A Boolean algebra of contracts for assume-guarantee reasoning

Yann Glouche[1] Paul Le Guernic Jean-Pierre Talpin
Thierry Gautier

INRIA, Unité de Recherche Rennes-Bretagne-Atlantique
Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract

Contract-based design is an expressive paradigm for a modular and compositional specification of programs. It is in turn becoming a fundamental concept in mainstream industrial computer-aided design tools for embedded system design. In this paper, we elaborate new foundations for contract-based embedded system design by proposing a general-purpose algebra of assume/guarantee contracts based on two simple concepts: first, the assumption or guarantee of a component is defined as a filter and, second, filters enjoy the structure of a Boolean algebra. This yields a structure of contracts that is a Heyting algebra.

1 Introduction

Common methodological guidelines for attacking the design of large embedded architectures advise the validation of specifications as early as possible and an iterative validation of each refinement or modification made to the initial specification, until the implementation of the system is finalized. 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 [10,15]. These components have to be provided with their conditions of use and guarantees that they have been validated when these conditions are satisfied. This represents 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 a different paradigm of contract to define a component-based validation technique in the context of a synchronous modeling framework. In our model, a component is represented by an abstract view of its behaviors. It has a finite set of input/output variables to

[1] Partially funded by the EADS Foundation.

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 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 subject 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. It is based on two simple concepts.

First, 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. Second and foremost, we define a Boolean algebra to manipulate process-filters. This yields an algebraically rich structure which allows us to reason on 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 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.

Plan

The article is organized as follows. Section 2 introduces a suitably general algebra of processes which borrows its notations and concepts from domain theory [1]. A contract (**A**,**G**) is viewed as a pair of logical devices filtering processes: the assumption **A** filters processes to select (accept or conversely reject) those that are asserted (accepted or conversely rejected) by the guarantee **G**. Process-filters are defined in Section 3 and contracts in Section 4. Section 5 discusses application of our model to the synchronous Signal language. Related works are further discussed in Section 6. Section 7 concludes the presentation. Detailed proofs of all properties presented in this article are available in a technical report [12].

2 An algebra of processes

We start with the definition of a suitable algebra for behaviors and processes. Usually, a behavior describes the trace of a discrete process (a Mazurkiewicz trace [24] or a tuple of signals in Lee's tagged signal model [18]). We deliberately choose a more abstract definition in order to encompass not only sequences of Boolean, integer, real variables but also behaviors of more complex systems such as hybrid systems or, on the contrary, simpler "behaviors" associating scalar values with variables, to represent execution cost, memory size, etc. In this paper we focus the presentation on usual process behaviors.

Definition 2.1 [Behavior] Let $\mathcal{V}$ be an infinite, countable set of variables, and $\mathcal{D}$ a set of values; for $\mathbf{Y}$, a finite set of variables included in $\mathcal{V}$ (written $\mathbf{Y} \subset \mathcal{V}$), $\mathbf{Y}$ nonempty, a $\mathbf{Y}$-*behavior* is a function $b : \mathbf{Y} \to \mathcal{D}$.

The set of $\mathbf{Y}$-behaviors is denoted by $\mathbb{B}_\mathbf{Y} =_\Delta \mathbf{Y} \to \mathcal{D}$. Definition 2.1 is extended to the empty variable domain: $\mathbb{B}_\emptyset =_\Delta \emptyset$ (there is no behavior associated with the empty set of variables). For $\mathbf{Y}$, a finite set of variables included in $\mathcal{V}$, $\mathbf{Y}$ nonempty, c a $\mathbf{Y}$-behavior, $\mathbf{X}$ a (possibly empty) subset of $\mathbf{Y}$, $c_{|\mathbf{X}}$ is the restriction of c on $\mathbf{X}$, $c_{|\mathbf{X}} =_\Delta \{(x, c(x)) | x \in \mathbf{X}\}$, $c_{|\emptyset} =_\Delta \emptyset$; then $c_{|\mathbf{Y}} = c$.

In Figure 1, the x, y-behaviors b_1 and b_2 are functions from the variables x, y to functions that denote signals. Behavior b_1 is a discrete sampling mapping a domain of time represented by natural numbers to values. Behavior b_2 associates x, y to continuous functions of time.

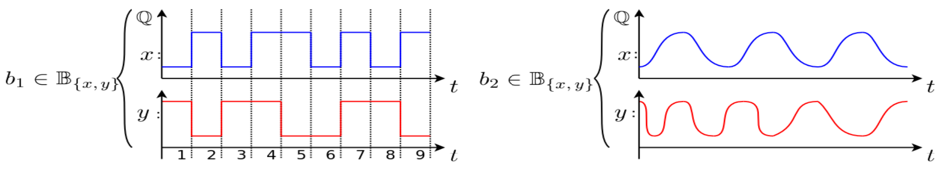

Figure 1. Examples of behaviors.

We define a *process* as a set of behaviors on a given set of variables.

Definition 2.2 [Process] For $\mathbf{X}$, a finite set of variables ($\mathbf{X} \subset \mathcal{V}$), an $\mathbf{X}$-*process* p is a nonempty set of $\mathbf{X}$-behaviors.

Thus, since $\mathbb{B}_\emptyset = \emptyset$, there is a unique $\emptyset$-process, designated by $\Omega =_\Delta \{\emptyset\}$; Ω has the empty behavior as unique behavior. The *empty process* is denoted by $\mho =_\Delta \emptyset$.

Since Ω does not have any variable, it has no effect when composed (intersected) with other processes. It can be seen as the universal process, for constraint conjunction, in contrast with $\mho$, the empty set of behaviors, the use of which in constraint conjunction always results in the empty set. $\mho$ can be seen as the null process.

For $\mathbf{X}$, a finite set of variables ($\mathbf{X} \subset \mathcal{V}$), we denote by $\mathbb{P}_\mathbf{X} =_\Delta \mathcal{P}(\mathbb{B}_\mathbf{X}) \setminus \{\mho\}$ the set of $\mathbf{X}$-processes ($\mathbb{P}_\emptyset = \{\Omega\}$). $\mathbb{P} =_\Delta \cup_{(\mathbf{X} \subset \mathcal{V})} \mathbb{P}_\mathbf{X}$ denotes the set of all processes. The domain of behaviors in an $\mathbf{X}$-process p is denoted by $var(p) =_\Delta \mathbf{X}$.

A process is a nonempty set of behaviors. Then we extend $\mathbb{P}$ to $\mathbb{P}^\star =_\Delta \mathbb{P} \cup \{\mho\}$ and $\forall \mathbf{X} \subset \mathcal{V}$, $\mathbb{P}_\mathbf{X}$ to $\mathbb{P}^\star{}_\mathbf{X} =_\Delta \mathbb{P}_\mathbf{X} \cup \{\mho\}$. Moreover, we extend the definition of $var(p)$ to $var(\mho) =_\Delta \mathcal{V}$.

The following operators will be used to define filters and contracts: the complementary of a process p in $\mathbb{P}_\mathbf{X}$ is a process in $\mathbb{P}^\star{}_\mathbf{X}$; the restriction of a process p in $\mathbb{P}_\mathbf{X}$ to $\mathbf{Y} \subseteq \mathbf{X} \subset \mathcal{V}$ is the abstraction (projection) of p to $\mathbf{Y}$; finally, the extension of p in $\mathbb{P}_\mathbf{X}$ to $\mathbf{Y} \subset \mathcal{V}$, $\mathbf{Y}$ finite, is the process on $\mathbf{Y}$ that has the same constraints as p.

Definition 2.3 [Complementary, restriction and extension] For $\mathbf{X}$, a finite set of variables ($\mathbf{X} \subset \mathcal{V}$), the complementary $\widetilde{p}$ of a process $p \in \mathbb{P}_{\mathbf{X}}$ is defined by $\widetilde{p} =_{\Delta} (\mathbb{B}_{\mathbf{X}} \setminus p)$. Also, $\widetilde{\mathbb{B}_{\mathbf{X}}} = \mho$. When $\mathbf{X}$, $\mathbf{Y}$ are finite sets of variables such that $\mathbf{X} \subseteq \mathbf{Y} \subset \mathcal{V}$, we define by $q_{|\mathbf{X}} =_{\Delta} \{c_{|\mathbf{X}} | c \in q\}$ the restriction $q_{|\mathbf{X}} \in \mathbb{P}_{\mathbf{X}}$ of $q \in \mathbb{P}_{\mathbf{Y}}$ and by $p^{|\mathbf{Y}} =_{\Delta} \{c \in \mathbb{B}_{\mathbf{Y}} | c_{|\mathbf{X}} \in p\}$ the extension $p^{|\mathbf{Y}} \in \mathbb{P}_{\mathbf{Y}}$ of $p \in \mathbb{P}_{\mathbf{X}}$. Hence, we have $q_{|\emptyset} = \Omega$, $q_{|var(q)} = q$, $\Omega^{|\mathbf{Y}} = \mathbb{B}_{\mathbf{Y}}$ and $p^{|var(p)} = p$.

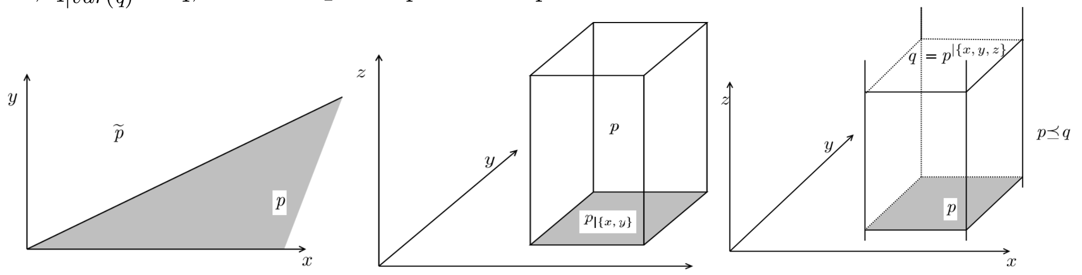

Figure 2. Complementary, restriction and extension of a process.

The complementary $\widetilde{p}$ of a process p defined on the variables x and y, Figure 2, consists of all behaviors defined on x, y not belonging to p. The restriction $p_{|\{x,y\}}$ of a process p defined on x, y, z, consists of its projection on the restricted domain; right, the extension $p^{|\{x,y,z\}}$ of a process p defined on x, y is the largest process defined on x, y, z whose restriction on x, y is equal to p.

The set $\mathbb{P}^{\star}{}_{\mathbf{X}}$, equipped with union, intersection and complementary, extended with $\widetilde{\mho} = \mathbb{B}_{\mathbf{X}}$, is a Boolean algebra with supremum $\mathbb{P}^{\star}{}_{\mathbf{X}}$ and infimum $\mho$. The definition of restriction is extended to $\mho$, the null process, with $\mho_{|\mathbf{X}} =_{\Delta} \{c_{|\mathbf{X}} | c \in \emptyset\} = \mho$. Since $\mathcal{V}$ is the set of all variables, the definition of extension is simply extended to $\mho$, with $\mho^{|\mathcal{V}} =_{\Delta} \mho$. The process extension operator induces a partial order $\preceq$, such that $p \preceq q$ if q is an extension of p to variables of q; the relation $\preceq$, used to define filters, is studied below.

Definition 2.4 [Process extension relation] The process extension relation $\preceq$ is defined by: $(\forall\ p \in \mathbb{P})\ (\forall\ q \in \mathbb{P})\ \ (p \preceq q) =_{\Delta} ((var(p) \subseteq var(q)) \wedge (p^{|var(q)} = q))$

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 $\mathbb{P}^{\star}$ with $(\mho \preceq \mho)$.

Property 2.5 $(\mathbb{P}^{\star}, \preceq)$ is a poset.

Checking transitivity, antisymmetry and reflexivity is immediate. 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 $p{\uparrow_{\preceq}} =_{\Delta} \{q \in \mathbb{P} | p \preceq q\}$. The extension upper set is illustrated in Figure 3.

To study properties of *extension upper sets*, we characterize the set of variables that are constrained by a given process: we write that a process $q \in \mathbb{P}$ *controls* some variable y, if y belongs to $var(q)$ and q is not equal to the extension on $var(q)$ of its projection on $(var(q) \setminus \{y\})$.

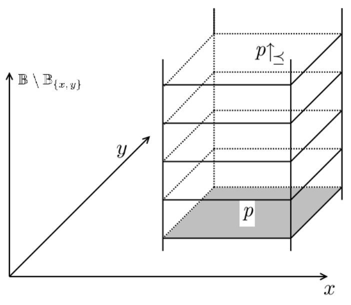

Figure 3. Extension upper set.

This is illustrated in Figure 4, there is some behavior b in q that has the same restriction on $(var(q)\backslash\{y\})$ as some behavior c in $\mathbb{B}_{var(q)}$ such that c does not belong to q; thus q is strictly included in $(q_{|(var(q)\backslash\{y\})})^{|var(q)}$.

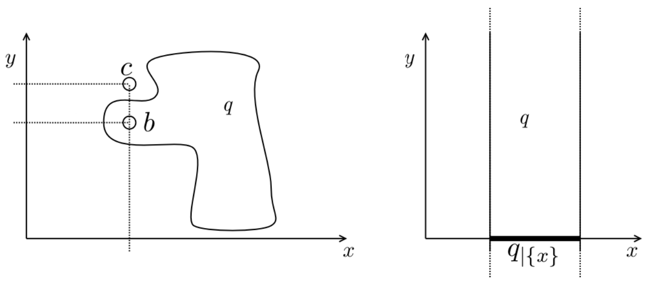

Figure 4. Controlled (left) and non-controlled (right) variable y in a process q.

Formally, a process $q \in \mathbb{P}$ controls a variable y, written $(q \rhd y)$, iff $(y \in var(q))$ and $q \neq ((q_{|(var(q)\backslash\{y\})})^{|var(q)})$. A process $q \in \mathbb{P}$ controls a variable set $\mathbf{X}$, written $(q \rhd \mathbf{X})$, iff $(\forall\ x \in \mathbf{X})\ (q \rhd x)$.
Moreover, $\rhd$ is extended to $\mathbb{P}^\star$ with $\mho \rhd \mathcal{V}$. Note that if a process p controls $\mathbf{X}$, this does not imply that, for all $x \in \mathbf{X}$, $y \in \mathbf{X}$, $x \neq y$, $(p_{|(\mathbf{X}\backslash\{x\})})$ controls y: it may be the case that x is constrained in p by y; then if x is "removed" (by the projection on other controlled variables), y may be free in this projection. We define a *reduced process* (the key concept to define filters) as being a process that controls all of its variables.

Definition 2.6 [Reduced process] A process $p \in \mathbb{P}^\star$ is *reduced* iff $p \rhd var(p)$.

For instance, Ω is reduced. On the contrary, $\mathbb{B}_\mathbf{X}$ is never reduced when $\mathbf{X}$ is not empty. *Reduced processes* are minimal in $(\mathbb{P},\preceq)$. We denote by $\overset{\triangledown}{q}$, called *reduction of* q, the (minimal) process such that $\overset{\triangledown}{q} \preceq q$ (p is reduced iff $\overset{\triangledown}{p} = p$). For all $\mathbf{X}$, we have $\overset{\triangledown}{\mathbb{B}_\mathbf{X}} = \Omega$.

Figure 5 illustrates the reduction $\overset{\triangledown}{q}$ of a process q and a process p, in the extension upper set $\overset{\triangledown}{q}\!\uparrow_\preceq$. Assuming that $var(q) = (\{x_{1...n}\} \cup \{y_{1...m}\})$ and that q controls the variables $\{x_{1...n}\}$, we have $var(\overset{\triangledown}{q}) = \{x_{1...n}\}$; the process p is such that $p \in \overset{\triangledown}{q}\!\uparrow_\preceq$ with $var(p) \subseteq (\{x_{1...n}\} \cup \{y_{1...m}\} \cup \{z_{1...l}\})$; it controls the variables $\{x_{1...n}\}$, and $\{y_{1...m}\} \cup \{z_{1...l}\}$ is a set of free variables, such that $\overset{\triangledown}{q} = \overset{\triangledown}{p}$.

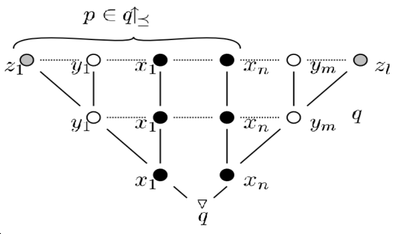
$p \in \overset{\triangledown}{q}\!\uparrow_\preceq$
z_1
y_1
x_1
x_n
y_m
z_l
y_1
x_1
x_n
y_m
q
x_1
x_n
$\overset{\triangledown}{q}$

Figure 5. Reduction of a process.

Property 2.7 The complementary $\widetilde{p}$ of a nonempty process p strictly included in $\mathbb{B}_{var(p)}$ is reduced iff p is reduced; then $\widetilde{p}$ and p control the same set of variables $var(p)$.

From the above, the extension upper set $\overset{\triangledown}{p}\!\uparrow_\preceq$ of the reduction of p is a (principal) filtered set [1]: it is nonempty and each pair of elements has a lower bound. Then $\overset{\triangledown}{p}\!\uparrow_\preceq$ 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 $\overset{\triangledown}{p}\!\uparrow_\preceq$). We also observe that $var(\overset{\triangledown}{q})$ is the greatest subset of variables such that $q \rhd var(\overset{\triangledown}{q})$. For a process $q \in \mathbb{P}^\star$, we extend the definition of $var()$ to the extension upper set of its reduction by $var(\overset{\triangledown}{q}\!\uparrow_\preceq) =_\Delta var(\overset{\triangledown}{q})$. Notice that $\mho\!\uparrow_\preceq = \{\mho\}$.

We define the *inclusion lower set* of a set of processes to capture all the subsets of behaviors of these processes. Let $\mathbf{R} \subseteq \mathbb{P}^\star$, $\mathbf{R}{\downarrow_\subseteq}$ is the inclusion lower set of $\mathbf{R}$ for $\subseteq$ defined by $\mathbf{R}{\downarrow_\subseteq} =_\Delta \{p \in \mathbb{P}^\star | (\exists\ q \in \mathbf{R})\ (p \subseteq q)\}$. Hence, none of the processes but $\mho$ belongs to the *inclusion lower set* of the *extension upper set* of $\mho$; on the contrary, all processes belong to the *inclusion lower set* of the *extension upper set* of Ω: $[\overset{\triangledown}{\mho}{\uparrow_\preceq}]{\downarrow_\subseteq} = \{\mho\}$ and $[\overset{\triangledown}{\Omega}{\uparrow_\preceq}]{\downarrow_\subseteq} = \mathbb{P}^\star$.

3 An algebra of filters

In this section, we define a *process-filter* by the set of processes that satisfy a given property. We propose an order relation ($\sqsubseteq$) on the set of process-filters Φ. We establish that (Φ,$\sqsubseteq$) is a Boolean algebra. A *process-filter* $\mathbf{R}$ is a subset of $\mathbb{P}^\star$ 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 $\mho$ 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 3.1 [Process-filter] A set of processes $\mathbf{R}$ is a *process-filter* iff $(\exists\ r \in \mathbb{P}^\star)$ $((r = \overset{\triangledown}{r}) \wedge (\mathbf{R} = [r{\uparrow_\preceq}]{\downarrow_\subseteq}))$. 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 $\widehat{p}$ $=_\Delta [\overset{\triangledown}{p}{\uparrow_\preceq}]{\downarrow_\subseteq}$. The generator of a process-filter $\mathbf{R}$ is unique, we refer to it as $\overset{\triangledown}{\mathbf{R}}$. Ω generates the set of all processes (including $\mho$) and $\mho$ belongs to all filters. Formally, $(\forall\ p,r,s \in \mathbb{P}^\star)$, we have:

$$(p \in \widehat{r}) \Longrightarrow (var(\overset{\triangledown}{r}) \subseteq var(p)) \qquad \widehat{r} = \widehat{s} \Longleftrightarrow \overset{\triangledown}{r} = \overset{\triangledown}{s} \qquad \Omega \in \widehat{r} \Longleftrightarrow \widehat{r} = \mathbb{P}^\star$$

Figure 6 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).

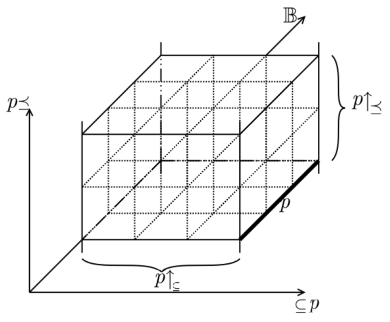

Figure 6. Example of process-filter.

We denote by Φ the set of process-filters. We call *strict process-filters* the process-filters that are neither $\mathbb{P}^\star$ nor $\{\mho\}$. The filtered variable set of a process-filter $\mathbf{R}$ is $var(\mathbf{R})$ defined by $var(\mathbf{R}) =_\Delta var(\overset{\triangledown}{\mathbf{R}})$.

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

Definition 3.2 [Process-filter relaxation] For $\mathbf{R}$ and $\mathbf{S}$, two process-filters, let $\mathbf{Z}$ $= var(\mathbf{R}) \cup var(\mathbf{S})$. The relation $\mathbf{S}$ *relaxes* $\mathbf{R}$, written $\mathbf{R} \sqsubseteq \mathbf{S}$, is defined by:
$\{\mho\} \sqsubseteq \mathbf{S} \qquad (\mathbf{R} \sqsubseteq \{\mho\}) \Longleftrightarrow \{\mho\} = \mathbf{R} \qquad (\mathbf{R} \sqsubseteq \mathbf{S} \Longleftrightarrow \overset{\triangledown}{\mathbf{R}}^{|\mathbf{Z}} \subseteq \overset{\triangledown}{\mathbf{S}}^{|\mathbf{Z}})$

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

Lemma 3.3 *$(\Phi,\sqsubseteq)$ is a lattice of supremum $\mathbb{P}^{\star}$ and infimum $\{\mho\}$. Let $\mathbf{R}$ and $\mathbf{S}$ be two process-filters, $\mathbf{V} = var(\mathbf{R}) \cup var(\mathbf{S})$, $\mathbf{R_V} = \overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}$ and $\mathbf{S_V} = \overset{\triangledown}{\mathbf{S}}^{|\mathbf{V}}$. Conjunction $\mathbf{R} \sqcap \mathbf{S}$, disjunction $\mathbf{R} \sqcup \mathbf{S}$ and complementary $\widetilde{\mathbf{R}}$ are defined by:*

$$\{\mho\} \sqcap \mathbf{R} =_{\Delta} \{\mho\} \qquad \mathbf{R} \sqcap \mathbf{S} =_{\Delta} [(\mathbf{R_V} \overset{\triangledown}{\cap} \mathbf{S_V})\Uparrow_{\preceq}]\downarrow_{\subseteq} \qquad \widetilde{\mathbf{R}} =_{\Delta} [\overset{\widetilde{\triangledown}}{\mathbf{R}}\uparrow_{\preceq}]\downarrow_{\subseteq}$$

$$\{\mho\} \sqcup \mathbf{R} =_{\Delta} \mathbf{R} \qquad \mathbf{R} \sqcup \mathbf{S} =_{\Delta} [(\mathbf{R_V} \overset{\triangledown}{\cup} \mathbf{S_V})\Uparrow_{\preceq}]\downarrow_{\subseteq} \qquad \widetilde{\mathbb{P}^{\star}} =_{\Delta} \{\mho\}$$

$$\widetilde{\{\mho\}} =_{\Delta} \mathbb{P}^{\star}$$

If $\mathbf{R} \neq \{\mho\}$ then $\widetilde{\mathbf{R}} \neq \{\mho\}$ and $\overset{\widetilde{\triangledown}}{\mathbf{R}} = (\mathbb{B}_{var(\mathbf{R})} \setminus \overset{\triangledown}{\mathbf{R}})$ is reduced and $var(\mathbf{R}) = var(\widetilde{\mathbf{R}})$.

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 $\overset{\triangledown}{\mathbf{R}}$ and $\overset{\triangledown}{\mathbf{S}}$ 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 same mechanism, with union, is used to define disjunction. And the complementary of a strict process-filter $\mathbf{R}$ is the process-filter generated by the complementary of its generator $\overset{\triangledown}{\mathbf{R}}$.

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 $\mathbf{S}$.

Example 3.4 Let x, a variable taking values in {0,1,2,3} and u, y, v three variables taking values in {0,1}; let $r \in \mathbb{P}_{\{u,\ x,\ y\}}$, $s \in \mathbb{P}_{\{x,\ y,\ v\}}$, two reduced processes defined by

$$r = \{b|b(u) \in \{0,1\} \wedge b(x) \in \{0,1\} \wedge b(y) \in \{0,1\}\} \cup \{\{(u,1),(x,2),(y,0)\}\}$$
$$s = \{b|b(x) \in \{0,1\} \wedge b(y) \in \{0,1\} \wedge b(v) \in \{0,1\}\} \cup \{\{(x,3),(y,1),(v,0)\}\}$$

We observe that $r \rhd \{u,x,y\}$; u and y are free in r when x is 0 or 1; v is free whatever the value of x is in r. We also have $s \rhd \{x,y,v\}$; y and v are free in s when x is 0 or 1; thus u is free whatever the value of x is in s. From the above definitions, we have that $p =_{\Delta} r \cap s = \{\mathrm{b}|b(u) \in \{0,1\} \wedge b(x) \in \{0,1\} \wedge b(y) \in \{0,1\} \wedge b(v) \in \{0,1\}\}$ and $\overset{\triangledown}{p} = \{\mathrm{b}|b(x) \in \{0,1\}\}$.

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 $\mathbf{S}$.

Example 3.5 Let x, a variable taking values in {0,1,2,3} and u, y, v three variables taking values in {0,1}; let $r \in \mathbb{P}_{\{u,\ x,\ y\}}$, $s \in \mathbb{P}_{\{x,\ y,\ v\}}$, two reduced processes such that

$$r = \{b|b(u) \in \{0,1\} \wedge b(x) \in \{0,1\} \wedge b(y) = 0\}$$
$$s = \{b|b(x) \in \{0,1\} \wedge b(y) = 1 \wedge b(v) \in \{0,1\}\}$$

Hence, $p =_{\Delta} r \cup s = \{\mathrm{b} | b(u) \in \{0,1\} \wedge b(x) \in \{0,1\} \wedge b(y) \in \{0,1\} \wedge b(v) \in \{0,1\}\}$ and $\overset{\triangledown}{p} = \{\mathrm{b} | b(x) \in \{0,1\}\}$.

Now we can state a first main result, which is that process-filters form a Boolean algebra.

Theorem 3.6 *$(\Phi,\sqsubseteq)$ is a Boolean algebra with $\mathbb{P}^\star$ as 1, $\{\mho\}$ as 0 and the complementary $\widetilde{\mathbf{R}}$.*

Variable elimination operators are defined on process-filters.

Definition 3.7 [Variable elimination in process-filter] Let x be a variable, $\mathbf{R}$ a process-filter, and $\mathbf{X} =_{\Delta} var(\mathbf{R})$. The E-elimination of x in $\mathbf{R}$, noted $\mathbf{R}_{|\exists x}$, is the projection of $\mathbf{R}$ on controlled variables other than x. The generator of the U-elimination of x in $\mathbf{R}$ (U-elimination of x in $\mathbf{R}$ is noted $\mathbf{R}_{|\forall x}$) contains the behaviors of $\overset{\triangledown}{\mathbf{R}}$ restricted on $\mathbf{X}\backslash\{x\}$ for which x is free in $\overset{\triangledown}{\mathbf{R}}$.

$$\mathbf{R}_{|\exists x} =_{\Delta} \begin{cases} \widehat{(\overset{\triangledown}{\mathbf{R}})_{|\mathbf{X}\backslash\{x\}}}, & x \in \mathbf{X} \\ \mathbf{R}, & \text{otherwise} \end{cases} \qquad \mathbf{R}_{|\forall x} =_{\Delta} \widetilde{\widetilde{\mathbf{R}}_{|\exists x}}$$

Notice that $\mathbf{R}_{|\forall x} \sqsubseteq \mathbf{R} \sqsubseteq \mathbf{R}_{|\exists x}$.

Example 3.8 Let $\mathbf{R}$, a process-filter generated by $((x > 0) \Rightarrow (y > 0)) \wedge ((y > 0) \Rightarrow (z > 0))$. Then $\mathbf{R}_{|\exists x}$ is generated by $((y > 0) \Rightarrow (z > 0))$ and $\mathbf{R}_{|\forall x}$ is generated by $((y > 0) \wedge (z > 0))$.

4 An algebra of contracts

We define the notion of assume/guarantee contract and propose a refinement relation on contracts.

Definition 4.1 [Contract] A *contract* $\mathbf{C} = (\mathbf{A},\mathbf{G})$ is a pair of process-filters. $var(\mathbf{C})$, the variable set of $\mathbf{C} = (\mathbf{A},\mathbf{G})$, is defined by $var(\mathbf{C}) =_{\Delta} var(\mathbf{A}) \cup var(\mathbf{G})$. $\mathbb{C} =_{\Delta} \Phi \times \Phi$ 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 $\mathbf{A}$.

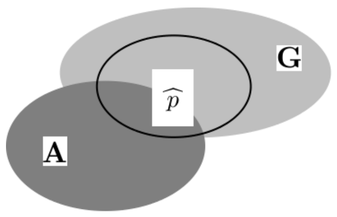

Figure 7. A process p satisfying a contract $(\mathbf{A},\mathbf{G})$.

Figure 7 depicts a process p *satisfying* the contract $(\mathbf{A},\mathbf{G})$ ($\widehat{p}$ is the process-filter generated by the reduction of p).

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 $\mathbf{G}$; this is made more precise and formal by the following definition.

Definition 4.2 [Satisfaction] Let $\mathbf{C} = (\mathbf{A},\mathbf{G})$ a contract; a process p satisfies $\mathbf{C}$, written $p \vDash \mathbf{C}$, iff $(\widehat{p} \sqcap \mathbf{A}) \sqsubseteq \mathbf{G}$.

Property 4.3 $p \vDash \mathbf{C} \iff \widehat{p} \sqsubseteq (\widetilde{\mathbf{A}} \sqcup \mathbf{G})$

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 4.4 [Satisfaction preorder] $(\mathbf{A}_1,\mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)$ iff $(\forall\ p \in \mathbb{P})((p \vDash (\mathbf{A}_1,\mathbf{G}_1)) \implies (p \vDash (\mathbf{A}_2,\mathbf{G}_2)))$

The preorder on contracts satisfies the following property:

Property 4.5 $(\mathbf{A}_1,\mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)$ iff $(\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqsubseteq (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)$

Definition 4.6 [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 $\mathbf{C}_1 \preccurlyeq \mathbf{C}_2$, 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.

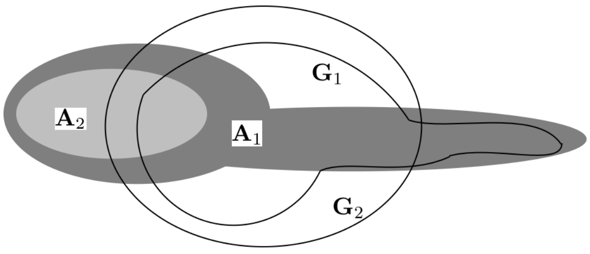

Figure 8. Refinement of contracts.

Figure 8 depicts a contract $(\mathbf{A}_1,\mathbf{G}_1)$ that refines a contract $(\mathbf{A}_2,\mathbf{G}_2)$. Among contracts that can 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 $\mathbf{A}_1 \sqcup \mathbf{G}_2$. But other choices could have been made.

By definition of the satisfaction pre-order, we can express the refinement relation in the algebra of process-filters as follows:

Property 4.7 $(\mathbf{A}_1,\mathbf{G}_1) \preccurlyeq (\mathbf{A}_2,\mathbf{G}_2)$ 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 $(\preccurlyeq)$ 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.

Lemma 4.8 (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:*

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

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 refines b [6]. 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 $\mathbf{X} = (\mathbf{I},\mathbf{J})$ of $\mathbb{C}$ such that the greatest lower bound of $\mathbf{C}_1$ and $\mathbf{X}$ refines $\mathbf{C}_2$, with:
$\mathbf{I} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_1 \sqcap \widetilde{\mathbf{G}_2})$ $\qquad$ $\mathbf{J} = \mathbf{G}_2 \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_1}) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \widetilde{\mathbf{G}_1})$
Then our contract algebra is a Heyting algebra. In particular, it is distributive.

Theorem 4.9 *$(\mathbb{C}, \preccurlyeq)$ is a Heyting algebra with supremum $(\{\mho\}, \mathbb{P}^\star)$ and infimum $(\mathbb{P}^\star, \{\mho\})$.*

Let x a variable, $\mathbf{C} = (\mathbf{A}, \mathbf{G})$ a contract, the elimination of x in $\mathbf{C}$ is the contract $\mathbf{C}_{\setminus x}$ defined by $\mathbf{C}_{\setminus x} =_\Delta (\mathbf{A}_{|\forall x}, \mathbf{G}_{|\exists x})$.

Property 4.10 A contract $\mathbf{C}$ refines the elimination of a variable in $\mathbf{C}$: $\mathbf{C} \preccurlyeq \mathbf{C}_{\setminus x}$

5 Application to the Signal language

In the synchronous multiclocked model of Signal [19], a process (noted p), consists of the synchronous composition (noted $p \,|\, q$) of equations on signals (noted $x = y \, f \, z$). A signal x consists of an infinite flow of values that is discretely sampled according to the pace of its clock. A set of tags t is to denote symbolic periods in time during which transitions take place. It samples a signal over a countable series of causally related tags. Then the events, signals, behaviors and processes are defined as follows:
- an *event* e is a pair consisting of a tag t and a value v,
- a *signal* s is a function from a *chain* of tags to a set of values,
- a *behavior* b is a function from a set of names to signals,
- a *process* $p \in \mathbb{P}$ is a set of behaviors that have the same domain.

Synchronous composition $p \,|\, q$ consists of the simultaneous solution of the equations in p and q at all times.

Definition 5.1 [Synchronous composition of processes] The synchronous composition of two processes $p, q \in \mathbb{P}$ is defined by: $p \,|\, q =_\Delta \{\, b \cup c \mid (b, c) \in p \times q \wedge b_{|var(p) \,\cap\, var(q)} = c_{|var(p) \,\cap\, var(q)} \,\}$

In the context of component-based or contract-based engineering, refinement and substitutability are recognized as being fundamental requirements [9]. 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 properties 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 compositon of Signal processes and the greatest lower bound as a composition operator for contracts, we have:

Property 5.2 Let two processes $p, q \in \mathbb{P}$, and contracts $\mathbf{C}_1$, $\mathbf{C}_2$, $\mathbf{C'}_1$, $\mathbf{C'}_2 \in \mathbb{C}$.

(1) $\mathbf{C}_1 \preccurlyeq \mathbf{C}_2 \implies ((p \vDash \mathbf{C}_1) \implies (p \vDash \mathbf{C}_2))$
(4) $((p \vDash \mathbf{C}_1) \wedge (q \vDash \mathbf{C}_2)) \implies ((p \,|\, q) \vDash (\mathbf{C}_1 \Downarrow \mathbf{C}_2))$
(2) $\mathbf{C}_1 \rightsquigarrow \mathbf{C}_2 \iff ((p \vDash \mathbf{C}_1) \implies (p \vDash \mathbf{C}_2))$
(5) $((p \vDash \mathbf{C}_1) \wedge (p \vDash \mathbf{C}_2)) \iff (p \vDash (\mathbf{C}_1 \Downarrow \mathbf{C}_2))$
(3) If $\mathbf{C'}_1 \preccurlyeq \mathbf{C}_1$ and $\mathbf{C'}_2 \preccurlyeq \mathbf{C}_2$ then $\mathbf{C'}_1 \Downarrow \mathbf{C'}_2 \preccurlyeq \mathbf{C}_1 \Downarrow \mathbf{C}_2$

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

In [13], we develop a module system based on our paradigm of contract for the Signal formalism, and applied it to the specification of a component-based design process. This module system, embedding data-flow equations defined in Signal syntax, 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 tended to Signal's model-checker, Sigali [21], 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 [20] 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.

We now illustrate the distinctive features of our contract algebra by considering the specification of a four-stroke engine and its translation into observers in the synchronous language Signal.

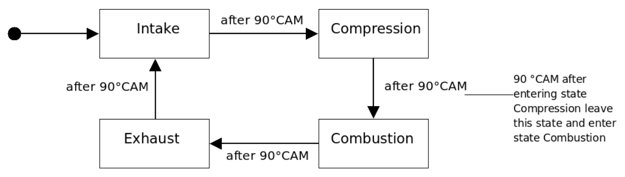
Intake
after 90°CAM
Compression
after 90°CAM
after 90°CAM
90 °CAM after entering state Compression leave this state and enter state Combustion
Exhaust
after 90°CAM
Combustion

Figure 9. State machine of 4-stroke engine cycle.

Figure 9 represents a state machine that denotes the successive operation modes of a 4-stroke engine: *Intake*, *Compression*, *Combustion* and *Exhaust*. They are driven by the camshaft whose position is measured in degrees.

The angle of the camshaft defines a discrete timing reference, the *clock cam*, measured in degrees $CAM°$, of initial value 0. Transitions in the state machine are triggered by measures of the camshaft angle. The variables cam, $Intake$, $Compression$, $Combustion$, $Exhaust$ model the behavior of the engine. We wish to define a contract to stipulate that intake always takes place in the first quarter on the camshaft revolution. To do this, we define the *generator* of a process-filter for the assumption. It should be a measure of the environmental variable cam. Namely cam should be in the first quarter. Under these assumptions, the state machine should be guaranteed to be in the intake mode, hence the generator of the process-filter for the guarantee: $\mathbf{A}_{Intake} =_{\Delta} (cam\ mod\ 360\ <\ 90)$ $\mathbf{G}_{Intake} =_{\Delta} Intake$

The generic structure of processes in contracts finds a direct instance and compositional translation into the synchronous multi-clocked model of computation of Signal. Using Signal equations:

$$\mathbf{A}_{Intake} = true\ when (cam\ mod 360 < 90) \mathbf{G}_{Intake} = true\ when\ intake\ default\ false$$

A subtlety of the Signal language is that the contract not only talks about the value, true or false, of the signals, but also about the status of the signal names, present or absent. Hence, the signal $\mathbf{A}_{Intake}$ is present and true iff cam is present and less than 90. Hence, in Signal, the complementary of the assumptions is simply defined by $\widetilde{\mathbf{A}_{Intake}} = false\ when\ \mathbf{A}_{Intake}\ default\ true$ to mean that it is true iff cam is absent or bigger than 90. Notice that, for a trace of the assumptions $\mathbf{A}_{Intake}$, the set of possible traces corresponding to $\widetilde{\mathbf{A}_{Intake}}$ is infinite (and dense) since it is not defined on the same clock as $\mathbf{A}_{Intake}$.

$\mathbf{A}_{Intake} = 1\_0\_1\_0\_1\_0\_1\_0\_1\_$ and $\widetilde{\mathbf{A}_{Intake}} = 0\_\_0\_\_0\_\_\_0\_\_0\_$ or $0111011\_01\_10\_\_101\ldots$
It is also worth noticing that the clock of $\widetilde{\mathbf{A}_{Intake}}$ (its reference in time) need not be explicitly related to or ordered with $\mathbf{A}_{Intake}$ or $\mathbf{G}_{Intake}$: it implicitly and partially relates to the *cam* clock.

Notice that our model of contract is agnostic as to a particular model of computation and accepts a generic domain of behaviors. Had we instead considered executable specifications, such as synchronous observers [22], it would have been more difficult to compositionally define the complementary of a proposition without referring to a global reference of time in the environment (e.g., a clock for the camshaft), hence abstracting every assumption or guarantee with respect to that global clock. This does not need to be the case in the present example. Beside its Boolean structure, our algebra supports the capability to compositionally refine contracts (without altering or abstracting individual properties) and accepts both synchronous and asynchronous specifications.

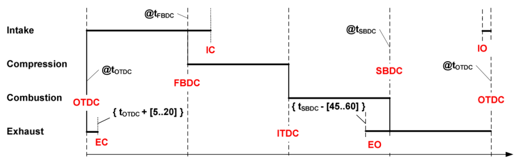

Figure 10. Model of 4-stroke engine cycle.

For instance, consider a more precise model of the 4-stroke engine found in [3](Figure 10). To additionally require that, while in the intake mode, the engine should reach the EC state (Exhaust closes) between 5 and 20 degrees, one will simply compose (greatest lower bound) the intake contract with the additional one:

$$A_{EC} = true\ when (4 < cam\ mod\ 360 < 21)\ G_{EC} = true\ when\ EC\ default\ false$$

Contracts can be used to express exclusion properties. For instance, when the engine is in the intake mode, one should not start compression.

$$\mathbf{A}_{excl} =_{\Delta} OTDC \quad \mathbf{G}_{excl} =_{\Delta} \neg FBDC$$

In addition to the above safety properties, contracts can also be used to express liveness properties. For instance, consider the protocol for starting the engine. A battery is used to initiate its rotation. When the engine has successfully started, the battery can be turned off. We can specify a contract to guarantee that engine cycles are properly counted. We write $cycle'$ for the next value of the variable $cycle$.

$$\mathbf{A}_{count} =_{\Delta} Exhaust \quad \mathbf{G}_{count} =_{\Delta} cycle' = cycle + 1$$

Another contract is defined to specify that the starter battery ($starter$) will eventually be turned off after a few cycles. We write $F()$ for the future property of LTL.

$$\mathbf{A}_{live} =_{\Delta} (cycle > 0) \quad \mathbf{G}_{live} =_{\Delta} F(\neg starter)$$

6 Related work

The use of contracts has been advocated for a long time in computer science [23,14] 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 context of software engineering, the notion of assertion-based contract has been adapted for a wide variety of languages and formalisms but the central notion of time and/or trace needed for reactive system design is not always taken into account. For instance, extensions of OCL with linear or branching-time temporal logics have been proposed in [28,11], focusing on the expressivity of the proposed constraint language (the way constraints may talk about the internals of classes and objects), and considering a fixed "sequence of states". This is a serious limitation for concurrent system design, as this sequence becomes an interleaving of that of individual objects.

In the theory of interface automata [2], the notion of interface offers benefits similar to our notion of contracts 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 allows better isolation between modules and hence favor compositionality. 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 [7], 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 complementary 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 [16], the authors show that the framework of interface automata may be embedded into that of modal I/O automata. [27] further develops this approach by considering modal specifications. 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 [7], 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 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.

In [22], a notion of synchronous contracts is proposed for the programming language Lustre. In this approach, contracts are executable specifications (synchronous 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 [8], a compositonal 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 [5] 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 [17], 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 the way 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 the ATL (Alternating-time Temporal Logic [4]) can directly be used to express assumptions about the context of a process and guarantees provided by that process.

7 Conclusion

Starting from the choice of 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-filters to formally characterize the logical device that filters behaviors from process much like the assumption and guarantee of a contract do. In our model, a process p fulfils its requirements (or satisfies) ($\mathbf{A}$,$\mathbf{G}$) if either it is rejected by $\mathbf{A}$ (it is then out of the scope of the contract ($\mathbf{A}$,$\mathbf{G}$)), or it is accepted by $\mathbf{G}$.

Our main results are that the structure of process-filters is a Boolean algebra and that the structure of contracts is a Heyting algebra, respectively. This rich structure allows for reasoning on contracts with great flexibility to abstract, refine and combine them. Moreover, contracts are not limited to expressing safety properties, as is the case in most related frameworks, but encompass the expression of liveness properties. This is all again due to the central notion of process-filter.

In the aim of assessing the generality and scalability of our approach, we are designing a module system based on the paradigm of contract for Signal and applying 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 module or 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] Abramsky S. and Jung A. Domain theory. In *Handbook of logic in computer science (vol. 3): semantic structures*, pages 1–168. Oxford University Press, 1997.

[2] Alfaro L. and Henzinger T. A. Interface automata. In *ESEC / SIGSOFT FSE*, pp. 109–120, 2001.

[3] André C., Mallet F., and Peraldi-Frati M.-A. A multi-form time approach to real-time system modeling. Research Report RR 2007-14-FR, I3S, 2007.

[4] Alur, R., Henzinger, T., and Kupferman, O. Alternating-time Temporal Logic. In *Journal of the ACM, v. 49*, ACM Press, 2002.

[5] Bartetzko D., Fischer C., Möller M., and Wehrheim H. Jass - Java with assertions. *In Havelund, K., Rosu, G. eds : Runtime Verification*, Volume 55 of ENTCS(1):91–108, 2001.

[6] Bell John L. Boolean Algebras and Distributive Lattices Treated Constructively, Math. Logic Quarterly 45, 1999.

[7] Benveniste A., Caillaud B., and Passerone R. A generic model of contracts for embedded systems. In *Research report N° 6214, INRIA-IRISA, S4 Team*, 2007.

[8] Broy, M. Compositional refinement of interactive systems. Journal of the ACM, v. 44. ACM Press, 1997.

[9] Doyen L., Henzinger T. A., Jobstmann B., and Petrov T., Interface theories with component reuse, In *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT'08*, 2008, pp. 79-88.

[10] Edwards S., Lavagno L., Lee E.A., and Sangiovanni-Vincentelli A. Design of Embedded Systems: Formal Models, Validation, and Synthesis. In *Proceedings of the IEEE, 85(3)*, pages 366–390, 1997.

[11] Flake S. and Mueller W. An OCL extension for realtime constraints. In *Lecture Notes in Computer Science 2263*, pp. 150–171, 2001.

[12] Glouche Y., Le Guernic P., Talpin J.-P., and Gautier T. A boolean algebra of contracts for logical assume-guarantee reasoning. Research Report RR 6570, INRIA, 2008.

[13] Glouche Y., Talpin J.-P., Le Guernic P., and Gautier T. A module language for typing by contracts. In *Nasa Formal Methods Symposium*, Springer, 2009.

[14] Hoare C.A.R. An axiomatic basis for computer programming. In *Communications of the ACM*, pp. 576–583, 1969.

[15] Kopetz H. Component-Based Design of Large Distributed Real-Time Systems. In *Control Engineering Practice, 6(1)*, pages 53–60, 1997.

[16] Larsen K.G., Nyman U., and Wasowski A. Modal I/O automata for interface and product line theories. In *Programmming Languages and Systems, 16th European Symposium on Programming, ESOP'07*, ser. Lecture Notes in Computer Science, vol. 4421. Springer, 2007, pp. 64-79.

[17] Leavens G. T., Baker A. L., and Ruby C. JML: A notation for detailed design. In Kilov H., Rumpe B., and Simmonds I., editors, *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Kluwer Academic Publishers, 1999.

[18] Lee E.A., Sangionanni-Vincentelli A., A framework for comparing models of computation. In IEEE transaction on computer-aided design, v. 17, 1998.

[19] Le Guernic P., Talpin J.-P., and Le Lann J.-C. Polychrony for system design. *Journal for Circuits, Systems and Computers*, Special Issue on Application Specific Hardware Design, 2003.

[20] 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, v. 10(4), 2000.

[21] 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, v. 41(1). Elsevier, 2001.

[22] Maraninchi F. and Morel L. Logical-time contracts for reactive embedded components. In *In 30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04, Rennes, France*, 2004.

[23] Martin A. and Lamport L. Composing specifications. In *ACM Trans. Program. Lang. Syst. 15*, pp. 73–132, 1993.

[24] Mazurkiewicz A. Basic notions of trace theory. In *Lecture Notes In Computer Science*, v. 354, pp. 285 - 363, 1989.

[25] Meyer B. *Object-Oriented Software Construction, Second Edition.* ISE Inc., Santa Barbara, 1997.

[26] Mitchell R. and McKim J. *Design by Contract, by Example*, Addison-Wesley, 2002.

[27] Raclet J.-B., Badouel E., Benveniste A., Caillaud B., Passerone R. Why are modalites good for Interface Theories? In *9th International Conference on Application of Concurrency*, 2009.

[28] Ziemann P. and Gogolla M. An extension of OCL with temporal logic. In *Critical Systems Development with UML-Proceedings of the UML'02 workshop, Technische Universität München, Institut für Informatik*, 2002.