisms are complimentary addressing different aspects of use. For instance, we may distinguish several modes of operation in a control application; a start mode, normal continuous behaviour, exception mode, and a termination mode. Lustre is most adeaquate for modelling the normal continous mode, and, maybe, an exception mode if its behaviour is cyclic, Argos is well suited to model change of mode, while Esterel may be useful for the start and termination modes which typically involve some sequential processing. Real applications

⋆ The work was partially funded by the Esprit LTR Action **SYRF**, "Synchronous Reactive Formalisms" (Esprit Project 22703).

address such issues, hence a combination of formalisms should prove itself to be useful which allows the user freely to merge the different points of view. This is issue we address in this paper.

The different styles of formalisms are reflected by the different styles of semantics and of implementation:

- Data-flow formalisms are based on a "flow" semantics where each signal is related to a trace of values

$$v_0 \quad v_1 \quad v_2 \quad \ldots$$

  Data flow programs constrain these traces. They essentially compile into (definitional) equation systems with additional registers. This compilation process amounts to a great extend to a clock resolution calculus.
- On the other hand state-based languages typically rely on an structural operational semantics. For compilation, these are symbolically presented by *purely boolean* equation systems, with additional boolean variables inserted to drive memory operations on data, these operations being stored in an *action table*. This corresponds to a separation between control and data flows.

Combination of formalisms presumes integration of semantics as well of the compilation schemes. This report sets up such a semantical framework as well as an language of reactive processes on which the translation schemes of the SYNCHRONIE WORKBENCH (SWB [14]) are based. In particular, some semantic invariants are specified which prove to be beneficial for a smooth integration. Section 2 addresses the semantic issues. In Section 3 we introduce an language of synchronous automata, our presentation of reactive processes, discuss invariants, and we present in Section 4 particular efficient translation schemes for control-based code. Section 5 deals with declarative code, while Section 6 is concerned with combination. Much of the work is based on and extends [12] and [8] as well as [11]. Our style is informal in order to present the ideas in as little space as possible, and we assume some familiarity with synchronous languages.

2 The Semantic Framework

2.1 The Model

Behaviour manifests in what we are able or want to observe. We classify observations by attributing a name we refer to as a *signal*. A signal s may be *present* having a value taken from a set $\mathcal{V}$, or it may be *absent*. Let $\mathcal{S}$ be the set of all signals.

We are concerned with *linear time* only as modeled by the ordered set ω of natural numbers. A *trace* δ_s of a signal s is specified by a subset $!\delta_s \subseteq \omega$, the *frequency* of δ_s, and a *valuation* $\delta_s : !\delta \to \mathcal{V}$ to a set of values. A *system trace* δ consists of a set of signal traces $\{\delta_s | s \in S\}$. We use $\Delta^{\mathcal{S}}$ to denote the domain of system traces, and speak of *synchronous* behaviour reflecting awareness of "global" time which allows to reason about the presence and the <u>absence</u> of a signal.

A system trace $\delta \in \Delta^{\mathcal{S}}$ is called a *flow* if $!\delta := \bigcup !\delta_s$ is downward closed, i.e. $m \in !\delta$ if $n \in !\delta$ and $m \leq n$. The idea is that passing of "time" is bound to an event, i.e. the presence of at least one signal. We refer to a set of flows as a *process*. Note that every set $\mathcal{T} \subseteq \Delta^{\mathcal{S}}$ of system traces determines a set $\mathcal{T}\downarrow$ of those flows which are in $\mathcal{T}$. The domain $\mathcal{P}roc$ of processes will be the semantic domain for synchronous languages.

Reflecting the synchrony hypothesis we require processes to be reactive, and deterministic:
Let $\mathcal{I} \subseteq \mathcal{S}$ be a set of *input signals*, and let $\Upsilon \in \Delta^{\mathcal{I}}$ be a set of (admissible) input flows. We say that a process $\mathcal{P}$ is *reactive* with regard to Υ if $\mathcal{P}_{\mathcal{I}} = \Upsilon$ where $\mathcal{P}_{\mathcal{I}} = \{\{\delta_s \mid s \in \mathcal{I}\} \mid \delta \in \mathcal{P}\}$ is the projection of $\mathcal{P}$ to input signals. Further, a process $\mathcal{P}$ is *deterministic* if $|\{\delta \in \mathcal{P} \mid \delta_{\mathcal{I}n} \in \Upsilon\}| \leq 1$. In other words, a process is reactive and deterministic if, for every input flow $\delta \in \Upsilon$, there is exactly one flow $\delta' \in \mathcal{P}$ such that $\delta = \delta'_{\mathcal{I}n}$.

We avoid explicitly discussing *typing* here (and elsewhere) but assume that all the relevant entities are (well-) typed in that the values of a signal s are chosen from a specified set $\mathcal{V}_s$. In particular, we assume existence of a type `bool` of Booleans with a constants `true` and `false`, and existence of a type `unit` with the only value being `void`. In the latter case we speak of a *pure signal* which is fully specified by its frequency.

Synchronous data-flow languages impose constraints on signal traces by lifting relations on data to traces:

$$R(\delta_1, \ldots, \delta_n) \text{ iff } \forall i \in \omega.\ [(R(\delta_1(i), \ldots, \delta_n(i)) \Leftrightarrow i \in \bigcap !\delta_j)].$$

where $R \subseteq \mathcal{V}^n$. Such a lifting is called *strong* if additionally

$$\bigcap !\delta_j \subseteq \bigcup !\delta_j$$

In contrast, we speak of a weak lifting. Strongness implies that all traces in a relation are of the same frequency. A relation is typically obtained by lifting a functional equation

$$\delta = f(\delta_1, \ldots, \delta_n)$$

where $f : \mathcal{V}^n \to \mathcal{V}$ is a function (we assume equality $\delta = \delta'$ to be a special case with f being the identity function).

The more interesting aspect of synchronous data-flow languages is that they provide a variety of operators for manipulating time indexes:

memorisation $\quad \delta' = \mathtt{pre}(\delta) \quad !\delta' = !\delta$

$$\delta'(i) = \begin{cases} \mathtt{init} & \text{if } i = 0 \\ \delta(\max\{m \in !\delta \mid m < i\}) & \text{else} \end{cases}$$

flèche	$\delta'' = \delta$ `->` δ'	$!\delta'' = !\delta' \cap !\delta$ $\delta''(i) = \begin{cases} \delta(i) & \text{if } i = \min !\delta \\ \delta'(i) & \text{else} \end{cases}$
downsampling	$\delta' = \delta$ `when` β	$!\delta' = !\delta \cap !\beta$ $\delta'(i) = \delta(i)$
upsampling	$\delta'' = \delta$ `default` δ'	$!\delta'' = !\delta \cup !\delta'$ $\delta''(i) = \begin{cases} \delta(i) & \text{if } i \in !\delta \\ \delta'(i) & \text{else} \end{cases}$

where β is of type `bool`. All operators except for the default operator are required to be strong. There is nothing particular about our choice of operators. Synchronous data flow languages such as LUSTRE [4] or SIGNAL [7] sport a different choice.

Elementary declarative programs consists of set of clauses which are interpreted disjunctively, plus a declaration of of input, output, and local signals, e.g.

```
node raising_edge (x:bool)(y:bool);
let
  y = false -> x and not pre(x);
tel
```

Notation and style considerably differs according to a specific language.

2.2 The Operational Model

We complement our semantic model by an *operational model* which is as simple: Let *synchronous computation* be specified by a labeled transition system $\mathcal{P}$ of the form

$$\sigma \xrightarrow[\mathcal{P}]{E/E'} \sigma'$$

where σ, σ' are states, and where $E \neq \emptyset$ indicates which signals are present when changing state. We refer to E as an *event*, and to single transition as an *instant* of time. The set E' specifies which signals are emitted by $\mathcal{P}$ at a given instant. We require that $E' \subset E$, i.e. the output event is part of the overall event. This property is referred to as *consistency*.

An event is specified by a partial function E from $\mathcal{S}$ to $\mathcal{V}$ which we present by its graph, i.e. the set $E \subseteq \mathcal{S} \times V$ such that $v = v'$ whenever $\langle s, v\rangle, \langle s, v'\rangle \in E$ (which justifies the subset notation above). Let $\mathcal{V}_\circ^\mathcal{S}$ denote the set of partial functions from $\mathcal{S}$ to $\mathcal{V}$.

States are of the form $\sigma \in \mathcal{V}_\circ^\mathcal{R}$ where $\mathcal{R}$ is a finite set of *registers* (where $\mathcal{S} \cap \mathcal{R} = \emptyset$). Registers behave similar to signals; a register $r \in \mathcal{R}$ may be *active* having a valuation $v \in \mathcal{V}$, or it may be *inactive*. The difference between signals and registers is that signals are set for the present instant while registers are set for the next instant.

In other words, our operational model is a Mealy machine with a particular structure of events and of states. We note that every sequence

$$\sigma_0 \xrightarrow[\mathcal{P}]{E_0/E_0'} \sigma_1 \xrightarrow[\mathcal{P}]{E_1/E_1'} \dots$$

of reactions specifies a flow $\delta \in \Delta^{\mathcal{S}}$ such that $n \in !\delta_s$ and $\delta_s(n) = v$ iff $\langle s, v\rangle \in E_n$.

2.3 Synchronous Automata

Favourite synchronous languages such as ESTEREL [1], or ARGOS [9], the synchronous variant of STATECHARTS [5] are closely related to this operational model. We present our own brand of very elementary language, *synchronous automata*, for programming such machines. Synchronous automata will serve as an intermediate layer for compilation schemes.

A synchronous automaton $\mathcal{P}$ consists of a set of *actions* of the form

$$\textbf{\textit{if}}\ \phi\ \textbf{\textit{then}}\ x = \epsilon$$

where x is a name, ϵ is a *data expression*, and ϕ is a pure (signal) expression (of appropriate form). For the semantics, let an *environment* be defined to be a union $\sigma \cup E$ with $\sigma \in \mathcal{V}_\circ^{\mathcal{R}}$, and $E \in \mathcal{V}_\circ^{\mathcal{S}}$. The notation is justified because of the isomorphism $\mathcal{V}_\circ^{\mathcal{R}+\mathcal{S}} \approx \mathcal{V}_\circ^{\mathcal{R}} \times \mathcal{V}_\circ^{\mathcal{S}}$. Then

$$\langle x, \epsilon(\sigma \cup E)\rangle \in \sigma' \cup E' \text{ iff } \sigma \cup E \models \phi$$

"if the condition ϕ holds in the environment $\sigma \cup E \in \mathcal{V}_\circ^{\mathcal{R}+\mathcal{S}}$, the the register or state x is set with the value obtained by evaluating ϵ in the environment $\sigma' \cup E'$". A *synchronous component* is a synchronous automaton wrapped with a header:

syn_aut *raising_edge* $(\alpha,\beta$*:unit;x:bool)(y:bool);*
register *pre_x:bool;*
let
  if $\alpha \ \vee\ \beta$ ***then*** *pre_x = x;*
  if α ***then*** *y = false;*
  if β ***then*** *y = x and not pre_x;*
tel

recodes the declarative raising edge program. The input parameters α and β explicitly represent the time index in the declarative model, α is true at the very first instant of time, and β is true at all instants of time except for the first one. Hence $\alpha \vee \beta$ represents the whole time scale ω. With regard to the original program, the `pre`-operator on time indices has been replaced by a register which stores the previous value.

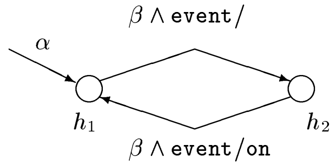
To give another example, let us consider the ESTEREL program

```
module one_bit (event:unit)(on:unit);
  loop
    await event;
    await event;
    emit on
  end
end
```

If started, after each second "event" the signal on is emitted. A corresponding synchronous automaton is specified by

> ***syn_aut*** *one_bit* $(\alpha,\beta,event{:}unit)(on{:}unit)$;
> ***register*** h_1, h_2 *:unit;*
> ***let***
> ***if*** $\alpha \vee (event \wedge \beta \wedge h_2) \vee (not\ event \wedge \beta \wedge h_1)$ ***then*** $h_1 = void$
> ***if*** $(event \wedge \beta \wedge h_1) \vee (not\ event \wedge \beta \wedge h_2)$ ***then*** $h_2 = void$
> ***if*** $event \wedge \beta \wedge h_2$ ***then*** $on = void$
> ***tel***

The same automaton implements the ARGOS diagram



2.4 Reactivity and Control

For the *semantics* of synchronous automata, we require (for the time being) that all actions are consistent: two actions (***if*** ϕ ***then*** $x = \epsilon$) and (***if*** ϕ' ***then*** $x' = \epsilon'$) are *consistent at* $\sigma \cup E$ if $x = x'$, and if $\epsilon(\sigma \cup E) = \epsilon'(\sigma \cup E)$ whenever $\sigma \cup E \models \phi$ and $\sigma \cup E \models \phi'$. Then a synchronous automaton determines a *transition function*

$$\mathcal{P}^{\triangleright} : \mathcal{V}_{\circ}^{\mathcal{R}+\mathcal{S}} \to \mathcal{V}_{\circ}^{\mathcal{R}}$$

and an *output function*

$$\mathcal{P}^{!} : \mathcal{V}_{\circ}^{\mathcal{R}+\mathcal{S}} \to \mathcal{V}_{\circ}^{\mathcal{S}}.$$

The reaction of an automaton at an instant may depend on the signals emitted by itself. Hence, given some set E of "inputs", the reaction should be stable in that

stability $\quad \mathcal{P}^{!}(\sigma \cup E) \subseteq E.$

holds. Moreover, we would like to distinguish between signals E which are broadcast by the environment, and those signals E' which are emitted by $\mathcal{P}$. We say that a reaction is *coherent* if each signal is either an "input" or emitted by $\mathcal{P}$:

coherence Given an "input" $E \in \mathcal{S}$, a reaction $E' \in \mathcal{S}$ is *coherent* if $E \subseteq E'$, E' is stable, and if $E' - P^!(\sigma, E \cup E') \subseteq E$.

We expect an automaton to react to every input coherently and reproducible:

reactivity For every input, there exists a unique[1] coherent reaction.

Given reactivity, the automaton P defines *reaction functions* $\tilde{P}^!$ and $\tilde{P}^{\triangleright}$ of the same arity as $P^!$ and $P^{\triangleright}$ such that $\tilde{P}^!(E) = E'$ and $\tilde{P}^{\triangleright}(E) = P^{\triangleright}(E')$ where E' is the unique coherent reaction with regard to the input E. The automaton P may be considered as presenting these reaction functions. Of course, not every automaton is reactive in that $\tilde{P}$ may fail to exist. It will be a matter of *causality analysis* [13] to weed those which are not.

So far the assumption is that, once started, a system reacts forever. For structuring purposes we would rather have that (sub-) systems may be *active* or *inactive*, or are *in control* or *out of control*, at different stages of evolution. We relate control to a particular set of pure registers $\mathcal{C}$, the *control registers*, and stipulate that a synchronous automaton is out of control if none of these registers is active. Then the automaton should not be able to react except for the trivial reaction. The idea is captured by the *control axiom*:

control $P^{\triangleright}(\sigma \cup E) = \emptyset$ and $P^!(\sigma \cup E) = \emptyset$ if $\sigma \cap \mathcal{C} = \emptyset$.

Note that a program such as the raising edge program does not a priori satisfy the control axiom. There is no obvious candidate for a control register. In fact, "control" is external here, hidden in the frequency $\chi = \alpha \vee \beta$; the automaton computes only if χ is present, hence the respective parameters.

2.5 The Initial Reaction

So far, synchronous automata only specify ongoing behaviours. Some activation mechanism is needed. We assume that the reaction of a system is immediate if "switched on", hence assume a *initial reaction* (rather than an initial state as in traditional automata theory). The initial reaction should not refer to some previous state in that both the *initial transition function*

$$\mathcal{P}^{\triangleright}_{\alpha} : \mathcal{V}_{\circ}^{\mathcal{S}} \to \mathcal{V}_{\circ}^{\mathcal{R}}$$

and the *initial output function*

$$\mathcal{P}^{!}_{\alpha} : \mathcal{V}_{\circ}^{\mathcal{S}} \to \mathcal{V}_{\circ}^{\mathcal{S}}.$$

do not depend on registers.

Syntactically, let $\mathcal{P}_{\alpha} \subseteq \mathcal{P}$, and $\mathcal{P}_{\beta} = \mathcal{P} - \mathcal{P}_{\alpha}$. We require that

initialization $\mathcal{P}^{\triangleright}_{\alpha}(E) \cup \mathcal{P}^{!}_{\alpha}(E) \neq \emptyset \Rightarrow \mathcal{P}^{\triangleright}_{\beta}(\sigma \cup E) \cup \mathcal{P}^{!}_{\beta}(\sigma \cup E) = \emptyset$

for all $\sigma \in \mathcal{V}_\circ^\mathcal{R}$ and $E \in \mathcal{V}_\circ^\mathcal{S}$; either an initial reaction takes place exclusively, or an reaction depends on a previous state. Note that $\sigma = \emptyset$ is a state which is, however, persistent due to the control axiom.

We will later use a specific pure *system signal*

α - *start*

which determines the initial reaction of a synchronous automaton. We than can rephrase initialization to

initialization $\quad \alpha \Rightarrow \neg \bigvee \mathcal{C}$

3 An Language of Synchronous Automata

3.1 Some Basic Operators and Predicates

Pursuing our program we introduce an intermediate layer of operators on synchronous automata each of which reflects semantic as well as programming concepts. A first definition of such an algebra has been given in [8] which carefully elaborates the choices made. Here we define a slightly revised version.

As a first step, we factor synchronous automata into a pure control part, we refer to as *Boolean automaton* and an "action table". All registers of the Boolean automaton are control registers. The two subautomata communicate via pure signals only. Hence we adopt a control-based view, but this does not restrict generality. Let $\mathcal{P}$ from now on range over Boolean automata.

Boolean automata will be enhanced by three predicates:

$\mathcal{P}.\omega$ - *termination*
$\mathcal{P}.\tau$ - *interrupt*
$\mathcal{P}.\eta$ - *control*

which are synthesized. The predicate $\mathcal{P}.\omega$ evaluates to true if $\mathcal{P}$ terminates, $\mathcal{P}.\tau$ evaluates to true if $\mathcal{P}$ issues an interrupt, and $\mathcal{P}.\eta$ evaluates to true if $\mathcal{P}$ is in control. We expect that the invariant

termination $\quad \mathcal{P}.\omega \Rightarrow \neg \mathcal{P}.\eta'$

holds where we use the notation $\ldots'$ to refer to the value at the next instant; if $\mathcal{P}$ terminates all control registers become inactive.

Now the most elementary of our operators are

$$
\begin{aligned}
s \;\texttt{<=}\; \phi &- \text{emit the pure signal } s\\
h \;\texttt{<-}\; \phi &- \text{activate the control register } h\\
\textit{nothing} &- \text{which does nothing, but terminates}\\
\mathcal{P} \vee \mathcal{Q} &- \text{the union of } \mathcal{P} \text{ and } \mathcal{Q}, \text{ and}\\
\phi \wedge \mathcal{P} &- \text{guarding } \mathcal{P} \text{ by the condition } \phi.
\end{aligned}
$$

where ***if*** $\phi \wedge \psi$ ***then*** $x = \epsilon$ is an action of $\phi \wedge \mathcal{P}$ if ***if*** ψ ***then*** $x = \epsilon$ is an action of $\mathcal{P}$. We stipulate:

- Emittance is defined by

$$s \ \texttt{<=}\ \phi \quad :\Leftrightarrow \quad \textbf{\textit{if}}\ \phi\ \textbf{\textit{then}}\ s = \mathit{void}.$$

  It terminates instantaneously, but neither raises an interrupt, nor keeps control:

$$\begin{aligned}(s \ \texttt{<=}\ \phi).\omega &= \mathit{tt}\\ (s \ \texttt{<=}\ \phi).\tau &= \mathit{ff}\\ (s \ \texttt{<=}\ \phi).\eta &= \mathit{ff}\end{aligned}$$

- Activation is defined by

$$h \ \texttt{<-}\ \phi \quad :\Leftrightarrow \quad \mathit{if}\ \phi\ \mathit{then}\ h = \mathit{void}.$$

  If the register is activated then control is kept, and no termination takes place. No interrupt is raised:

$$\begin{aligned}(h \ \texttt{<=}\ \phi).\omega &= \neg\mathtt{h}\\ (h \ \texttt{<=}\ \phi).\tau &= \mathit{ff}\\ (h \ \texttt{<=}\ \phi).\eta &= \mathtt{h}\end{aligned}$$

- *nothing* has the empty set of actions

$$\mathit{nothing} \quad :\Leftrightarrow \quad \emptyset,$$

  and it terminates instantaneously

$$\begin{aligned}\mathit{nothing}.\omega &= \mathit{tt}\\ \mathit{nothing}.\tau &= \mathit{ff}\\ \mathit{nothing}.\eta &= \mathit{ff}\end{aligned}$$

- Disjunction means disjunction:

$$\begin{aligned}(\mathcal{P} \vee \mathcal{Q}).\omega &= \mathcal{P}.\omega \vee \mathcal{Q}.\omega\\ (\mathcal{P} \vee \mathcal{Q}).\tau &= \mathcal{P}.\tau \vee \mathcal{Q}.\tau\\ (\mathcal{P} \vee \mathcal{Q}).\eta &= \mathcal{P}.\eta \vee \mathcal{Q}.\eta\end{aligned}$$

- Guarding raises an interrupt signal, and restricts termination, but does not affect control[2]:

$$\begin{aligned}(\phi \wedge \mathcal{P}).\omega &= \phi \wedge \mathcal{P}.\omega\\ (\phi \wedge \mathcal{P}).\tau &= \phi\\ (\phi \wedge \mathcal{P}).\eta &= \mathcal{P}.\eta\end{aligned}$$

[2] This abstracts a mechanism introduced by Reinhard Budde for compilation of *synchronousEifel* [3]

Further, we have already introduced a number of operators implicitly:

$\mathcal{P}_\alpha$	- the initial reaction
$\mathcal{P}_\beta$	- the ongoing reaction
$\mathcal{P}^\triangleright$	- the transition function
$\mathcal{P}^!$	- the output function

This completes the set of basic operators.

A number of familiar operators are easily specified in terms of these base operators. We give a couple of examples using our syntactical convention that the system signal α specifies the first reaction.

$$\mathit{emit}\ s = s \texttt{ <= } \alpha \tag{1}$$

$$\mathit{halt}^h = h \texttt{ <- } \alpha \vee h \tag{2}$$

$$\mathit{start}\ \mathcal{P}\ \mathit{at}\ \phi = (\phi \wedge \mathcal{P}_\alpha) \vee \mathcal{P}_\beta \tag{3}$$

$$\mathit{if}\ \phi\ \mathit{then}\ \mathcal{P}\ \mathit{else}\ \mathcal{Q}\ \mathit{fi} = (\mathit{start}\ \mathcal{P}\ \mathit{at}\ \phi) \vee (\mathit{start}\ \mathcal{Q}\ \mathit{at}\ \neg\phi) \tag{4}$$

$$\mathit{if}\ \phi\ \mathit{then}\ \mathcal{P}\ \mathit{fi} = \mathit{if}\ \phi\ \mathit{then}\ \mathcal{P}\ \mathit{else}\ \mathit{nothing}\ \mathit{fi} \tag{5}$$

$$\mathcal{P}\ ;\ \mathcal{Q} = \mathcal{P} \vee (\mathit{start}\ \mathcal{Q}\ \mathit{at}\ \mathcal{P}.\omega) \tag{6}$$

$$\mathit{loop}\ \mathcal{P}\ \mathit{end} = \mathcal{P} \vee (\mathit{start}\ \mathcal{P}\ \mathit{at}\ \mathcal{P}.\omega) \tag{7}$$

$$\mathit{terminate}\ \mathcal{P}\mathit{if}\ \phi = \mathcal{P}^! \vee (\neg\phi \wedge \mathcal{P}^\triangleright) \tag{8}$$

$$\mathit{terminate}\ \mathcal{P}\mathit{if}\ \mathit{next}\ \phi = \mathcal{P}^! \vee \mathcal{P}^\triangleright_\alpha \vee (\neg\phi \wedge \mathcal{P}^\triangleright_\beta) \tag{9}$$

$$\mathit{cancel}\ \mathcal{P}\mathit{if}\ \phi = \neg\phi \wedge \mathcal{P} \tag{10}$$

$$\mathit{cancel}\ \mathcal{P}\mathit{if}\ \mathit{next}\ \phi = \mathcal{P}_\alpha \vee (\neg\phi \wedge \mathcal{P}_\beta) \tag{11}$$

$$\mathit{await}\ \phi = \mathit{cancel}\ \mathit{halt}^h\ \mathit{if}\ \phi \tag{12}$$

$$\mathit{await}\ \mathit{next}\ \phi = \mathit{cancel}\ \mathit{halt}^h\ \mathit{if}\ \mathit{next}\ \phi \tag{13}$$

$$\mathit{do}\ \mathcal{P}\mathit{when}\ \phi = (\phi \wedge \mathcal{P}) \vee (\neg\phi \wedge \mathit{KEEP}) \tag{14}$$

$$\mathit{do}\ \mathcal{P}\mathit{when}\ \mathit{next}\ \phi = \mathcal{P}_\alpha \vee (\phi \wedge \mathcal{P}_\beta) \vee (\neg\phi \wedge \mathit{KEEP}) \tag{15}$$

where $\mathit{KEEP} = \{h \texttt{ <- } h \mid h \in \mathcal{C}\}$. In words:

1. The signal s is emitted in the first instant only.
2. The control register h is activated in the first instant, and then kept activated.
3. $\mathcal{P}$ starts only if the condition ϕ holds.
4. If ϕ holds then $\mathcal{P}$ is executed, otherwise $\mathcal{Q}$.
5. obvious.
6. $\mathcal{P}$ computes first. If $\mathcal{P}$ terminates, $\mathcal{Q}$starts to compute.
7. $\mathcal{P}$ is immediately reinitialised if it terminates.
8. $\mathcal{P}$ looses control if ϕ holds (*weak preemption*)
9. As above but preemption does not take place in the first instant.
10. As (8), but signals are not emitted either (*strong preemption*)
11. As above but preemption does not take place in the first instant.

12. Whenever the control register h gets control, control will stay with h till the condition ϕ holds. If ϕ holds in the first instant, h does not get control.
13. As above, but h gets control in the first instant whether or not the condition holds.
14. $\mathcal{P}$ is down sampled in that $\mathcal{P}$ computes only if the condition ϕ holds, otherwise the present status of control registers is kept. Note that the process is out of control instantaneously if the condition ϕ holds in the first instant, but does not terminate.
15. Here $\mathcal{P}$ is started in the first instant whatever ϕ says.

With regard to preemption it may be worthwhile to observe that the diagram

signals at first instant	*signals at later instants*
registers at first instant	*registers at later instants*

displays all the preemption strategies of our model. Overall there are sixteen different strategies starting with no preemption at all up to preemption of all signals and registers. The latter is the strategy specified by the operator *cancel* $\mathcal{P}$ *if* ϕ. A modification is to cancel only at later instants which covers the two squares on the right, and which more or less corresponds to the *do ... watching* mechanism of ESTEREL. Preemption by termination does not affect signals hence covers only the two lower squares, being a mild variant of the *trap*-statement of ESTEREL. If applied only in later instants, termination covers exactly the lower square on the right. The latter corresponds to the preemption mechanisms used in ARGOS.

3.2 Concerning Compositionality

Our set of base operators has several defects with regard to compositionality, meaning that our semantical requirements / invariants are not preserved.

Most notably, disjunction does not preserve reactivity: e.g.

$$(a \texttt{ <= } b) \vee (b \texttt{ <= } a)$$

is not well behaved though its components are. This is well known and inherent, and there is no way to for a compositional analysis of reactivity. Hence we abandon any hope of a compositional solution but use global causality analysis as everybody does.

However, disjunction does not preserve the control axiom either: e.g.

$$(h \texttt{ <= } \mathit{tt}) \vee (a \texttt{ <= } \mathit{tt})$$

terminates though it keeps control. In fact, disjunction is only a very useful auxiliary operator, and the basis of

$\mathcal{P} \otimes \mathcal{Q}$ - parallel composition

where $\mathcal{P} \otimes \mathcal{Q}$ is equivalent to $\mathcal{P} \vee \mathcal{Q}$ except that we have a new termination condition:

$$(\mathcal{P} \otimes \mathcal{Q}).\omega = (\mathcal{P}.\omega \wedge \mathcal{Q}.\omega) \vee (\mathcal{P}.\omega \wedge \mathcal{P}.\eta \wedge \neg\mathcal{Q}.\eta) \vee (\mathcal{Q}.\omega \wedge \mathcal{Q}.\eta \wedge \neg\mathcal{P}.\eta)$$

The parallel of $\mathcal{P}$ and $\mathcal{Q}$ terminates if both automata terminate at the same instant $(\mathcal{P}.\omega \wedge \mathcal{Q}.\omega)$, or if $\mathcal{Q}$ has already terminated and $\mathcal{P}$ terminates $(\mathcal{P}.\omega \wedge \mathcal{P}.\eta \wedge \neg\mathcal{Q}.\eta)$, and vice versa. As a point of fine tuning, observe that, in the latter cases, both automata must be in control (i.e. $\mathcal{P}.\eta \wedge \mathcal{Q}.\eta$ holds); otherwise, e.g., $\mathcal{P}$ may terminate in the first instant, hence the parallel computation as well since $\mathcal{Q}.\eta$ must be false because of the initialization axiom. However, $\mathcal{Q}$ may gain control as in $(h \texttt{ <= } \mathit{tt}) \vee (a \texttt{ <= } \mathit{tt})$.

As a kind of dual, we have

$\mathcal{P} \oplus \mathcal{Q}$ - choice

where $\mathcal{P} \oplus \mathcal{Q}$ is equivalent to $\mathcal{P} \vee \mathcal{Q}$ except that we require that

$$\mathcal{P}^{\triangleright}_{\alpha}(E) \cap \mathcal{Q}^{\triangleright}_{\alpha}(E) = \emptyset;$$

only one of $\mathcal{P}$ or $\mathcal{Q}$ can obtain control. We can now state a first "compositionality" result:

Proposition 1. *The operators $s \texttt{ <= } \phi$, $h \texttt{ <- } \phi$, `nothing`, $\mathcal{P} \otimes \mathcal{Q}$, $\mathcal{P} \oplus \mathcal{Q}$, and $\phi \wedge \mathcal{P}$ preserve the control axiom.*

Having in mind this proposition, the resulting strategy should be to replace the ill-behaved $\mathcal{P} \vee \mathcal{Q}$ by the well-behaved $\mathcal{P} \otimes \mathcal{Q}$ or $\mathcal{P} \oplus \mathcal{Q}$ exploiting

Lemma 1. $\mathcal{P}_{\alpha} \vee \mathcal{P}_{\beta} = \mathcal{P}_{\alpha} \otimes \mathcal{P}_{\beta}$

Inspection of the derived operators above proves that we could have used

$$\begin{aligned}
\mathit{start}\ \mathcal{P}\ \mathit{at}\ \phi &= (\phi \wedge \mathcal{P}_{\alpha}) \otimes \mathcal{P}_{\beta} \\
\mathit{if}\ \phi\ \mathit{then}\ \mathcal{P}\ \mathit{else}\ \mathcal{Q}\ \mathit{fi} &= (\phi \wedge \mathcal{P}_{\alpha}) \oplus \mathcal{P}_{\beta} \vee (\neg\phi \wedge \mathcal{Q}_{\alpha}) \vee \mathcal{Q}_{\eta} \\
\mathcal{P}\ ;\ \mathcal{Q} &= \mathcal{P} \oplus (\mathit{start}\ \mathcal{Q}\ \mathit{at}\ \mathcal{P}.\omega) \\
\mathit{loop}\ \mathcal{P}\ \mathit{end} &= (\mathcal{P} \otimes (\mathit{start}\ \mathcal{P}\ \mathit{at}\ \mathcal{P}.\omega) \\
\mathit{terminate}\ \mathcal{P}\ \mathit{if}\ \phi &= \mathcal{P}^{!} \otimes (\neg\phi \wedge \mathcal{P}^{\triangleright}) \\
\mathit{cancel}\ \mathcal{P}\ \mathit{if}\ \phi &= \neg\phi \wedge \mathcal{P} \\
\mathit{terminate}\ \mathcal{P}\ \mathit{if}\ \mathit{next}\ \phi &= \mathcal{P}^{!} \oplus \mathcal{P}^{\triangleright}_{\alpha} \otimes (\neg\phi \wedge \mathcal{P}^{\triangleright}_{\beta}) \\
\mathit{cancel}\ \mathcal{P}\ \mathit{if}\ \mathit{next}\ \phi &= \mathcal{P}_{\alpha} \otimes (\neg\phi \wedge \mathcal{P}_{\beta}) \\
\mathit{do}\ \mathcal{P}\ \mathit{when}\ \phi &= (\phi \wedge \mathcal{P}) \oplus (\neg\phi \wedge \mathit{KEEP}) \\
\mathit{do}\ \mathcal{P}\ \mathit{when}\ \mathit{next}\ \phi &= \mathcal{P}_{\alpha} \otimes (\phi \wedge \mathcal{P}_{\beta}) \oplus (\neg\phi \wedge \mathit{KEEP})
\end{aligned}$$

rather than the original definitions.

4 Adding Efficiency

4.1 System Signals

Implementation of a concrete languages has one other prerogative beside semantic transparency: efficiency of the resultant code in terms of time and space. The operators of our algebra may obviously fail to produce such code. However, concrete languages use the operators in rather specific ways. If we are able to discern these specific patterns, and to implement them efficiently, we may get the best of both worlds, semantic transparency and efficiency of the generated code.

We use particular pure signals, so-called *system signals* to represent semantic concepts in a convenient fashion. One of these operators has already been introduced, the *start* signal α. The idea of α is to specify the notion of the first instant, hence α corresponds to the operator $\mathcal{P}_\alpha$: let $\mathcal{P}$ be a synchronous automaton. Let us define

$$\mathcal{P}_\alpha := \mathcal{P}[\mathit{ff}/\mathcal{C}] \quad \text{and} \quad \mathcal{P}_\beta := \mathcal{P}[\mathit{ff}/\alpha]$$

where $\mathcal{P}[\mathit{ff}/\alpha]$ states that we substitute ff for α on the right hand side of actions in $\mathcal{P}$. Similarly, $\mathcal{P}[\mathit{ff}/\mathcal{C}]$ states that ff is substituted for each of the control registers. We give an example rather than a formal definition:

$$\begin{aligned}
&h_1 \ \texttt{<-}\ \alpha \vee (\mathit{event} \wedge \beta \wedge h_2) \vee (\neg \mathit{event} \wedge h_1)\\
&h_2 \ \texttt{<-}\ (\mathit{event} \wedge \beta \wedge\ h_1) \vee (\neg \mathit{event} \wedge h_2)\\
&\mathit{on} \ \texttt{<=}\ \mathit{event} \wedge\ h_2
\end{aligned}$$

where $\mathtt{h}_1$ and $\mathtt{h}_2$ are control registers. Then $\mathcal{P}_\alpha$ is defined by

$$h_1 \ \texttt{<-}\ \alpha,$$

and $\mathcal{P}_\beta$ by

$$\begin{aligned}
&h_1 \ \texttt{<-}\ (\mathit{event} \wedge \beta \wedge h_2) \vee (\neg \mathit{event} \wedge h_1)\\
&h_2 \ \texttt{<-}\ (\mathit{event} \wedge \beta \wedge\ h_1) \vee (\neg \mathit{event} \wedge h_2)\\
&\mathit{on} \ \texttt{<=}\ \mathit{event} \wedge\ h_2.
\end{aligned}$$

In order to make the start system semantically well behaved we require that

$$(\alpha) \qquad \mathcal{P} \cong \mathcal{P}_\alpha \otimes \mathcal{P}_\beta.$$

The example, quite deliberately, suggests existence of a system signal β, we refer to as *run* signal: if β is not present, no computation will take place but the status of control variables is retained. The run signal provides for a simple implementation of the *when next*-operator, or dually the *suspend*-operator:

$$\mathcal{P} \ \mathit{when\ next}\ \phi = \mathcal{P}[\phi \wedge \beta/\beta]$$

The run signal behaves semantically correct if

$(\beta) \qquad \mathcal{P} \cong \mathcal{P}_\alpha \otimes (\beta \wedge \mathcal{P}_\beta \oplus \neg\beta \wedge \mathit{KEEP}).$

Next, we have a system signal τ, the *preempt* signal, which, if present, deactivates all control registers for the next instant. Semantically we require that

$(\tau) \qquad \mathcal{P} \cong \mathcal{P}^{!} \oplus \neg\tau \wedge \mathcal{P}^{\triangleright}.$

These are all the system signals we shall use for efficient translation schemes of control based programs.

4.2 Starring

Substitution is an "expensive" operation due to possible code multiplication. In order to be more efficient we can use "lazy" substitution, i.e. we introduce a *local* signal, bind the respective term to this signal, and substitute the signal instead of the term:

$$\mathcal{P}[\phi/x] \quad \text{becomes} \quad \mathcal{P}[\gamma/x] \vee (\gamma \texttt{ <= } \phi)$$

This operation is constant in size, however not quite equivalent in our setup. For example, consider $\mathcal{P}[\alpha \vee h/\alpha]$ where $\mathcal{P} = (a \texttt{ <= } \alpha)$, and where h is a control register. Clearly $\mathcal{P}[\alpha \vee h/\alpha] = (a \texttt{ <= } \alpha \vee h)$, and

$$\mathcal{P}[\alpha \vee h/\alpha]_\alpha = (a \texttt{ <= } \alpha) \quad \text{and} \quad \mathcal{P}[\alpha \vee h/\alpha]_\beta = (a \texttt{ <= } h)$$

On the other hand we have

$$\begin{array}{l}(\mathcal{P}[\gamma/\alpha] \oplus (\gamma \texttt{ <= } \alpha \vee h))_\alpha = (\gamma \texttt{ <= } \alpha) \vee (a \texttt{ <= } \gamma) \\ (\mathcal{P}[\gamma/\alpha] \oplus (\gamma \texttt{ <= } \alpha \vee h))_\beta = (\gamma \texttt{ <= } h) \vee (a \texttt{ <= } \gamma)\end{array}$$

Obviously, the operators of our algebra of concrete synchronous automata are not compositional with regard to local variables.

We resolve the problem by adding a new operation, *starring*, which renames all local signals. Let $\mathcal{L}(\mathcal{P})$ be the set of local signals of $\mathcal{P}$. Then

$$\mathcal{P}^* := \mathcal{P}[s^*//s \mid s \in \mathcal{L}(\mathcal{P})]$$

where $\mathcal{P}[s^*//s \mid s \in \mathcal{L}(\mathcal{P})]$ states that every local signal $s \in \mathcal{L}(\mathcal{P})$ is renamed to s^* everywhere in $\mathcal{P}$ (this is different from substitution which affects only the ϕ's in $s \texttt{ <= } \phi$ and $h \texttt{ <- } \phi$). Again an example should be sufficient to grasp the idea:

$$((\gamma \texttt{ <= } \alpha) \vee (a \texttt{ <= } \gamma))^* = (\gamma^* \texttt{ <= } \alpha) \vee (a \texttt{ <= } \gamma^*)$$

where γ is a local variable. Thus we should in general redefine $\mathcal{P}_\alpha$ to

$$\mathcal{P}_\alpha = \mathcal{P}[\mathit{ff}/\mathcal{C}]^*.$$

Then lazy substitution is compositional, with our trivial example hopefully being enough of a witness to substatiate the claim. These observations have used in [12] where we give a translation of ESTEREL and prove its correctness.

4.3 Reincarnation

The staring operator resolves another subtlety of synchronous languages related to the loop construct: reincarnation [1]. For example, if control is with *await c* in

```
loop
  signal a in
    if a then emit b;
    await c;
    emit a
  end
end
```

then presence of c will imply emittance of a, reinitialisation of the loop, evaluation of the *if*-statement, and finally control will again be with the *await*-statement. The question is whether b will be emitted or not. It should not: in a block structured language, leaving a block will forget all bindings. Entering the block a new incarnation of bound variables, here a, is generated. Due to coherence this new incarnation cannot be present, hence b is not emitted. Note that we have two incarnations of a existing at the same instant, one incarnation being present, the other not.

With a being local, starring generates two copies, a and a^*, the latter covers the reentry of the loop. Applying the definitions we roughly obtain the automaton

$$\begin{aligned}
&b \texttt{ <= } a^* \wedge (\alpha \vee c \wedge h) \\
&h \texttt{ <- } \alpha \vee (c \wedge h) \vee (\neg c \wedge h) \\
&a \texttt{ <= } c \wedge h
\end{aligned}$$

which behaves perfectly well.

4.4 Complexity

The *splitting* of $\mathcal{P}$ into $\mathcal{P}_\alpha$ and $\mathcal{P}_\beta$ may cause quadratic growth. Assume we have the action

$$\mathcal{P} \quad \equiv \quad (a \texttt{ <= } b \wedge (\alpha \vee h_1))$$

with h being a control register. Then splitting generates two copies of b,

$$\begin{aligned}
\mathcal{P}_\alpha \quad &\equiv \quad (a \texttt{ <= } b \wedge \alpha) \\
\mathcal{P}_\beta \quad &\equiv \quad (a \texttt{ <= } b \wedge h_1)
\end{aligned}$$

Now we may need to substitute $(\mathcal{P}_\alpha \otimes \mathcal{P}_\beta)[\gamma/\alpha] \otimes \{\gamma \texttt{ <= } c \wedge (\alpha \vee h_2)\}$, and to split again which leaves us with three copies of b,

$$\begin{aligned}
(\mathcal{P}_\alpha \otimes \mathcal{P}_\beta)[\gamma/\alpha]_\alpha \quad &\equiv \quad (a \texttt{ <= } b \wedge \gamma*) \vee (\gamma^* \texttt{ <= } b \wedge \alpha) \\
(\mathcal{P}_\alpha \otimes \mathcal{P}_\beta)[\gamma/\alpha]_\beta \quad &\equiv \quad (a \texttt{ <= } b \wedge h_1) \vee (a \texttt{ <= } b \wedge \gamma) \vee (\gamma \texttt{ <= } b \wedge h_2).
\end{aligned}$$

In general, we get a geometric sum $\sum_{i=1}^{n} i = n^2$ with n being the number of nested splittings.

Further, splitting is expensive in terms of translation time since every subexpression needs to be touched. Hence, splitting should be avoided whenever this is feasible.

4.5 An Imperative Core

With these definition, the basic operators of an imperative synchronous language can be implemented as follows, omitting the predicates $\mathcal{P}.\omega, \mathcal{P}.\beta, \mathcal{P}.\eta$ the definition of which are not changed:

$$\begin{aligned}
\textit{start } P \textit{ at } \phi^{\ \alpha\ \beta\ \tau} &= P^{\ \gamma\ \beta\ \tau} \vee (\gamma \ \texttt{<=}\ \phi) \\
\textit{loop } P \textit{ end}^{\ \alpha\ \beta\ \tau} &= P_{\alpha}^{\alpha\ \beta\ \tau} \vee P^{\ \gamma\ \beta\ \tau} \vee (\gamma \ \texttt{<=}\ P^{\ \alpha\ \beta\ \tau}.\omega) \\
\textit{terminate } P \textit{ if } \phi^{\ \alpha\ \beta\ \tau} &= P^{\ \alpha\ \beta\ \tau'} \vee (\tau' \ \texttt{<=}\ \tau \vee \phi) \\
\textit{terminate } P \textit{ if next } \phi^{\ \alpha\ \beta\ \tau} &= P^{\ \alpha\ \beta\ \tau'} \vee (\tau' \ \texttt{<=}\ \tau \vee (\phi \wedge P^{\ \alpha\ \beta\ \tau}.\eta)) \\
\textit{do } P \textit{ when next } \phi^{\ \alpha\ \beta\ \tau} &= P^{\ \alpha\ \beta'\ \tau} \vee (\beta' \ \texttt{<=}\ \beta \wedge \phi)
\end{aligned}$$

All the signals named with Greek letters are local except for α. We have

Proposition 2. *All these operators satisfy the control axiom, the initialization axiom, and they satisfy the* $(\alpha), (\beta), (\tau)$ *axiom.*

The proof works very much along the lines of that given in [12] except that we have axiomatized some of invariants in this proof. Most cases are in fact, straightforward but tedious.

All the other operators are in fact derived:

$$\begin{aligned}
\textit{do } \mathcal{P} \textit{ when } \phi &= \textit{start (do } \mathcal{P} \textit{ when next } \phi\textit{) at } \phi \\
\textit{cancel } \mathcal{P} \textit{ if } \phi &= \textit{cancel (do } \mathcal{P} \textit{ when } \neg\phi\textit{) if } \phi \\
\textit{cancel } \mathcal{P} \textit{ if next } \phi &= \textit{terminate (do } \mathcal{P} \textit{ when next } \neg\phi\textit{) if next } \phi
\end{aligned}$$

We have reduced the number of splitting to one case, the *loop*-construct, but the more intuitive definition

$$\textit{loop } P \textit{ end} = P^{\ \gamma\ \beta\ \tau} \vee (\gamma \ \texttt{<=}\ \alpha \vee P^{\ \alpha\ \beta\ \tau}.\omega)$$

may be incorrect semantically. In an inductive proof, the initialisation axiom is crucial for avoiding the splitting operator. However, this may fail to hold in case of the "intuitive " definition of the loop; if we reenter a loop some control register of $\mathcal{P}$ must have been active in contrast to the initialisation axiom. Hence proper use of the start symbol cannot be guaranteed. Our definition circumvents the problem because there are no interferences between $\mathcal{P}_{\alpha}$ and $\mathcal{P}_{\beta}$, which means that the initialisation axiom holds (by brute force so to speak). Splitting is, however, in many cases unnecessary, and it is a matter of efficiency to anticipate such cases.

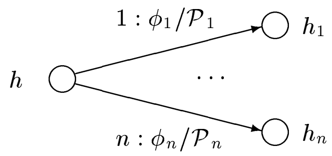
The operators discussed are, modulo syntactic sugar, those of ESTEREL V5 except for traps which need an additional attribute not covered here (but in [12]). We omit the discussion of traps not only because of the additional space needed but as well as we believe that traps are pragmatically unsound in that few users control the inherent priority scheme.

4.6 State-based Control

As an exercise we give the translation of a state-based language such as ARGOS or STATECHARTS in terms of the operators above. Let each state h correspond to a control register h. We collect all the out-going transitions from h to some h_i which we assume to be triggered if a condition ϕ_i holds. While changing state a synchronous automaton $\mathcal{P}_i$ will be executed. We use the textual notation

$$\textit{trans } h : \phi_1/P_1; \ldots; \phi_n/\mathcal{P}_n \textit{ end}$$

for



The numbering specifies priorities in the graphical presentation. Then, for each state h, we have

$$\textit{trans } h : \phi_1/\mathcal{P}_1; \ldots; \phi_n/\mathcal{P}_n \textit{ end}^{\alpha\ \beta\ \tau} = \textit{HALT} \otimes \textit{TRAN}$$

where

$$\begin{aligned} \textit{HALT} &\equiv (\textit{start } (\textit{cancel halt}^h \textit{ at } \tau_h) \textit{ at } \alpha_h) \vee (\tau_h \texttt{ <= } \tau \vee \textit{TRAN}.\tau) \\ \textit{TRAN} &\equiv \textit{start } (\textit{if } \phi_1 \textit{ then } \mathcal{P}_1; \textit{emit } \alpha_{h_1} \textit{ end}; \ldots \\ &\qquad\qquad ; \phi_n \wedge (\mathcal{P}_n; \textit{emit } \alpha_{h_1})) \textit{ at } h \end{aligned}$$

For each state h, α_h is a local signal: if α_h is present the control register h is activated for the next instant. Only then *TRAN* is initialized. If one of the conditions ϕ_i holds the register h will be preempted, and the $\mathcal{P}_i$ will be executed.

For the hierarchical structure one may add

$$\textit{trans } h = \mathcal{P} \ : \ \phi_1/\mathcal{P}_1; \ldots; \phi_n/\mathcal{P}_n \textit{ end}$$

to state that $\mathcal{P}$ refines the state h, and redefine *HALT* to:

$$\begin{aligned} \textit{HALT} \equiv\ & (\textit{start } (\textit{cancel halt}^h \otimes (\textit{do } \mathcal{P} \textit{ when next } h) \textit{ at } \tau_h) \textit{ at } \alpha_h) \\ & \vee (\tau_h \texttt{ <= } \tau \vee \textit{TRAN}\eta) \end{aligned}$$

4.7 Data

Dealing with data one has to be more specific about the "action table'. We consider here only a very simple kind of data action, *assignment to a memory cell*:

$$(\textbf{\textit{if}}\ \chi\ \textbf{\textit{then}}\ x := \epsilon) \equiv (\textbf{\textit{if}}\ \chi\ \textbf{\textit{then}}\ x = \epsilon) \vee (\textbf{\textit{if}}\ \neg\chi\ \textbf{\textit{then}}\ x = x)$$

Here χ is a pure *trigger* signal, x is a data register, and ϵ is a data expression (of suitable type). At every instant, the data expression is computed and its values assigned to the memory cell, provided that the trigger signal is present. Otherwise the old value is kept. Hence the assignment notation. Emittance of a value is then implemented by

$$\textit{emit}\ s(\epsilon) = (s\ \texttt{<=}\ \alpha) \vee (a\ \texttt{<=}\ \alpha) \vee (\textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ ?s := \epsilon)$$

where a is a pure (non-local !) signal. We use a "dual-rail" implementation in that every *valued signal* s is presented by a pure signal and a memory register $?s$. The notation $?s$ refers to the value of the register if used in an expression. The declaration of a valued signal is of the form

$$\textit{signal}\ x\ (:\ \textit{type})\ \textit{in}\ \mathcal{P}\textit{end.}$$

The signal is pure if the type information is missing.

Since control may depend on values of the language we allow to use $?b$ as a pure atomic expression (overloading notation). This completes the simple protocol of how a Boolean automaton and an action table communicate.

5 Declarative Code - Dealing with Frequencies

5.1 Generating Synchronous Automata

Declarative statements define constraint on flows (cf. Section 2.1). We will concentrate on two such statements, *declarations*

$$\textit{flow}\ x\ (:\ \textit{type})\ \textit{when}\ \mathcal{E}\ \textit{in}\ \mathcal{P}\ \textit{end}$$

and *equations*

$$x = \mathcal{E}$$

We assume that the declaration specifies the frequency $!x$ of x which, as we freely admit, is a restricted view related to LUSTRE. $\mathcal{P}$ is a synchronous automaton, and $\mathcal{E}$ is a *valued synchronous automaton*, i.e. a pair $\mathcal{E} = \langle \epsilon, \mathcal{E} \rangle$ with ϵ being an data expression, and $\mathcal{E}$ being a synchronous automaton (overloading notation). We require that $\mathcal{E}\eta = \mathit{ff}$, i.e. the automaton related to an expression terminates instantaneously. The value of $\mathcal{E}$ is computed at the frequency $!\mathcal{E}$ which is a synthesized predicate.

Let, for every pure signal s, $!s$ determine its "base frequency" such that $!s \prec\ s$ where

$$s \prec s' \quad \text{iff, for all events } E \text{, if } s \in E \text{ then } s' \in E$$

defines a preorder on signals; if s is part of an event then s' is part of it as well.

Then declarations translate to (forgetting about types)

$$\textit{flow } x \textit{ when } \mathcal{E} \textit{ in } \mathcal{P} \textit{ end} \equiv \mathcal{E} \otimes \mathcal{P} \vee (!x \texttt{ <= } !\mathcal{E} \wedge ?b) \vee (\textbf{\textit{if}}\ !\mathcal{E} \textbf{\textit{ then }} ?b := \epsilon)$$

where

$$\begin{aligned}(\textit{flow}\ldots).\omega &= \mathcal{P}.\omega\\ (\textit{flow}\ldots).\tau &= \mathcal{P}.\tau\\ (\textit{flow}\ldots).\eta &= \mathcal{P}.\eta\end{aligned}$$

and equations to

$$(x = \mathcal{E}) \equiv \mathcal{E} \vee (\chi \texttt{ <= } !x \wedge !\mathcal{E}) \vee (\textbf{\textit{if}}\ \chi \textbf{\textit{ then }} ?x := \epsilon)$$

where

$$\begin{aligned}(x = \mathcal{E}).\omega &= \mathit{tt}\\ (x = \mathcal{E}).\tau &= \mathit{ff}\\ (x = \mathcal{E}).\eta &= \mathit{ff}.\end{aligned}$$

Data expressions generate valued synchronous automata in a straightforward way:

$$\begin{aligned}\mathit{op}(\mathcal{E}_1, \ldots, \mathcal{E}_n) &\equiv \langle \mathit{op}(\epsilon_1, \ldots, \epsilon_n), \mathcal{E}_1 \otimes \ldots \otimes \mathcal{E}_n \rangle\\ !\mathit{op}(\mathcal{E}_1, \ldots, \mathcal{E}_n) &= \bigwedge !\mathcal{E}_j\end{aligned}$$

Flow variables translate to

$$x = \langle ?x, \mathit{stop} \rangle.$$

All declarative synchronous languages share the concept of memorization which is an operator on expressions, and which is implemented by

$$\begin{aligned}\mathit{pre}(\mathcal{E}) &\equiv \langle m, \mathcal{E} \vee (\textbf{\textit{if}}\ !\mathcal{E} \textbf{\textit{ then }} m \texttt{ := } \epsilon) \rangle\\ !\mathit{pre}(\mathcal{E}) &= !\mathcal{E}.\end{aligned}$$

m is a new register. A complimentary initialization operator has many incarnations, Lustre uses $\mathcal{E}_1$ `->` $\mathcal{E}_2$ which translates to

$$\begin{aligned}(\mathcal{E}_1 \texttt{ -> } \mathcal{E}_1) \equiv \langle \epsilon, \mathcal{E}_1 \otimes \mathcal{E}_2 \vee &\\ &\{\chi_\alpha \texttt{ <= } !\mathcal{E}_1 \wedge \neg h,\\ &\ \textbf{\textit{if}}\ \chi_\alpha \textbf{\textit{ then }} \epsilon := \epsilon_1,\\ &\ \chi_\beta \texttt{ <= } !\mathcal{E}_2 \wedge h,\\ &\ \textbf{\textit{if}}\ \chi_\beta \textbf{\textit{ then }} \epsilon := \epsilon_2\}\rangle\\ !(\mathcal{E}_1 \texttt{ -> } \mathcal{E}_2) = !\mathcal{E}_1 \wedge !\mathcal{E}_2 &\end{aligned}$$

The pure register h allows to distinguish the first instant the frequency $!\mathcal{E}_1$ is present from later instants. The assumption is that h is inactive when starting the computation. However, h is not a control register.

Downsampling is implemented by

$$
\begin{aligned}
(\mathcal{E}_1 \ \mathit{when}\ \mathcal{E}_2) &\equiv \langle \epsilon, (\mathcal{E}_1 \ \mathit{when}\ \chi) \vee\ \mathcal{E}_2 \\
&\qquad\qquad\qquad \vee\ (\mathit{if}\ !\mathcal{E}_1\ \mathit{then}\ b := \epsilon_2) \vee\ (\chi\ \texttt{<=}\ !\mathcal{E}_1 \wedge b) \\
&\qquad\qquad\qquad \vee (\mathit{if}\ \chi\ \mathit{then}\ \epsilon := \epsilon_1 \rangle \\
!(\mathcal{E}_1 \ \mathit{when}\ \mathcal{E}_2) &= \chi \\
!\chi &= !\mathcal{E}
\end{aligned}
$$

At the frequency of $\mathcal{E}_1$, the value is changed if $\mathcal{E}_2$ computes to tt. The new value is that of $\mathcal{E}_1$ at this instant. The upsampling operator in LUSTRE the *current*-operator:

$$
\begin{aligned}
\mathit{current}(\mathcal{E}) &\equiv \mathcal{E} \\
!\mathit{current}(\mathcal{E}) &= !!\mathcal{E}
\end{aligned}
$$

Changes only depend on $\mathcal{E}$, otherwise the value is latched, but on the faster frequency.

5.2 Checking Frequencies

While being an acceptable implementation the above fails to satisfy strongness of equality (cf. Section 2.1) for which one needs that

$$!x \ \prec\ !\mathcal{E} \quad \text{and} \quad !\mathcal{E}\ \prec\ !x.$$

Frequency analysis is applied to reject programs which do not satisfy such frequency requirements, similar to causality analysis which rejects causally incorrect programs. Frequency analysis, of course, is a problem of comparing boolean flows, ergo maps to the satisfiability problem of Boolean expression which is NP-hard. The problem gets even worse due to the presence of memorization. To reduce complexity, only an approximation $\chi \prec\!\!\prec \chi'$ of $\chi \prec \chi'$ is considered. The approximations may be more or less sophisticated. SIGNAL offers an elaborate "clock" calculus, while LUSTRE promote a more down-to-earth approach which will be sketched.

In LUSTRE the frequency of a flow is determined by its declaration

```
flow x (: type) when e
```

Let us bind $\mathtt{e}$ to x by $\Delta(x) = \mathtt{e} :: \Delta(\mathtt{e})$ to obtain a stack of expressions, where

$$
\begin{aligned}
\Delta(\mathtt{op}()) &= [\ ] \\
\Delta(\mathtt{op}(\mathtt{e}_1,\ldots,\mathtt{e}_n)) &= \Delta(\mathtt{e}_1), && \text{provided that } \Delta(\mathtt{e}_1) = \ldots = \Delta(\mathtt{e}_n) \\
\Delta(\mathtt{pre}(\mathtt{e})) &= \Delta(\mathtt{e}) \\
\Delta(\mathtt{e}_1\ \texttt{->}\ \mathtt{e}_2) &= \Delta(\mathtt{e}_1), && \text{provided that } \Delta(\mathtt{e}_1) = \Delta(\mathtt{e}_2) \\
\Delta(\mathtt{e}_1\ \mathtt{when}\ \mathtt{e}_2) &= \mathtt{e}_2 :: \Delta(\mathtt{e}_1), && \text{provided that } \Delta(\mathtt{e}_1) = \Delta(\mathtt{e}_2) \\
\Delta(\mathtt{current}(\mathtt{e})) &= \mathit{tl}(\Delta(\mathtt{e}))
\end{aligned}
$$

and let

$$!\mathsf{e} \prec\!\!\prec !\mathsf{e}' \Leftrightarrow \Delta(\mathsf{e}) = \Delta(\mathsf{e}')$$

lifting the frequency operator ! to the syntactic level.

Lemma 2. *Let $\mathcal{E}$ and $\mathcal{E}'$ be the valued synchronous automata obtained by translating the expressions* e *and* e*'. Then*

$$!\mathcal{E} \prec !\mathcal{E}' \text{ if } !\mathsf{e} \prec\!\!\prec \; e'.$$

The frequency analysis can be refined if flow variables are substituted by the expressions defining their frequency.

5.3 A More Liberal Policy

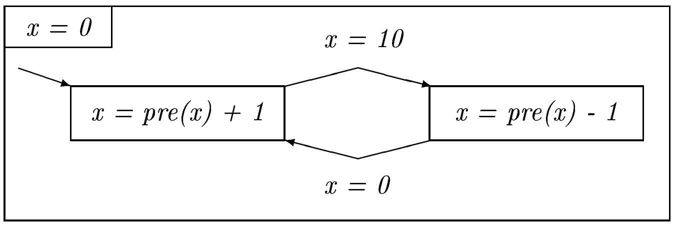
The requirement that equalities of flows are strong may be too rigid a requirement. A flow definition such as

$$x = 1 \textit{ when } \phi$$
$$x = 2 \textit{ when } \neg\phi$$

is well defined since at each instant of time only one definition applies. In terms of the flow model (cf. Section 2.1) this means that each such equation generates a flow $!x_j$ indexed by the respective equation and that

$$!x = \bigcup !x_j$$

specifies the frequency of x with the proviso that the frequencies $!x_j$ are pairwise disjoint. An application of this idea can be for instance found in [10] where different flow equation operate in different states of an Argos automaton:



6 Finally - the Combination

6.1 Adding Control to Equations

The synchronous automata generated by the declarative code does not provide any mechanism to preempt and to (re-) initialize a computation. In more technical terms, it does not satisfy the control axiom which is pivotal for controlling computations. As a brute force solution, we just add a control register to each equation

$(x = e)^{\alpha\ \beta\ \tau} = \mathcal{E}\ \otimes\ halt^h$
$\vee\, (\chi$ <= $!x \wedge !\epsilon \wedge (\alpha \vee \beta \wedge h))$
$\vee\, (\textbf{\textit{if}}\ \chi\ \textbf{\textit{then}}\ x := \epsilon)$

Whenever started the equation is evaluated until the control register is preempted. This is inefficient in terms of the number of control registers added, but effective. In a more efficient translation, equations share such a control register, in a pure declarative program only one such register is needed, the one relates to the base frequency. One should note though that now *equations and imperative statements may be freely mixed*, e.g. a list $\langle eq_1, \ldots, eq_n \rangle$ of equation translates to $eq_1^{\alpha\ \beta\ \tau} \otimes \ldots \otimes eq_1^{\alpha\ \beta\ \tau}$.

6.2 A (Re-)View of Data

We have distinguish three kings of variable entities: *valued signals*, for short *signals*, *flows*, and *program variables*, for short *variables*. The latter have not been discussed yet. There are subtle differences which are probably best explained with regard to our implementation model.

If a valued signal s is emitted several times at the same instance, non-determinism may arise in that different values may be assigned to $?s$. This non-determinism can be resolved

- by adding an associative *combine* operator, e.g. addition in case of integers, which operates on all the values $v_1, \ldots, v_n$ emitted at the same instant in that the assignment $?s := v_1 + \ldots + v_n$ takes place, or
- by *scheduling* the emittances.

The strategy of Esterel is to have :

- *valued signals* which may have a combine operator, but otherwise non-deterministic behaviour is (should be) rejected, and
- *program variables* which are scheduled (according to the program structure), which are never present, but always has a value (which is the standard notion of a program variable).

E.g. the Esterel code

```
x := 1 ; x := x + 1
```

with `x` being a program variable is evaluated at the same instant, scheduled sequentially and yields a result 2 for `x`. The statement

```
x := 1 || x := x + 1
```

is rejected because of non-determinism. Similarly,

```
emit s(1) || emit s(?s + 1)
```

will be rejected if not, e.g., addition is a combine operator for `s`. The scheduling strategy of valued signals as well as that of flows is that all ''writes" at an instant should precede all ''reads". This rules out a construction such as

```
emit s(1) ; emit s(?s + 1).
```

Our semantic modeling does not cover scheduling which can be extended by adding scheduling information on the level of actions, at the expense of definitions being more cumbersome. Causality analysis has to take care of this flow of data as well as of the flow of control.

The difference between flows and valued signals is more subtle. Of course, flows are ''type-checked" in that frequencies are constrained while this does not apply to valued signals. However, this is not the essential difference since we may assume that valued signals run at (some relative) base frequency. The more serious problem is due to the fact that declarative code usually assumes that every variable has a unique equational definition, otherwise interferences of definitions may occur: consider the equation

$$x = 0 \texttt{ -> } 1 + \texttt{pre}(x)$$

which counts the number of instants. If a second equation

$$x = 1$$

is computed in parallel the behaviour is either non-deterministic, and will be rejected which would be the same for valued signals. Applying a combine operator, e.g. $+$, would, however, result in unacceptable behaviour from the declarative point of view in that uniqueness of definition (at a instant) is violated. Hence, a combine operator should not be related to flows.

All the properties are summarized in the table

x	signal	flow	variable
presence	x		
value	$?x$	x	x
clock		$\surd$	
combine	$\surd$		
schedule	r $\prec$ w	r $\prec$ w	$\surd$

where x is a name of the respective entity.

What are the consequences?

- Either we consider a unified data model as we did in [6] introducing the language *LEA* as a combination of Lustre, Esterel, and Argos. based on a unified data model,
- or we accept the differences of the data models but are liberal in the way the concepts interoperate as we do in this paper using coercions.

There are various possible coercions, and it is a matter of a language designer to make these explicit or not:

- a valued signal or a program variable x can be coerced into a flow by providing some frequency, e.g. by overloading the downsampling operator: `x when e`.

– a flow x cannot turned into a valued signal because uniqueness of definition may be lost due to a combine operator. We propose to overload the notation in that the pair $!x$, $?x$ is of kind *signal*; $!x$ refers to the clock of x, and $?x$ to the value. However, to preserve uniqueness of definition we translate to

$$\langle \epsilon, \textbf{\textit{if}}\ !x\ \textbf{\textit{then}}\ \epsilon :=?x\rangle$$

where ϵ is a new memory cell.

The hinge between components of different nature are the declarations of input and output parameters. Their kind should be specified by the keywords `signal`, `flow`, and `var`, e.g.

```
node raising_edge (flow x:bool)(signal y:bool);
flow z:bool;
let
   z = false -> x and not pre(x)
 ||
   if !z then emit y(?z)
tel
```

Then a component call, e.g.,

```
raising_edge (x' when true) (y')
```

is well defined where `x'` and `y'` are signals. We propose

```
node raising_edge (flow x:bool)(signal_of_flow y:bool);
let
   y = false -> x and not pre(x)
tel
```

as a shorthand notation to the same effect.

7 Conclusion

We have presented a unified view of declarative and control-based synchronous programming based on very few, semantically meaningful operators on synchronous automata. We claim no originality with regard to the notion of synchronous automata which are closely related to Berry's hardware interpretation [2], though our more algebraic approach, first described in [12] and later extended in [8] (which we never bothered to publish, but were we tried to give, for ourselves, an account of the ideas underlying the Synchrony Workbench) may be mildly interesting, even novel. Of course, the real implementation uses some more shortcuts to increase efficiency of the generated code. The shortcuts do not affect correctness because they are always based on well understood assumptions. Anyway, we start from a semantically well-defined basic scheme which already proved to be reasonably efficient if implemented as is, and which has proved to be extremely versatile. The latter is, in fact, the rationale of the Synchrony Workbench: to have a generic framework for synchronous programming.

References

1. Berry, G., Gonthier, G., The synchronous programming language Esterel: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
2. Berry, G, A hardware implementation of pure Esterel. *Sadhana, Academy Proceedings in Engineering Sciencies, Indian Academy of Sciences*, 17:95–130, 1992.
3. Budde, R., Sylla, K.H., Objekt-orientierte Echtzeitanwendungen auf Grundlage perfekter Synchronisation, *Object-Spektrum*, Feb. 1995
4. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D., The synchronous data flow programming language Lustre, *Proceedings of the IEEE*, 79(9):1305–1321, Sept. 1991.
5. Harel, D., STATECHARTS: A Visual Formalism for Complex Systems, *Science of Computer Programming*, 8(3)231-274,1987
6. Holenderski, L., Poigné, A., The Multi-Paradigm Synchronous Programming Language *LEA*, submitted, 97
7. Le Guernic, P., Gautier, T., , Le Borgne, M., , Le Maire, C., Programming Real-time Applications with SIGNAL, *Proceedings of the IEEE*, 79(9), Sept. 1991
8. Maffeis, O., Poigné, A., Synchronous Automata for Reactive, Real-time, and Embedded Systems, Arbeitspapiere der GMD 964, Forschungszentrum Informationstechnik GmbH, Jan. 19965. (`http://set.gmd.de/EES/papers/SAforRRES/SAforRRES.html`)
9. Maraninchi,F., Operational and compositional semantics of synchronous automaton compositions, In *Proceedings of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, Springer-Verlag, 550–564, Aug. 1992.
10. Maranchini, F, Y.Rémond, Y., Mode-Automata: About Modes and States in Reactive Systems, Research Report, Verimag, 1997
11. Poigné, A., Maffeis, O. ,Morley, M., Holenderski, L., Budde, R., The Synchronous Approach to Designing Reactive Systems, In: *Formal Methods in System Design*, Kluwer, 1998 (an earlier version can be found at `http://set.gmd.de/EES/papers/SP.ps.gz`)
12. Poigné, A., Holenderski, L., Boolean automata for implementing pure Esterel, Arbeitspapiere der GMD 964, Forschungszentrum Informationstechnik GmbH, 1–47, Dec. 1995. (`http://set.gmd.de/EES/papers/E2BA.ps.gz`)
13. Shiple, T.R., Berry, G., Toutai, H., Constructive Analysis of Cyclic Circuits, In : *Proceedings of the European Design and Testing Conference*, IEEE Computer Society, March 1996
14. The *Synchrony Workbench*, `http://set.gmd.de/EES/synchrony/swb.html`