

# Chapter 1

## A Module Language for Typing SIGNAL programs by Contracts

Yann Glouche\*, Thierry Gautier, Paul Le Guernic, and Jean-Pierre Talpin

### 1.1 Introduction

Methodological guidelines for the design of real-time embedded systems advise the validation of specifications as early as possible. Moreover, in a refinement-based development methodology of large embedded systems, an iterative validation of each refinement or modification made to the initial specification, until the implementation of the system is finalized, is highly desirable. Additionally, cooperative component-based development requires to use and to assemble components, which have been developed by different suppliers, in a safe and consistent way [11, 17]. These components have to be provided with their conditions of use and guarantees that they have been validated when these conditions are satisfied. These conditions of use and guarantees represent a notion of *contract*. Contracts are now often required as a useful mechanism for validation in robust software design. Design by Contract, as advocated in [26], is being made available for usual languages like C++ or Java. Assertion-based contracts express program invariants, pre- and post-conditions, as Boolean type expressions that have to be true for the contract being validated. We adopt here a different paradigm of contract to define a component-based validation technique in the context of a synchronous modeling framework. In our theoretical model, a component is represented by an abstract view of its behaviors. It has a finite set of input/output variables to cooperate with its environment. Behaviors are viewed as multi-set traces on the variables of the component. The abstract model of a component is thus a *process*, defined as a set of such behaviors.

A *contract* is a pair (*assumptions*, *guarantees*). *Assumptions* describe properties expected by a component to be satisfied by the context (the environment) in which

---

INRIA, centre INRIA de Rennes - Bretagne-Atlantique, Campus de Beaulieu, Rennes, France, e-mail: Yann.Glouche@inria.fr, Thierry.Gautier@inria.fr, Paul.LeGuernic@inria.fr, Jean-Pierre.Talpin@inria.fr

\* Partially funded by the EADS Foundation.

this component is used; on the opposite, *guarantees* describe properties that are satisfied by the component itself when the context satisfies the *assumptions*. Such a contract may be documentary; however, when a suitable formal model exists, contracts can be supplied to some formal verification tool. We want to provide designers with such a formal model allowing “simple” but powerful and efficient computation on contracts. Thus, we define a novel algebraic framework to enable formal reasoning on contracts.

The assumptions and guarantees of a component are defined as *process-filters*: assumptions *filter* the processes (sets of behaviors) a component may accept and guarantees *filter* the processes a component provides. A process-filter is the set of processes, *whatever their input and output variables are*, that are compatible with some property (or constraint) *expressed on the variables of the component*. Foremost, we define a Boolean algebra to manipulate process-filters. This yields an algebraically rich structure that allows us to reason about contracts (to abstract, refine, combine and normalize them). This algebraic model is based on a minimalist model of execution traces, allowing one to adapt it easily to a particular design framework.

A main characteristic of this model is that it allows one to precisely handle the variables of components and their possible behaviors. This is a key point. Indeed, assumptions and guarantees are expressed, as usual, by properties constraining or relating the behaviors of some variables. What has to be considered very carefully is thus the “compatibility” of such constraints with the possible behaviors of *other variables*. This is the reason why we introduce partial order relations on processes and on process-filters. Moreover, having a Boolean algebra on process-filters allows one to formally, unambiguously and finitely express complementation within the algebra. This is, in turn, a real advantage compared to related formalisms and models.

We put this algebra to work for the definition of a general purpose module system whose typing paradigm is based on the notion of contract. The type of a module is a contract holding assumptions made on and guarantees offered by its behaviors. It allows to associate a module with an interface which can be used in varieties of scenarios such as checking the composability of modules or efficiently supporting modular compilation. The corresponding module language is generic in that processes and contracts may be expressed in some external target language. In the context of real-time, safety-critical applications, we consider here the synchronous language SIGNAL to specify processes.

## Organization

We start with a highlight on some key features of our module system by considering the specification of a protocol for Loosely Time-Triggered Architectures, Section 2. This example is used in this article to illustrate our approach. We give an outline of our contract algebra, Section 3, and demonstrate its capabilities for logical and compositional reasoning on the assumptions and guarantees of component-based embedded systems. This algebra is used, Section 4, as a foundation for the defi-

nition of a strongly-typed module system: contracts are used to type components with behavioral properties. Section 5 demonstrates the use of our module system by considering the introductory example and by illustrating its contract based specification.

## 1.2 A case study

We illustrate our approach by considering a protocol that ensures a coherent system of logical clocks on the top of Loosely Time-Triggered Architectures (LTTA). This protocol has been presented in [7]. We define contracts to characterize properties of this protocol.

### 1.2.1 Description of the protocol

In general, a distributed real-time control system has a time-triggered nature just because the physical system for control is bound to physics. A LTTA features a quasi-periodic and non-blocking bus access and independent read-write operations. The LTTA is composed of three devices, a *writer*, a *bus*, and a *reader* (Figure 1.1). Each device is activated by its own, approximately periodic, clock.

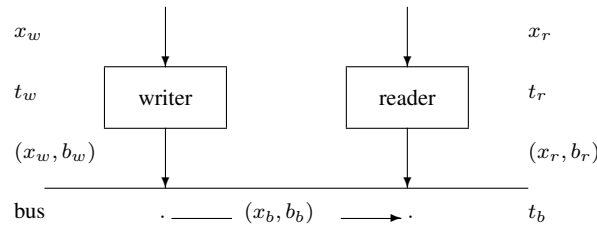


Fig. 1.1: The three devices of the LTTA.

At the  $n$ th clock tick (time  $t_w(n)$ ), the *writer* generates the value  $x_w(n)$  and an alternating flag  $b_w(n)$  s.t.:

$$b_w(n) = \begin{cases} false & \text{if } n = 0 \\ not\ b_w(n-1) & \text{otherwise} \end{cases}$$

Both values are stored in its output buffer, denoted by  $y_w$ . At any time  $t$ , the writer's output buffer  $y_w$  contains the last value that was written into it:

$$y_w(t) = (x_w(n), b_w(n)), \text{ where } n = \sup\{n' \mid t_w(n') < t\} \quad (1.1)$$

At  $t_b(n)$ , the *bus* fetches  $y_w$  to store in the input buffer of the reader, denoted by  $y_b$ . Thus, at any time  $t$ , the reader input buffer is defined by:

$$y_b(t) = y_w(t_b(n)), \text{ where } n = \sup\{n' \mid t_b(n') < t\} \quad (1.2)$$

At  $t_r(n)$ , the *reader* loads the input buffer  $y_b$  into the variables  $x(n)$  and  $b(n)$ :

$$(x(n), b(n)) = y_b(t_r(n))$$

Then, the reader extracts  $x(n)$  iff  $b(n)$  has changed. This defines the sequence  $m$  of ticks:

$$\begin{aligned} m(0) &= 0, \quad m(n) = \inf\{k > m(n-1) \mid b(k) \neq b(k-1)\} \\ x_r(k) &= x(m(k)) \end{aligned} \quad (1.3)$$

In any execution of the protocol, the sequences  $x_w$  and  $x_r$  must coincide, i.e.,  $\forall n. x_r(n) = x_w(n)$ . In [7] it is proved that this is the case iff the following conditions hold:

$$w \geq b \text{ and } \lfloor \frac{w}{b} \rfloor \geq \frac{r}{b}, \quad (1.4)$$

where  $w$ ,  $b$  and  $r$  are the respective periods of the clocks of the writer, the bus and the reader (for  $x \in \mathbb{R}$ ,  $\lfloor x \rfloor$  denotes the largest integer less or equal to  $x$ ). Conditions (1.4) are abstracted by conditions on ordering between events. The first condition,  $w \geq b$ , is abstracted by the predicate:

$$w \geq b \leftrightarrow \text{never two } t_w \text{ between two } t_b. \quad (1.5)$$

The abstraction of the second condition,  $\lfloor w/b \rfloor \geq r/b$  requires the following definition of the first instant (of the bus)  $\tau_b(n) = \min\{t_b(p) \mid t_b(p) > t_w(n)\}$  where the bus can fetch the  $n$ th writing. The second condition is then restated as the requirement (1.6) that no two successive  $\tau_b$  can occur between two successive  $t_r$ :

$$\lfloor \frac{w}{b} \rfloor \geq \frac{r}{b} \leftrightarrow \text{never two } \tau_b \text{ between two successive } t_r. \quad (1.6)$$

Under the specific conditions (1.5) and (1.6), the correctness of the protocol reduces to the assumption:

$$\forall n \in \mathbb{N}, \exists k \in \mathbb{N}, \text{ s.t. } \tau_b(n) < t_r(k) \leq \tau_b(n+1)$$

It guarantees that all written values are actually fetched by the bus ( $\tau_b(n)$  always exists, and  $\tau_b(n+1) \neq \tau_b(n)$  since there is at least one instant  $t_r(k)$  which occurs in between them), and all fetched values are actually read by the reader ( $\tau_b(n) < t_r(k) \leq \tau_b(n+1)$ ): see Figure 1.2.

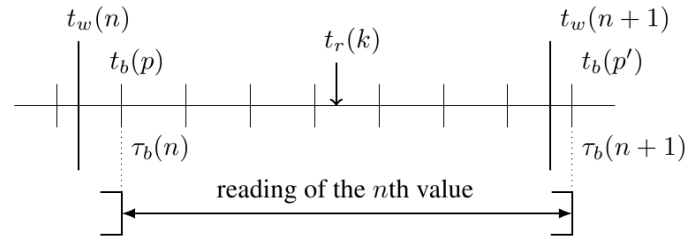


Fig. 1.2: Correctness of the protocol.

### 1.2.2 Introduction to the module language

Considering first the writer and the bus, the protocol will be correct only if  $w \geq b$ , so that the data flow emitted by the bus is equal to the data flow emitted by the writer ( $\forall n \cdot x_b(n) = x_w(n)$ ).

In the module language, a specification is designated by the keyword **contract**. It defines a set of input and output variables (interface) subject to a contract. The interface defines the way the component interacts with its environment through its variables. In addition, it embeds properties that are modeled by a composition of contracts. For instance, the specification of a bus controller could be defined by the assumption  $w \geq b$  and the guarantee  $\forall n \cdot x_b(n) = x_w(n)$ . An implementation of the specification, designated by the keyword **process**, contains a compatible implementation of the above contract.

```

module type WriterBusType =
  contract input  real w, b;
                boolean xw
                output boolean xb;
  assume w >= b
  guarantee xb = xw
end;

module WriterBus : WriterBusType =
  process input  real w, b;
                boolean xw
                output boolean xb;
  (|
   ...
  |)
end;

```

The specification of the properties we consider for the whole LTTA consists of two contracts. Each contract applies to a given component (bus or reader) of the LTTA. It is made of a clock relation as assumption and an equality of flows as guarantee. Instead of specifying two separate contracts, we define them as two instances of a generic one. To this end, we define an encapsulation mechanism to generically represent a class of specifications or implementations sharing a common pattern of behavior up to that of the parameters. In the example of the LTTA, for instance, we have used a functor to parameterize it with respect to its clock relations.

```

module type LTTAProperty =
functor(real c1, c2)
  contract input boolean xwb
  output boolean xbr;
  assume c1 >= c2
  guarantee xbr = xwb
end;

module type LTTAClockConstraints =
contract input real w, b, r;
boolean cw
output boolean xr;
LTTAProperty(w, b) (xw, xb) and
LTTAProperty(floor(w/b), r/b) (xb, xr)
end;

```

The generic contract, called `LTTAProperty`, is parameterized with two clocks. When the clock constraint associated with the context of the considered component (bus or reader) of the LTTA is respected, the preservation of the flows is ensured by this component. The contract of the LTTA is defined by the composition “**and**” of two applications of the generic `LTTAProperty` contract. Each application defines a property of the LTTA with its appropriate clock constraint. The composition defines the *greatest lower-bound* of both contracts. Each application of the `LTTAProperty` functor produces a contract which is composed with the other ones in order to produce the expected contract.

A module is hence viewed as a pair  $M : I$  consisting of an implementation  $M$  that is typed by (or viewed as) a specification  $I$  of satisfiable contract. The semantics of the specification  $I$ , written  $\llbracket I \rrbracket$ , is a set of processes (in the sense of section 1.3) whose traces satisfy the contract associated with  $I$ . The semantics  $\llbracket M \rrbracket$  of the implementation  $M$  is a process contained in  $\llbracket I \rrbracket$ .

### 1.3 An algebra of contracts for assume-guarantee reasoning

Section 1.3.1 introduces a suitably general algebra of processes. A contract  $(\mathbf{A}, \mathbf{G})$  is viewed as a pair of logical devices filtering processes: the assumption  $\mathbf{A}$  filters processes to select (accept or conversely reject) those that are asserted (accepted or conversely rejected) by the guarantee  $\mathbf{G}$ . Process-filters are defined in Section 1.3.2 and contracts in Section 1.3.3. The proofs of properties presented in this Section are provided in [12]. Section 1.3.4 discusses some related approaches for contracts.

#### 1.3.1 An algebra of processes

We start with the definition of a suitable algebra for behaviors and processes. We deliberately choose an abstract definition of behavior as a function from a set of variable names to a domain of typed values. These typed values may be themselves functions of time to some domain of data values: it is the case, for instance, when we consider the SIGNAL language, where a behavior describes the trace of a discrete process.

**Definition 1 (Behavior).** Let  $\mathcal{V}$  be an infinite, countable set of variables, and  $\mathcal{D}$  a set of values; for  $\mathbf{Y}$  a nonempty finite set of variables included in  $\mathcal{V}$ , a  $\mathbf{Y}$ -behavior is a function  $b : \mathbf{Y} \rightarrow \mathcal{D}$ .

The set of  $\mathbf{Y}$ -behaviors is denoted by  $\mathbb{B}_{\mathbf{Y}} = \mathbf{Y} \rightarrow \mathcal{D}$ ;  $\mathbb{B}_{\emptyset} = \emptyset$  denotes the set of behaviors (which is empty) on the empty set of variables. The notation  $c|_{\mathbf{X}}$  is used for the restriction of a  $\mathbf{Y}$ -behavior  $c$  on  $\mathbf{X}$ , a (possibly empty) subset of  $\mathbf{Y}$ .

In Figure 1.3, the  $x, y$ -behavior  $b_1$  is a function from the variables  $x, y$  to a function that denotes signals. Behavior  $b_1$  is a discrete sampling mapping a domain of time represented by natural numbers to values.

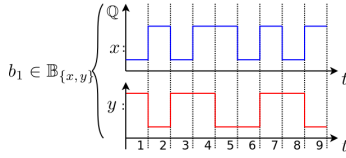


Fig. 1.3: Example of behavior.

A *process* is defined as a set of behaviors on a given set of variables.

**Definition 2 (Process).** For  $\mathbf{X}$  a finite set of variables, an  $\mathbf{X}$ -process  $p$  is a nonempty set of  $\mathbf{X}$ -behaviors.

The unique  $\emptyset$ -process, on the empty set of variables, is denoted by  $\Omega = \{\emptyset\}$ . It can be seen as the universal process; it has no effect when composed with other processes. The *empty process*, which is defined by the empty set of behaviors, is denoted by  $\mathcal{U} = \emptyset$ . It can be seen as the null process; when composed (intersected) with other processes, it always results in the empty process.

The set of  $\mathbf{X}$ -processes is denoted by  $\mathbb{P}_{\mathbf{X}} = \mathcal{P}(\mathbb{B}_{\mathbf{X}}) \setminus \{\mathcal{U}\}$  and  $\mathbb{P}^*_{\mathbf{X}} = \mathbb{P}_{\mathbf{X}} \cup \{\mathcal{U}\}$ . The set of all processes is denoted by  $\mathbb{P} = \cup_{(\mathbf{X} \subset \mathcal{V})} \mathbb{P}_{\mathbf{X}}$  and  $\mathbb{P}^* = \mathbb{P} \cup \{\mathcal{U}\}$ . For an  $\mathbf{X}$ -process  $p$ , the domain  $\mathbf{X}$  of its behaviors is denoted  $var(p)$ , and  $var(\mathcal{U}) = \mathcal{V}$ .

Complement, restriction and extension of a process have expected definitions:

**Definition 3 (Complement, restriction and extension).** For  $\mathbf{X}$  a finite set of variables, the complement  $\tilde{p}$  of a process  $p \in \mathbb{P}_{\mathbf{X}}$  is defined by  $\tilde{p} = (\mathbb{B}_{\mathbf{X}} \setminus p)$ . Also,  $\tilde{\mathcal{U}} = \mathbb{B}_{\mathbf{X}}$ . For  $\mathbf{X}, \mathbf{Y}$ , finite sets of variables such that  $\mathbf{X} \subseteq \mathbf{Y} \subset \mathcal{V}$ ,  $q|_{\mathbf{X}} = \{c|_{\mathbf{X}} \mid c \in q\}$  is the restriction  $q|_{\mathbf{X}} \in \mathbb{P}_{\mathbf{X}}$  of  $q \in \mathbb{P}_{\mathbf{Y}}$  and  $p|_{\mathbf{Y}} = \{c \in \mathbb{B}_{\mathbf{Y}} \mid c|_{\mathbf{X}} \in p\}$  is the extension  $p|_{\mathbf{Y}} \in \mathbb{P}_{\mathbf{Y}}$  of  $p \in \mathbb{P}_{\mathbf{X}}$ . Also,  $\mathcal{U}|_{\mathbf{X}} = \mathcal{U}$  and  $\mathcal{U}|_{\mathcal{V}} = \mathcal{U}$ .

Note that the extension of  $p$  in  $\mathbb{P}_{\mathbf{X}}$  to  $\mathbf{Y} \subset \mathcal{V}$  is the process on  $\mathbf{Y}$  that has the same constraints as  $p$ .

The set  $\mathbb{P}^*_{\mathbf{X}}$ , equipped with union, intersection and complement, is a Boolean algebra with supremum  $\mathbb{P}^*_{\mathbf{X}}$  and infimum  $\mathcal{U}$ .

The extension operator induces a partial order  $\preceq$  on processes, such that  $p \preceq q$  if  $q$  is an extension of  $p$  to the variables of  $q$ ; the relation  $\preceq$ , used to define filters, is studied below.

**Definition 4 (Process extension relation).** The process extension relation  $\preceq$  is defined by:  $(\forall p \in \mathbb{P}) (\forall q \in \mathbb{P}) (p \preceq q) = ((\text{var}(p) \subseteq \text{var}(q)) \wedge (p|_{\text{var}(q)} = q))$

Thus, if  $(p \preceq q)$ ,  $q$  is defined on more variables than  $p$ ; on the variables of  $p$ ,  $q$  has the same constraints as  $p$ ; its other variables are free. This relation extends to  $\mathbb{P}^*$  with  $(\cup \preceq \cup)$ .

*Property 1.*  $(\mathbb{P}^*, \preceq)$  is a poset.

In this poset, the upper set of a process  $p$ , called *extension upper set*, is the set of all its extensions; it is denoted by  $p \uparrow_{\preceq} = \{q \in \mathbb{P} / p \preceq q\}$ .

To study properties of extension upper sets, we characterize semantically the set of variables that are *constrained* by a given process: a process  $q \in \mathbb{P}$  *controls* a given variable  $y$  if  $y$  belongs to  $\text{var}(q)$  and  $q$  is not equal to the extension on  $\text{var}(q)$  of its projection on  $(\text{var}(q) \setminus \{y\})$ . Formally, a process  $q \in \mathbb{P}$  controls a variable  $y$ , written  $(q \triangleright y)$ , iff  $(y \in \text{var}(q))$  and  $q \neq ((q|_{(\text{var}(q) \setminus \{y\})})|_{\text{var}(q)})$ . A process  $q \in \mathbb{P}$  controls a variable set  $\mathbf{X}$ , written  $(q \triangleright \mathbf{X})$ , iff  $(\forall x \in \mathbf{X}) (q \triangleright x)$ . Also,  $\triangleright$  is extended to  $\mathbb{P}^*$  with  $\cup \triangleright \cup$ .

This is illustrated in Figure 1.4 (left): there is some behavior  $b$  in  $q$  that has the same restriction on  $(\text{var}(q) \setminus \{y\})$  as some behavior  $c$  in  $\mathbb{B}_{\text{var}(q)}$  such that  $c$  does not belong to  $q$ ; thus  $q$  is strictly included in  $(q|_{(\text{var}(q) \setminus \{y\})})|_{\text{var}(q)}$ .

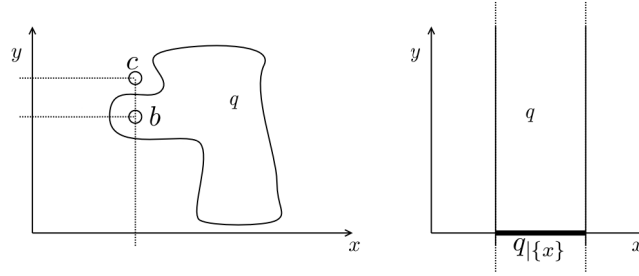


Fig. 1.4: Controlled (left) and non-controlled (right) variable  $y$  in a process  $q$ .

We define a *reduced process* (the key concept to define filters) as being a process that controls all of its variables.

**Definition 5 (Reduced process).** A process  $p \in \mathbb{P}^*$  is *reduced* iff  $p \triangleright \text{var}(p)$ .

Reduced processes are minimal in  $(\mathbb{P}, \preceq)$ . We denote by  $\overset{\nabla}{q}$ , called *reduction of  $q$* , the (minimal) process such that  $\overset{\nabla}{q} \preceq q$  ( $p$  is reduced iff  $\overset{\nabla}{p} = p$ ).

*Property 2.* The complement  $\tilde{p}$  of a nonempty process  $p$  strictly included in  $\mathbb{B}_{\text{var}(p)}$  is reduced iff  $p$  is reduced; then  $\tilde{p}$  and  $p$  control the same set of variables  $\text{var}(p)$ .

The extension upper set  $\overset{\nabla}{p}\uparrow_{\subseteq}$  of the reduction of  $p$  is composed of all the sets of behaviors, defined on variable sets that include the variables controlled by  $p$ , as maximal processes (for union of sets of behaviors) that have exactly the same constraints as  $p$  (variables that are not controlled by  $p$  are also not controlled in the processes of  $\overset{\nabla}{p}\uparrow_{\subseteq}$ ). We also observe that  $\text{var}(\overset{\nabla}{q})$  is the greatest subset of variables such that  $q \triangleright \text{var}(\overset{\nabla}{q})$ .

Then we define the *inclusion lower set* of a set of processes to capture all the subsets of behaviors of these processes. Let  $\mathbf{R} \subseteq \mathbb{P}^*$ ,  $\mathbf{R}\downarrow_{\subseteq}$  is the inclusion lower set of  $\mathbf{R}$  for  $\subseteq$  defined by  $\mathbf{R}\downarrow_{\subseteq} = \{p \in \mathbb{P}^* \mid (\exists q \in \mathbf{R}) (p \subseteq q)\}$ .

### 1.3.2 An algebra of filters

In this section, we define a *process-filter* by the set of processes that satisfy a given property. We define an order relation ( $\sqsubseteq$ ) on the set of process-filters  $\Phi$ . With this relation,  $(\Phi, \sqsubseteq)$  is a Boolean algebra. A *process-filter*  $\mathbf{R}$  is a subset of  $\mathbb{P}^*$  that *filters* processes: it contains all the processes that are “equivalent” with respect to some constraint or property, so that all processes in  $\mathbf{R}$  are accepted or all of them but  $\bar{U}$  are rejected. A process-filter is built from a unique *process generator* by extending it to larger sets of variables and then by including subprocesses of these “maximal allowed behavior sets”.

**Definition 6 (Process-filter).** A set of processes  $\mathbf{R}$  is a *process-filter* iff  $(\exists r \in \mathbb{P}^*) ((r = \overset{\nabla}{r}) \wedge (\mathbf{R} = r\uparrow_{\subseteq}\downarrow_{\subseteq}))$ . The process  $r$  is a *generator* of  $\mathbf{R}$  ( $\mathbf{R}$  is generated by  $r$ ).

The process-filter generated by the reduction of a process  $p$  is denoted by  $\hat{p} = \overset{\nabla}{p}\uparrow_{\subseteq}\downarrow_{\subseteq}$ . The generator of a process-filter  $\mathbf{R}$  is unique, we refer to it as  $\overset{\nabla}{\mathbf{R}}$ . Note that  $\Omega$  generates the set of all processes (including  $\bar{U}$ ) and  $\bar{U}$  belongs to all filters. Formally,  $(\forall p, r, s \in \mathbb{P}^*)$ , we have:

$$(p \in \hat{r}) \implies (\text{var}(\overset{\nabla}{r}) \subseteq \text{var}(p)) \quad \hat{r} = \hat{s} \iff \overset{\nabla}{r} = \overset{\nabla}{s} \quad \Omega \in \hat{r} \iff \hat{r} = \mathbb{P}^*$$

Figure 1.5 illustrates how a process-filter is generated from a process  $p$  (depicted by the bold line) in two successive operations. The first operation consists of building the extension upper set of the process: it takes all the processes that are compatible with  $p$  and that are defined on a larger set of variables. The second operation proceeds using the inclusion lower set of this set of processes: it takes all the processes that are defined by subsets of behaviors from processes in the extension upper set (in other words, those processes that remain compatible when adding constraints, since adding constraints removes behaviors).

We denote by  $\Phi$  the set of process-filters. We call *strict process-filters* the process-filters that are neither  $\mathbb{P}^*$  nor  $\{\bar{U}\}$ . The filtered variable set of a process-filter  $\mathbf{R}$  is  $\text{var}(\mathbf{R})$  defined by  $\text{var}(\mathbf{R}) = \text{var}(\overset{\nabla}{\mathbf{R}})$ .

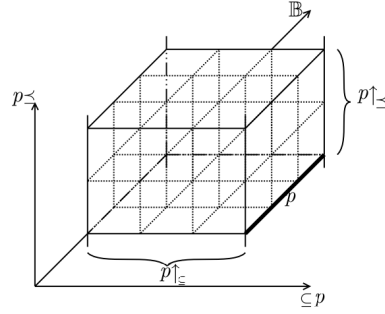


Fig. 1.5: Example of process-filter.

We define an order relation on process-filters, that we call *relaxation*, and write  $\mathbf{R} \sqsubseteq \mathbf{S}$  to mean that  $\mathbf{S}$  is a relaxation of  $\mathbf{R}$ .

**Definition 7 (Process-filter relaxation).** For  $\mathbf{R}$  and  $\mathbf{S}$ , two process-filters, let  $\mathbf{Z} = \text{var}(\mathbf{R}) \cup \text{var}(\mathbf{S})$ . The relation  $\mathbf{S}$  is a relaxation of  $\mathbf{R}$ , written  $\mathbf{R} \sqsubseteq \mathbf{S}$ , is defined by:

$$(\mathbf{R} \sqsubseteq \mathbf{S} \iff \mathbf{R} \stackrel{\nabla|\mathbf{Z}}{\sqsubseteq} \mathbf{S} \stackrel{\nabla|\mathbf{Z}}{\sqsubseteq} \mathbf{S}) \quad \{\mathcal{U}\} \sqsubseteq \mathbf{S} \quad (\mathbf{R} \sqsubseteq \{\mathcal{U}\}) \iff \{\mathcal{U}\} = \mathbf{R}$$

The relaxation relation defines the structure of process-filters, which is shown to be a lattice.

**Lemma 1.**  $(\Phi, \sqsubseteq)$  is a lattice of supremum  $\mathbb{P}^*$  and infimum  $\{\mathcal{U}\}$ . Let  $\mathbf{R}$  and  $\mathbf{S}$  be two process-filters,  $\mathbf{V} = \text{var}(\mathbf{R}) \cup \text{var}(\mathbf{S})$ ,  $\mathbf{R}_{\mathbf{V}} = \mathbf{R} \stackrel{\nabla|\mathbf{V}}{\sqsubseteq} \mathbf{R}$  and  $\mathbf{S}_{\mathbf{V}} = \mathbf{S} \stackrel{\nabla|\mathbf{V}}{\sqsubseteq} \mathbf{S}$ . Conjunction  $\mathbf{R} \sqcap \mathbf{S}$ , disjunction  $\mathbf{R} \sqcup \mathbf{S}$  and complement  $\widetilde{\mathbf{R}}$  are respectively defined by:

$$\begin{aligned} \mathbf{R} \sqcap \mathbf{S} &= \overbrace{(\mathbf{R}_{\mathbf{V}} \cap \mathbf{S}_{\mathbf{V}})}^{\nabla} \uparrow_{\sqsubseteq} \downarrow_{\sqsubseteq} & \{\mathcal{U}\} \sqcap \mathbf{R} &= \{\mathcal{U}\} \\ \mathbf{R} \sqcup \mathbf{S} &= \overbrace{(\mathbf{R}_{\mathbf{V}} \cup \mathbf{S}_{\mathbf{V}})}^{\nabla} \uparrow_{\sqsubseteq} \downarrow_{\sqsubseteq} & \{\mathcal{U}\} \sqcup \mathbf{R} &= \mathbf{R} \\ \widetilde{\mathbf{R}} &= \widetilde{\mathbf{R}} \uparrow_{\sqsubseteq} \downarrow_{\sqsubseteq} & \widetilde{\{\mathcal{U}\}} &= \mathbb{P}^* & \widetilde{\mathbb{P}^*} &= \{\mathcal{U}\} \end{aligned}$$

Let us comment the definitions of these operators. Conjunction of two strict process-filters  $\mathbf{R}$  and  $\mathbf{S}$ , for instance, is obtained by first building the extension of the generators  $\mathbf{R}$  and  $\mathbf{S}$  on the union of the sets of their controlled variables; then the intersection of these processes, which is also a process (set of behaviors) is considered; since this operation may result in some variables becoming free (not controlled), the reduction of this process is taken; and finally, the result is the process-filter generated by this reduction. The process-filter conjunction  $\mathbf{R} \sqcap \mathbf{S}$  of two strict process-filters  $\mathbf{R}$  and  $\mathbf{S}$  is the greatest process-filter  $\mathbf{T} = \mathbf{R} \sqcap \mathbf{S}$  that accepts all processes that are accepted by  $\mathbf{R}$  and by  $\mathbf{S}$ . The same mechanism, with union, is used to define disjunction. The process-filter disjunction  $\mathbf{R} \sqcup \mathbf{S}$  of two strict process-filters  $\mathbf{R}$  and  $\mathbf{S}$  is the smallest process-filter  $\mathbf{T} = \mathbf{R} \sqcup \mathbf{S}$  that accepts all processes that are accepted

by  $\mathbf{R}$  or by  $\mathbf{S}$ . And the complement of a strict process-filter  $\mathbf{R}$  is the process-filter generated by the complement of its generator  $\mathbf{R}$ .

Finally, we state a main result for process-filters, which is that process-filters form a Boolean algebra:

**Theorem 1.**  $(\Phi, \sqsubseteq)$  is a Boolean algebra with  $\mathbb{P}^*$  as 1,  $\{\emptyset\}$  as 0 and the complement  $\tilde{\mathbf{R}}$ .

### 1.3.3 An algebra of contracts

From process-filters, we define the notion of assume/guarantee contract and propose a refinement relation on contracts.

**Definition 8 (Contract).** A contract  $\mathbf{C} = (\mathbf{A}, \mathbf{G})$  is a pair of process-filters. The variable set of  $\mathbf{C} = (\mathbf{A}, \mathbf{G})$  is defined by  $var(\mathbf{C}) = var(\mathbf{A}) \cup var(\mathbf{G})$ .  $\mathcal{C} = \Phi \times \Phi$  is the set of contracts.

Usually, an assumption  $\mathbf{A}$  is an assertion on the behavior of the environment (it is typically expressed on the inputs of a process) and thus defines the set of behaviors that the process has to take into account. The guarantee  $\mathbf{G}$  defines properties that should be guaranteed by a process running in an environment where behaviors satisfy  $\mathbf{A}$ .

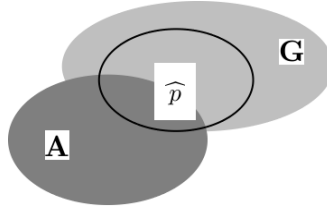


Fig. 1.6: A process  $p$  satisfying a contract  $(\mathbf{A}, \mathbf{G})$ .

A process  $p$  satisfies a contract  $\mathbf{C} = (\mathbf{A}, \mathbf{G})$  if all its behaviors that are accepted by  $\mathbf{A}$  (i.e., that are behaviors of some process in  $\mathbf{A}$ ), are also accepted by  $\mathbf{G}$ . Figure 1.6 depicts a process  $p$  satisfying the contract  $(\mathbf{A}, \mathbf{G})$  ( $\hat{p}$  is the process-filter generated by the reduction of  $p$ ). This is made more precise and formal by the following definition.

**Definition 9 (Satisfaction).** Let  $\mathbf{C} = (\mathbf{A}, \mathbf{G})$  a contract; a process  $p$  satisfies  $\mathbf{C}$ , written  $p \models \mathbf{C}$ , iff  $(\hat{p} \sqcap \mathbf{A}) \sqsubseteq \mathbf{G}$ .

*Property 3.*  $p \models \mathbf{C} \iff \hat{p} \sqsubseteq (\tilde{\mathbf{A}} \sqcup \mathbf{G})$

We define a preorder relation that allows to compare contracts. A contract  $(\mathbf{A}_1, \mathbf{G}_1)$  is *finer* than a contract  $(\mathbf{A}_2, \mathbf{G}_2)$ , written  $(\mathbf{A}_1, \mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2, \mathbf{G}_2)$ , iff all processes that satisfy the contract  $(\mathbf{A}_1, \mathbf{G}_1)$  also satisfy the contract  $(\mathbf{A}_2, \mathbf{G}_2)$ :

**Definition 10 (Satisfaction preorder).**  $(\mathbf{A}_1, \mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2, \mathbf{G}_2)$  iff  $(\forall p \in \mathbb{P})(p \models (\mathbf{A}_1, \mathbf{G}_1) \implies (p \models (\mathbf{A}_2, \mathbf{G}_2)))$

The preorder on contracts satisfies the following property:

*Property 4.*  $(\mathbf{A}_1, \mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2, \mathbf{G}_2)$  iff  $(\widetilde{\mathbf{A}}_1 \sqcup \mathbf{G}_1 \sqsubseteq (\widetilde{\mathbf{A}}_2 \sqcup \mathbf{G}_2))$

*Refinement of contracts* is further defined from the satisfaction preorder:

**Definition 11 (Refinement of contracts).** A contract  $\mathbf{C}_1 = (\mathbf{A}_1, \mathbf{G}_1)$  *refines* a contract  $\mathbf{C}_2 = (\mathbf{A}_2, \mathbf{G}_2)$ , written  $\mathbf{C}_1 \preceq \mathbf{C}_2$ , iff  $(\mathbf{A}_1, \mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2, \mathbf{G}_2)$ ,  $(\mathbf{A}_2 \sqsubseteq \mathbf{A}_1)$  and  $\mathbf{G}_1 \sqsubseteq (\mathbf{A}_1 \sqcup \mathbf{G}_2)$ .

Refinement of contracts amounts to relaxing assumptions and reinforcing promises under the initial assumptions. The intuitive meaning is that for any  $p$  that satisfies a contract  $\mathbf{C}$ , if  $\mathbf{C}$  refines  $\mathbf{D}$  then  $p$  satisfies  $\mathbf{D}$ . Our relation of refinement formalizes substitutability. Among contracts that could be used to refine an existing contract  $(\mathbf{A}_2, \mathbf{G}_2)$ , we choose those contracts  $(\mathbf{A}_1, \mathbf{G}_1)$  that “scan” more processes than  $(\mathbf{A}_2, \mathbf{G}_2)$  ( $\mathbf{A}_2 \sqsubseteq \mathbf{A}_1$ ) and that guarantee less processes than those of  $\mathbf{A}_1 \sqcup \mathbf{G}_2$ .

The refinement relation can be expressed as follows in the algebra of process-filters:

*Property 5.*  $(\mathbf{A}_1, \mathbf{G}_1) \preceq (\mathbf{A}_2, \mathbf{G}_2)$  iff  $\mathbf{A}_2 \sqsubseteq \mathbf{A}_1$ ,  $(\mathbf{A}_2 \sqcap \mathbf{G}_1) \sqsubseteq \mathbf{G}_2$  and  $\mathbf{G}_1 \sqsubseteq (\mathbf{A}_1 \sqcup \mathbf{G}_2)$ .

The refinement relation ( $\preceq$ ) defines the poset of contracts, which is shown to be a lattice. In this lattice, the union (or disjunction) of contracts is defined by their least upper bound and the intersection (or conjunction) of contracts is defined by their greatest lower bound. These operations provide two *compositions of contracts*.

**Lemma 2 (Composition of contracts).** *Two contracts  $\mathbf{C}_1 = (\mathbf{A}_1, \mathbf{G}_1)$  and  $\mathbf{C}_2 = (\mathbf{A}_2, \mathbf{G}_2)$  have a greatest lower bound  $\mathbf{C} = (\mathbf{A}, \mathbf{G})$ , written  $(\mathbf{C}_1 \downarrow \mathbf{C}_2)$ , defined by:*

$$\mathbf{A} = \mathbf{A}_1 \sqcup \mathbf{A}_2 \text{ and } \mathbf{G} = ((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}}_2 \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2))$$

*and a least upper bound  $\mathbf{D} = (\mathbf{B}, \mathbf{H})$ , written  $(\mathbf{C}_1 \uparrow \mathbf{C}_2)$ , defined by:*

$$\mathbf{B} = \mathbf{A}_1 \sqcap \mathbf{A}_2 \text{ and } \mathbf{H} = (\widetilde{\mathbf{A}}_1 \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_2 \sqcap \mathbf{G}_1)$$

A Heyting algebra  $H$  is a bounded lattice such that for all  $a$  and  $b$  in  $H$  there is a greatest element  $x$  of  $H$  such that the greatest lower bound of  $a$  and  $x$  is smaller than  $b$  [5]. For all contracts  $\mathbf{C}_1 = (\mathbf{A}_1, \mathbf{G}_1)$ ,  $\mathbf{C}_2 = (\mathbf{A}_2, \mathbf{G}_2)$ , there is a greatest element  $\mathbf{X}$  of  $\mathbb{C}$  such that the greatest lower bound of  $\mathbf{C}_1$  and  $\mathbf{X}$  refines  $\mathbf{C}_2$ . Then our contract algebra is a Heyting algebra (in particular, it is distributive):

**Theorem 2.**  $(\mathbb{C}, \preceq)$  is a Heyting algebra with supremum  $(\{\cup\}, \mathbb{P}^*)$  and infimum  $(\mathbb{P}^*, \{\cup\})$ .

Note that it is not a Boolean algebra since it is not possible to define in general a complement for each contract. The complement exists only for contracts of the form  $(\mathbf{A}, \widetilde{\mathbf{A}})$  and it is then equal to  $(\widetilde{\mathbf{A}}, \mathbf{A})$ .

### 1.3.4 Contracts: some related approaches

The use of contracts has been advocated for a long time in computer science [1, 16] and, more recently, has been successfully applied in object-oriented software engineering [25]. In object-oriented programming, the basic idea of design-by-contract is to consider the services provided by a class as a contract between the class and its caller. The contract is composed of two parts: requirements made by the class upon its caller and promises made by the class to its caller.

In the theory of interface automata [2], the notion of interface offers benefits similar to our notion of contract and for the purpose of checking interface compatibility between reactive modules. In that context, it is irrelevant to separate the assumptions from guarantees and only one contract needs to be and is associated with a module. Separation and multiple views become of importance in a more general-purpose software engineering context. Separation allows more flexibility in finding (contra-variant) compatibility relations between components. Multiple views allow better isolation between modules and hence favor compositionality. This is discussed in Section 1.5.3. In our contract algebra as in interface automata, a contract can be expressed with only one filter. To this end, a filtering equivalence relation [12] (that defines the equivalence class of contracts that accept the same set of processes) may be used to express a contract with only one guarantee filter and with its hypothesis filter accepting all the processes (or, conversely, with only one hypothesis filter and a guarantee filter that accepts no process).

In [6], a system of assume-guarantee contracts with similar aims of genericity is proposed. By contrast to our domain-theoretical approach, the EC Speeds project considers an automata-based approach, which is indeed dual but makes notions such as the complement of a contract more difficult to express from within the model. The proposed approach also leaves the role of variables in contracts unspecified, at the cost of some algebraic relations such as inclusion.

In [18], the authors show that the framework of interface automata may be embedded into that of modal I/O automata. This approach is further developed in [27], where modal specifications are considered. This consists of labelling transitions that *may* be fired and other that *must*. Modal specifications are equipped with a parallel composition operator and refinement order which induces a greatest lower bound. The glb allows addressing multiple-viewpoint and conjunctive requirements. With the experience of [6], the authors notice the difficulty in handling interfaces having different alphabets. Thanks to modalities, they propose different alphabet equalizations depending on whether parallel composition or conjunction is considered. Then they consider contracts as residuations  $G/A$  (the residuation is the adjoint of parallel composition), where assumptions  $A$  and guarantees  $G$  are both specified as modal specifications. The objectives of this approach are quite close to ours. Our model deals carefully with alphabet equalization. Moreover, using synchronous composition for processes and greatest lower bound for process-filters and for contracts, our model captures both substitutability and multiple-viewpoint (see Section 1.5.3).

In [22], a notion of synchronous contracts is proposed for the programming language LUSTRE. In this approach, contracts are executable specifications (syn-

chronous observers) timely paced by a clock (the clock of the environment). This yields an approach which is satisfactory to verify safety properties of individual modules (which have a clock) but can hardly scale to the modeling of globally asynchronous architectures (which have multiple clocks).

In [9], a compositional notion of refinement is proposed for a simpler stream-processing data-flow language. By contrast to our algebra, which encompasses the expression of temporal properties, it is limited to reasoning on input-output types and input-output causality graph.

The system Jass [4] is somewhat closer to our motivations and solution. It proposes a notion of trace, and a language to talk about traces. However, it seems that it evolves mainly towards debugging and defensive code generation. For embedded systems, we prefer to use contracts for validating composition and hope to use formal tools once we have a dedicated language for contracts. Like in JML [21], the notion of agent with inputs/outputs does not exist in Jass, the language is based on class invariants, and pre/post-conditions associated with methods.

Our main contribution is to define a complete domain-theoretical framework for assume-guarantee reasoning. Starting from a domain-theoretical characterization of behaviors and processes, we build a Boolean algebra of process-filters and a Heyting algebra of contracts. This yields a rich structure which is: 1/ generic, in that it can be implemented or instantiated to specific models of computation; 2/ flexible, in the way it can help structuring and normalizing expressions; 3/ complete, in the sense that all combinations of propositions can be expressed *within* the model.

Finally, a temporal logic that is consistent with our model, such as for instance ATL (Alternating-time Temporal Logic [3]), can directly be used to express assumptions about the context of a process and guarantees provided by that process.

## 1.4 A module language for typing by contracts

In this section, we define a module language to implement our contract algebra and apply it to the validation of component-based systems. For the peculiarity of our applications, it will be instantiated to the context of the synchronous language SIGNAL, yet it could equally be used in the context of related programming languages manipulating processes or agents. Its basic principle is to separate the interface, which declares properties of a program using contracts, and implementation, which defines an executable specification satisfying it.

### 1.4.1 Syntax

We define the formal syntax of our module language. Its grammar is parameterized by the syntax of programs, noted  $p$  or  $q$ , which belong to the target specification or programming language. Names are noted  $x$  or  $y$ . Types  $t$  are used to declare

parameters and variables in the interface of contracts. Assumptions and guarantees are described by expressions  $p$  and  $q$  of the target language. An expression  $exp$  manipulates contracts and modules to parameterize, apply, reference and compose them.

$x, y$	(name)
$p, q$	(process)
$b, c ::= \mathbf{event} \mid \mathbf{boolean} \mid \mathbf{short} \mid \mathbf{integer} \mid \dots$	(datatype)
$t ::= b \mid \mathbf{input} \ b \mid \mathbf{output} \ b \mid x \mid t \times t$	(type)
$dec ::= t \ x \ [, \ dec]$	(declaration)
$def ::= \mathbf{module} \ [\mathbf{type}] \ x = exp$	(definition)
$\mathbf{module} \ x \ [: \ t] = exp$	
$def; def$	
$ag ::= [\mathbf{assume} \ p] \ \mathbf{guarantee} \ q;$	(contract)
$ag \ \mathbf{and} \ ag \mid x(y^*)$	(process)
$exp ::= \mathbf{contract} \ dec; \ ag \ \mathbf{end}$	(contract)
$\mathbf{process} \ dec; \ p \ \mathbf{end}$	(process)
$\mathbf{functor} \ (dec) \ exp$	(functor)
$exp \ \mathbf{and} \ exp$	(composition)
$x \ (exp^*)$	(application)
$\mathbf{let} \ def \ \mathbf{in} \ exp$	(scoping)

### 1.4.2 A type system for contracts and processes

We define a type system for contracts and processes in the module language. In the syntax of the module language, contracts and processes are associated with names  $x$ . These names can be used to type formal parameters in a functor and become type declarations. Hence, in the type system, type names that stand for a contract or a process are associated with a module type  $T$ . A base module type is a tagged pair  $\tau(I, C)$ . The tag  $\tau$  is noted  $\pi$  for the type of a process and  $\gamma$  for the type of a contract. The set  $I$  consists of pairs  $x : t$  that declare the types  $t$  for its input and output variables  $x$ . The contract  $C$  is a pair of predicates  $(p, q)$  that represent its assumptions  $p$  and guarantees  $q$ . The type of a functor  $\Lambda(x : S).T$  consists of the name  $x$  and of the types  $S$  and  $T$  of its formal parameter and result.

$$S, T ::= t \mid \tau(I, C) \mid S \times T \mid \Lambda(x : S).T \quad (\text{type}) \quad \tau ::= \gamma \mid \pi \quad (\text{kind})$$

### 1.4.3 Subtyping as refinement

We define a subtyping relation on types  $t$  to extend the refinement relation of the contract algebra to the type algebra. In that aim, we apply the subtyping principle  $S \leq T$  ( $S$  is a subtype of  $T$ ) to mean that the semantic objects denoted by  $S$  are

contained in the semantic objects denoted by  $T$  ( $S$  refines  $T$ ). Hence, a module of type  $T$  can safely be replaced or substituted by a module of type  $S$ . Figure 1.7 depicts a process  $P$  with one long input  $x$  and two short outputs  $a, b$ , and a process  $Q$  with two integer inputs  $x, y$  and one integer output  $a$ , such that  $P$  refines  $Q$ . Then the type of a module  $M$  encapsulating  $P$  is a subtype of a module  $N$  encapsulating  $Q$ , thus  $M$  can replace  $N$ .

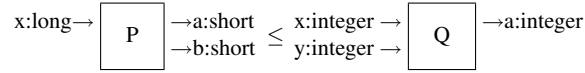


Fig. 1.7: Example of module refinement.

The subtyping relation  $\leq$  is defined inductively with axioms for datatypes, rules for declarations and rules for each kind of module type. The complete rules are described in [14]. In particular, a module type  $S = \tau(I, C)$  is a subtype of  $T = \tau(J, D)$ , written  $S \leq T$ , iff the inputs in  $J$  subtype those in  $I$ , the outputs in  $I$  subtype those in  $J$ , and the contract  $C$  refines  $D$  (written  $C \preceq D$ ).

We can interpret the relation  $C \preceq D$  as a mean to register the refinement constraint between  $C$  and  $D$  in the typing constraints. It corresponds to a proof obligation in the target language, whose meaning is defined by the semantic relation  $\llbracket C \rrbracket \preceq \llbracket D \rrbracket$  in the contract algebra, and whose validity may for instance be proved by model checking (the decidability of the subtyping relation essentially reduces to that of the refinement of contracts).

#### 1.4.4 Composition of modules

Just as the subtyping relation, which implements and extends the refinement relation of the contract algebra in the typing algebra, the operations that define the greatest lower bound (glb) and least upper bound (lub) of two contracts are extended to module types by induction on the structure of types.

Greatest lower bound:

- modules:  $\tau(I, C) \sqcap \tau(J, D) = \tau(I \sqcap J, C \downarrow D)$
- products:  $S \times T \sqcap U \times V = (S \sqcap U) \times (T \sqcap V)$
- functors:  $\Lambda(x : S).T \sqcap \Lambda(y : U).V = \Lambda(x : (S \sqcap U)).(T \sqcap V[y/x])$

Least upper bound:

- modules:  $\tau(I, C) \sqcup \tau(J, D) = \tau(I \sqcup J, C \uparrow D)$
- products:  $S \times T \sqcup U \times V = (S \sqcup U) \times (T \sqcup V)$
- functors:  $\Lambda(x : S).T \sqcup \Lambda(y : U).V = \Lambda(x : (S \sqcup U)).(T \sqcup V[y/x])$

Note that the intersection and union operators have to be extended to combine the set of input and output declarations of a module. For the greatest lower bound, for instance, the resulting set of input variables is obtained by intersection of the input sets, and the type of an input variable is defined as the lub of the considered corresponding types. For the lower bound again, the resulting set of output variables is obtained by the union of the output sets, and for common output variables, their type is defined as the glb of the considered corresponding types. Similar rules are defined for the least upper bound (union of sets for inputs with glb of types, intersection of sets for outputs with lub of types).

The composition of modules is made available in the module language through the “**and**” operator. The operands of this operator can be contracts, in that case, the resulting type is the greatest lower bound  $\Downarrow$  of both contracts. Or they can be expressions of modules, in that case the resulting type is the greatest lower bound  $\sqcap$  of both module types.

## 1.5 Application to SIGNAL

We illustrate the distinctive features of our contract algebra by reconsidering the specification of the LTTA and its translation into observers in the target language of our choice: the multi-clocked synchronous (or polychronous) data-flow language SIGNAL [19, 20].

### 1.5.1 Implementation of the LTTA

We model the LTTA protocol in SIGNAL by specifying the abstraction of all functionalities that write and read values on the bus. Refer to [8] for a description of the operators of the SIGNAL language. In particular, the `cell` operator `y := x cell b init y0` allows to memorize in `y` the latest value carried by `x` when `x` is present or when `b` is *true*. It is defined as follows:

```
(| y := x default (x$1 init y0) | y ^= x ^+ (when b) |)
```

We consider first the following basic functionalities:

```
process current =
  { boolean v0; }
  ( ? boolean wx; event c; ! boolean rx; )
  (| rx := (wx cell c init v0) when c |);
```

```
process interleave =
  ( ? boolean x, sx; )
  (| b := not (b$1 init false)
   | x ^= when b
```

```

    | sx ^= when (not b) |)
  where boolean b; end;

```

The process `current` defines a cell in which values are stored at the input clock  $\wedge w_x$  and loaded on  $r_x$  at the output clock  $c$  (the parameter  $v_0$  is used as initial value). The other functionality is the process `interleave`, that desynchronizes the signals  $x$  and  $s_x$  by synchronizing them to the true and false values of an alternating Boolean signal  $b$ .

A simple buffer can be defined from these functionalities:

```

process shift_1 =
  ( ? boolean x; ! boolean sx; )
  (| interleave(x, sx)
   | sx := current{false}(x, ^sx) |);

```

It represents a one-place FIFO, the contents of which is the last value written into it. Thanks to the `interleave`, the output (signal  $s_x$ ) may only be read/retrieved *strictly after* it was entered. Also, there is no possible loss nor duplication of data.

For the purpose of the LTTA, a couple of signals have to be memorized together: the value to be transmitted, and its associated Boolean flag. So we define the process `shift_2`, in which both values are memorized at some common clock:

```

process shift_2 =
  ( ? boolean x, b ! boolean sx, sb; )
  (| interleave(x, sx)
   | sx := current{false}(x, ^sb)
   | sb := current{true}(b, ^sb)
   |);

```

The shift processes ensure there is necessarily some delay between the input of a data and its output. But for a more general buffer, some data may be lost if a new one is entered and memorized values have to be sustained. Using the `shift_2` (for the LTTA), we may write:

```

process buffer =
  ( ? boolean x, b; event c ! boolean bx, bb, sb; )
  (| (sx, sb) := shift_2(x, b)
   | bx := current{false}(sx, c)
   | bb := current{true}(sb, c)
   |) where boolean sx; end;

```

The signal  $c$  provides the clock at which data are retrieved from the buffer. The clock of the output signal  $sb$  (which is the clock resulting from the internal `shift`) represents the clock of the first instants at which the buffer can fetch a new value. It will be used to express some assumptions on the protocol.

Then the process `ltta` is decomposed into a reader, a bus and a writer:

```

process ltta =
  ( ? boolean xw; event cw, cb, cr; ! boolean xr; )

```

```
(| (xb, bb, sbw) := bus(xw, writer(xw, cw), cb)
  | (xr, br, sbb) := reader(xb, bb, cr)
  |) where boolean bw, xb, bb, sbw, br, sbb; end;
```

Using the buffer process, the components have the following definition:

```
process writer =
  ( ? boolean xw; event cw; ! boolean bw; )
  (| bw ^= xw ^= cw
  | bw := not(bw$1 init true)
  |);
process bus =
  ( ? boolean xw, bw; event cb;
  ! boolean xb, bb, sbw; )
  (| (xb, bb, sbw) := buffer(xw, bw, cb) |);
process reader =
  ( ? boolean xb, bb; event cr;
  ! boolean xr, br, sbb; )
  (| (yr, br, sbb) := buffer(xb, bb, cr)
  | xr := yr when switched(br)
  |) where boolean yr; end;
```

The switched basic functionality allows to consider the values for which the Boolean flag has alternated:

```
process switched =
  ( ? boolean b; ! boolean c; )
  (| zb := b$1 init true
  | c := (b and not zb) or (not b and zb)
  |) where boolean zb; end;
```

### 1.5.2 Contracts in SIGNAL

In this section, we will specify assumptions and guarantees as SIGNAL processes representing generators of corresponding process-filters.

The behavior of the LTTA is correct if the data flow extracted by the reader is equal to the data flow emitted by the writer ( $\forall n \cdot x_r(n) = x_w(n)$ ). It is the case if the following conditions hold:

$$w \geq b \text{ and } \left\lfloor \frac{w}{b} \right\rfloor \geq \frac{r}{b},$$

We will consider this property as a contract to be satisfied by a given implementation of the protocol. Here, we use again the SIGNAL language to specify this contract with the help of clock constraints or of signals used as *observers* [15]. In general, the generic structure of observers specified in contracts will find a direct



This contract can be used as a type for specifying a LTТА protocol. A possible implementation of this protocol is the one described in Section 1.5.1 using the SIGNAL language:

```

module type spec_LTТА =
  contract input  boolean xw;
               event  cw,
               cb, cr
               output boolean xr,
               sbw, sbb;
  assume
    (| cb ^ = sbw ^ + cb
     | cr ^ = cr ^ +
     (when switched(sbb))
    |)
  where
    process switched = ...
  end;
  guarantee
    (| xok := fifo_3(xw)
     | obs := xr = xok
     |)
    where boolean xok;
    process fifo_3 = ...
  end;
end;

module ltta : spec_LTТА =
  process input  boolean xw;
               event  cw, cb, cr
               output boolean xr,
               sbw, sbb;
  (| (xb, bb, sbw) :=
    bus(xw, writer(xw, cw), cb)
   | (xr, br, sbb) :=
    reader(xb, bb, cr)
   |)
  where boolean bw, xb, bb, br;
  process writer =
    ( ? boolean xw; event cw;
      ! boolean bw; )
    (| ...
     |);
  process bus =
    ( ? boolean xw, bw; event cb;
      ! boolean xb, bb, sbw; )
    (| ...
     |);
  process reader =
    ( ? boolean xb, bb; event cr;
      ! boolean xr, br, sbb; )
    (| ...
     |);
  end;
end;

```

It is needless to say that a sophisticated solver, based for instance on Pressburger arithmetics, shall help us to verify the consistency of the LTТА property. Nonetheless, an implementation of the above LTТА specification, for the purpose of simulation, can straightforwardly be derived. As a by-product, it defines an observer which may be used as a proof obligation against an effective implementation of the LTТА controller to verify that it implements the expected LTТА property. Alternatively, it may be used as a medium to synthesize a controller enforcing the satisfaction of the specified property on the implementation of the model.

### *1.5.3 Salient properties of contracts in the synchronous context*

In the context of component-based or contract-based engineering, refinement and substitutability are recognized as being fundamental requirements [10]. Refinement allows one to replace a component by a finer version of it. Substitutability allows one to implement every contract independently of its context of use. These prop-

erties are essential for considering an implementation as a succession of steps of refinement, until final implementation. As noticed in [27], other aspects might be considered in a design methodology. In particular, shared implementation for different specifications, multiple viewpoints and conjunctive requirements for a given component.

Considering the synchronous composition of SIGNAL processes, the satisfaction relation of contracts and the greatest lower bound as a composition operator for contracts, we have the following properties:

*Property 6.* Let two processes  $p, q \in \mathbb{P}$ , and contracts  $C_1, C_2, C'_1, C'_2 \in \mathbb{C}$ .

- (1)  $C_1 \preceq C_2 \implies ((p \models C_1) \implies (p \models C_2))$
- (2)  $C_1 \rightsquigarrow C_2 \iff ((p \models C_1) \implies (p \models C_2))$
- (3)  $((C'_1 \preceq C_1) \wedge (C'_2 \preceq C_2)) \implies ((C'_1 \Downarrow C'_2) \preceq (C_1 \Downarrow C_2))$
- (4)  $((p \models C_1) \wedge (q \models C_2)) \implies ((p|q) \models (C_1 \Downarrow C_2))$
- (5)  $((p \models C_1) \wedge (p \models C_2)) \iff (p \models (C_1 \Downarrow C_2))$

(1) and (2) relate to refinement and implementation; (3) and (4) allow for substitutability in composition; (5) addresses multiple viewpoints.

- (1) and (2) illustrate the substitutability induced by the refinement relation. For relation (1), if a contract  $C_1$  refines a contract  $C_2$  then a process  $p$  which satisfies  $C_1$  also satisfies  $C_2$ . Consequently, the set of processes satisfying  $C_1$  being included in the set of processes satisfying  $C_2$ , a component which satisfies  $C_2$  can be replaced by a component which satisfies  $C_1$ . For relation (2), a contract  $C_1$  is finer than a contract  $C_2$  if and only if the processes which satisfy  $C_1$  also satisfy  $C_2$ .
- (3) and (4) illustrate the substitutability in composition. For relation (3), if a contract  $C'_1$  refines a contract  $C_1$  and a contract  $C'_2$  refines a contract  $C_2$ , then the greatest lower bound of  $C'_1$  and  $C'_2$  refines the greatest lower bound of  $C_1$  and  $C_2$ . Relation (4) expresses that a subsystem can be developed in isolation. Then, when developed independently, subsystems can be substituted to their specifications and composed as expected. If a SIGNAL process  $p$  satisfies a contract  $C_1$  and a SIGNAL process  $q$  satisfies a contract  $C_2$ , then the synchronous composition of  $p$  and  $q$  satisfies the greatest lower bound of  $C_1$  and  $C_2$ . Thus, each subsystem of a component can be analyzed and designed with its specific frameworks and tools. Finally, the composition of the subsystems satisfies the specification of the component. Property (4) could be illustrated as follows on the LTTA example: define the implementation of the LTTA as a functor parameterized by two components, `bus` and `reader`, respectively associated with the types (i.e., contracts) `busType` and `readerType`, such that the greatest lower bound of `busType` and `readerType` is equal to the type `spec_LTTA` associated with the LTTA implementation.
- (5) illustrates the notion of multiple viewpoints: a process  $p$  satisfies a contract  $C_1$  and a contract  $C_2$  if and only if  $p$  satisfies the greatest lower bound of contracts  $C_1$  and  $C_2$ . This property is a solution for the need for modularity coming

from the concurrent development of systems by different teams using different frameworks and tools. An example is the concurrent handling of the safety or reliability aspects and the functional aspect of a system. Other aspects may have to be considered too. Each of these aspects requires specific frameworks and tools for its analysis and design. Yet, they are not totally independent but rather interact. The issue of dealing with multiple aspects or multiple viewpoints is thus essential.

### 1.5.4 Implementation

The module system described in this paper, embedding data-flow equations defined in SIGNAL, has been implemented in OCaml. It produces a proof tree that consists of 1/ an elaborated SIGNAL program, that hierarchically renders the structure of the system described in the original module expressions, 2/ a static type assignment, that is sound and complete with respect to the module type inference system, 3/ a proof obligation consisting of refinement constraints, that are compiled as an observer or a temporal property in SIGNAL.

The property is then passed on to SIGNAL's model-checker, Sigali [24], which allows to prove or disprove that it is satisfied by the generated program. Satisfaction implies that the type assignment and produced SIGNAL program are correct with the initially intended specification. The generated property may however be used for other purposes. One is to use the controller synthesis services of Sigali [23] to automatically generate a SIGNAL program that enforces the property on the generated program. Another, in the case of infinite state system (e.g. on numbers) would be to generate defensive simulation code in order to produce a trace if the property is violated.

## 1.6 Conclusion

Starting from an abstract characterization of behaviors as functions from variables to a domain of values (Booleans, integers, series, sets of tagged values, continuous functions), we introduced the notion of process-filter to formally characterize the logical device that filters behaviors from processes much like the assumption and guarantee of a contract do. In our model, a process  $p$  fulfils its requirements (or satisfies a contract)  $(\mathbf{A}, \mathbf{G})$  if either it is rejected by  $\mathbf{A}$  (i.e., if  $\mathbf{A}$  represents assumptions on the environment, they are not satisfied for  $p$ ) or it is accepted by  $\mathbf{G}$ . The structure of process-filters is a Boolean algebra and the structure of contracts is a Heyting algebra. These rich structures allow for reasoning on contracts with great flexibility to abstract, refine and combine them. In addition to that, the negation of a contract can formally be expressed from within the model. Moreover, contracts are not limited to expressing safety properties, as is the case in most related frameworks, but

encompass the expression of liveness properties [13]. This is all again due to the central notion of process-filter. Our model deals with constraints or properties possibly expressed on different sets of variables, and takes into account variable equalization when combining them. In this model, assumption and guarantee properties are not necessarily restricted to be expressed as formulas in some logic, but are rather considered as sets of behaviors (generator of process-filter). Note that such a process can represent a constraint expressed in some temporal logic. We introduced a module system based on the paradigm of contract for a synchronous multi-clocked formalism, SIGNAL, and applied it to the specification of a component-based design process. The paradigm we are putting forward is to regard a contract as the behavioral type of a component and to use it for the elaboration of the functional architecture of a system together with a proof obligation that validates the correctness of assumptions and guarantees made while constructing that architecture.

## References

1. Abadi, M., Lamport, L.: Composing specifications. *ACM Trans. Program. Lang. Syst.* **15**(1), 73–132 (1993)
2. de Alfaro, L., Henzinger, T.A.: Interface automata. *SIGSOFT Softw. Eng. Notes* **26**(5), 109–120 (2001)
3. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. *Journal of the ACM* **49**(5), 672–713 (2002)
4. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass - Java with assertions. *Electronic Notes in Theoretical Computer Science* **55**(2), 1–15 (2001)
5. Bell, J.L.: Boolean algebras and distributive lattices treated constructively. *Math. Logic Quarterly* **45**, 135–143 (1999)
6. Benveniste, A., Caillaud, B., Passerone, R.: A generic model of contracts for embedded systems. Tech. Rep. 6214, INRIA Rennes (2007)
7. Benveniste, A., Caspi, P., Le Guernic, P., Marchand, H., Talpin, J.P., Tripakis, S.: A protocol for loosely time-triggered architectures. In: J. Sifakis, S.A. Vincentelli (eds.) *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software, Lecture Notes in Computer Science*, vol. 2491, pp. 252–265. Springer Verlag (2002)
8. Besnard, L., Gautier, T., Le Guernic, P., Talpin, J.P.: Compilation of polychronous dataflow equations. In: this book
9. Broy, M.: Compositional refinement of interactive systems. *Journal of the ACM* **44**(6), 850–891 (1997)
10. Doyen, L., Henzinger, T.A., Jobstmann, B., Petrov, T.: Interface theories with component reuse. In: *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pp. 79–88. ACM (2008)
11. Edwards, S., Lavagno, L., Lee, E.A., Sangiovanni-Vincentelli, A.: Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE* **85**(3), 366–390 (1997)
12. Glouche, Y., Le Guernic, P., Talpin, J.P., Gautier, T.: A boolean algebra of contracts for logical assume-guarantee reasoning. Tech. Rep. 6570, INRIA Rennes (2008)
13. Glouche, Y., Talpin, J.P., Le Guernic, P., Gautier, T.: A boolean algebra of contracts for logical assume-guarantee reasoning. In: *6th International Workshop on Formal Aspects of Component Software (FACS 2009)* (2009)
14. Glouche, Y., Talpin, J.P., Le Guernic, P., Gautier, T.: A module language for typing by contracts. In: E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.) *Proceedings of the First*

- NASA Formal Methods Symposium, pp. 86–95. NASA Ames Research Center, Moffett Field, CA, USA (2009)
15. Halbwachs, N., Lagnier, F., Raymond, P.: Synchronous observers and the verification of reactive systems. In: AMAST '93: Proceedings of the Third International Conference on Methodology and Software Technology, pp. 83–96. Springer-Verlag (1994)
  16. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM* **12**(10), 576–580 (1969)
  17. Kopetz, H.: Component-based design of large distributed real-time systems. *Control Engineering Practice* **6**(1), 53–60 (1997)
  18. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: R. De Nicola (ed.) *ESOP, Lecture Notes in Computer Science*, vol. 4421, pp. 64–79. Springer (2007)
  19. Le Guernic, P., Gautier, T., Le Borgne, M., Le Maire, C.: Programming real-time applications with SIGNAL. *Proceedings of the IEEE* **79**(9), 1321–1336 (1991)
  20. Le Guernic, P., Talpin, J.P., Le Lann, J.C.: Polychrony for System Design. *Journal for Circuits, Systems and Computers* **12**(3), 261–304 (2003)
  21. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: H. Kilov, B. Rumpe, W. Harvey (eds.) *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Kluwer Academic Publishers (1999)
  22. Marainchi, F., Morel, L.: Logical-time contracts for reactive embedded components. In: *EUROMICRO*, pp. 48–55. IEEE Computer Society (2004)
  23. Marchand, H., Bournai, P., Le Borgne, M., Le Guernic, P.: Synthesis of discrete-event controllers based on the Signal environment. *Discrete Event Dynamic System: Theory and Applications* **10**(4), 325–346 (2000)
  24. Marchand, H., Rutten, E., Le Borgne, M., Samaan, M.: Formal verification of programs specified with Signal: Application to a power transformer station controller. *Science of Computer Programming* **41**(1), 85–104 (2001)
  25. Meyer, B.: *Object-oriented software construction* (2nd ed.). Prentice-Hall, Inc. (1997)
  26. Mitchell, R., McKim, J., Meyer, B.: *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc. (2002)
  27. Racllet, J.B., Badouel, E., Benveniste, A., Caillaud, B., Passerone, R.: Why are modalities good for interface theories? In: *Proc. of the 9th International Conference on Application of Concurrency to System Design (ACSD'09)*, pp. 119–127. IEEE Computer Society Press (2009)