A Module Language for Typing by Contracts

Yann Glouche* Jean-Pierre Talpin Paul Le Guernic Thierry Gautier
INRIA, Unité de Recherche Rennes-Bretagne-Atlantique, Campus de Beaulieu, Rennes, France

Abstract

Assume-guarantee reasoning is a popular and expressive paradigm for modular and compositional specification of programs. It is becoming a fundamental concept in some computer-aided design tools for embedded system design. In this paper, we elaborate foundations for contract-based embedded system design by proposing a general-purpose module language based on a Boolean algebra allowing to define contracts. In this framework, contracts are used to negociate the correctness of assumptions made on the definition of a component at the point where it is used and provides guarantees to its environment. We illustrate this presentation with the specification of a simplified 4-stroke engine model.

1 Introduction

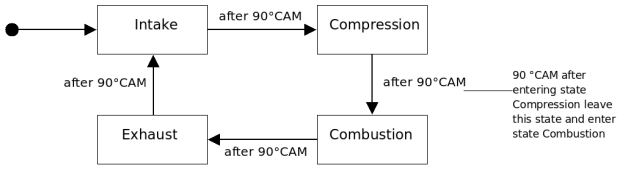
Methodological common sense for the design of large embedded architectures advises the validation of specifications as early as possible, and further advocates for an iterative validation of each refinement or modification made to any component of the initial specification, until the implementation of the system is finalized. As an example, in a cooperative industrial environment, designers often use and assemble components which have been developed by different suppliers. These components have to be provided with conditions of use and need to offer pre-validated guarantees on their function or service. The conditions of use and the service guarantee define a notion of *contract*. Design by contract, as advocated in [23], is now a common programming concept used in general-purpose languages such as 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 the paradigm of *contract* to define a component-based validation process in the context of an embedded software modeling framework. It consists of an algebraic framework, based on two simple concepts, enabling logical reasoning on contracts [11]. First, the assumptions and guarantees of a component are defined as devices called filters: assumptions filter the behaviors a component accepts and guarantees filter the behaviors a component provides. Filters form a Boolean algebra and contracts define a Heyting algebra. This yields a rich framework to abstract, refine, combine and normalize contracts. In this paper, 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 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.

Plan We start with a highlight on some key features of our module system by considering the specification of a simplified engine controler, Section 2. This example is used through the article to illustrate our approach. We give a short 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. Our main contribution, section 4, is to use this algebra as a foundation for the definition 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 the refinement mechanism of the language. We review related works, Section 6, before concluding, Section 7.

*Partially funded by the EADS Foundation.
E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 86-95

2 An example

We illustrate our approach by considering a simplified automotive application presented in [4]. We define contracts to characterize properties of a four-stroke engine controller. In this specification, the cyclic behavior of the engine is rendered by four successive phases: *Intake*, *Combustion*, *Compression*, and *Exhaust*. These phases are driven by a camshaft whose angle is measured in degrees. This unit of measure defines a discrete and symbolic time sampling in the system. The angle of the camshaft marks an occurrence of a clock tick (the *cam*) at which sensors are sampled and a reaction is triggered. For example, at 90°, the intake valve is closed and a transition to compressin mode is triggered.



In the module language, a specification is designated by the keyword **contract**. It defines a set of input and output variables subject to a contract. The interface defines the way the environment and the component interact through its variables. In addition, it embeds properties that are modeled by the composition of contracts. For instance, the specification of the intake mode of the engine controller could be defined by the assumption `0 <= (cam mod 360)< 90` and the guarantee `intake`. An implementation of the interface, designated by the keyword **process**, contains a compatible implementation of the contract assigning true to the signal `intake` when `0 <= (cam mod 360)< 90`.

```
module type IntakeType =
   contract
     input integer cam;
     output event intake;
     assume 0 <= (cam mod 360)< 90
     guarantee intake
   end;
```

```
Module IntakeMode : IntakeType =
   process
     input integer cam;
     output event intake;
     intake := true when cam mod 360<90
   end;
```

The specification of the properties we consider for the engine consists of four contracts. Each contract specifies a phase of the engine. It is associated to a position of the camshaft and triggers an event. Instead of specifying four separate contracts, we define them as four instances of a generic contract. To this end, we define an encapsulation mechanism to generically represent a class of interfaces or implementations sharing a common pattern of behavior up to that of the parameter. In the example of the engine, for instance, we have used a functor to parameterize it with respect to its angle position.

```
module type phase =
  functor (integer min, integer max)
    contract
      input integer cam ;
      output event trigger;
      assume min <= (cam mod 360) < max
      guarantee trigger
    end;
```

```
module type engine = contract
input integer cam ; output event intake,
        compression, combustion, exhaust;
phase (0, 90) (cam, intake)
and phase (90, 180) (cam, compression)
and phase (180, 270) (cam, combustion)
and phase (270, 360) (cam, exhaust)
end;
```

The generic contract, called `phase`, is parameterized with a trigger and with camshaft angle bounds. When the camshaft is within the specified angles, the engine controller activates the specified trigger. The contract of the engine is defined by the composition "**and**" of four applications of the generic phase contract. Each application defines a particular phase of the engine with its appropriate triggers and angles. The composition defines the greatest lower-bound of all four contracts. Each application of the phase functor produces a contract which is composed with the other ones in order to produce the expected contract. A component is hence viewed as a pair $M : I$ consisting of an implementation M that is typed by (or viewed as) an interface I of satisfiable contract. The semantics of the specification I, written ⟦I⟧, is a set of processes (in the sense of section 3) whose traces satisfy the contracts associated with I. The semantics ⟦M⟧ of the module M is the singleton containing the process ⟦IntakeMode⟧.

3 An algebra of contracts

A contract is viewed as a pair of logical devices that filter processes: the assumption **A** filters processes to select (accept or conversely reject) those of which that are to be asserted (accepted or conversely rejected) by the guarantee **G**. A process p fulfils (**A**,**G**) requirements (satisfies (**A**,**G**)) if either it is rejected by **A** (it is then out of the scope of (**A**,**G**) contract), or it is accepted by **A** and by **G**. We have chosen to represent a filtering device (such as **A** or **G**) by a *process-filter* which is the set of processes that it "accepts" (i.e., that it contains). We give more details of the algebra of contracts, including examples, additional properties and formal proofs in a technical report [11]. In the remainder, we only define the mathematical objects that are relevant to the presentation of our module system. They consist of two relations in the contract algebra, noted $\preccurlyeq$ for refinement and $\Downarrow$ for composition. In fact, our module system can be seen as a domain-specific language that syntactically reflects the structure of our contract algebra.

For $\mathbf{X}$ a nonempty finite set of variables, a behavior is a function $b : \mathbf{X} \to \mathscr{D}$ where $\mathscr{D}$ is a set of values (that may be traces), and a *process* p is a nonempty set of behaviors defined on $\mathbf{X}$. We denote by $\Omega = \{\emptyset\}$ the unique process defined on the empty set of variables: it has a single behavior that is the empty behavior. The set of processes is noted $\mathbb{P}$; it is extented to $\mathbb{P}^\star$ by adding the *empty "process"* noted $\mho = \emptyset$. A *process-filter* **R** is the set of processes that satisfy a given property on a set of variables $\mathbf{X}$; notice that if a process p is in **R** so are all processes defined on supersets of $\mathbf{X}$, whose sub-behaviors restricted to $\mathbf{X}$ are behaviors in p; moreover if a process p is in **R**, so are all nonempty subsets of p. By convention, $\{\mho\}$ is a process-filter. We define an order relation $\sqsubseteq$ on the set of process-filters Φ extended by $\{\mho\}$ and establish that $(\Phi,\sqsubseteq)$ is a lattice. Also, $(\Phi,\sqsubseteq)$ is a Boolean algebra with $\mathbb{P}^\star$ as *1*, $\{\mho\}$ as *0*. The complementary of **R** is noted $\widetilde{\mathbf{R}}$. The conjunction $\mathbf{R} \sqcap \mathbf{S}$ of two process-filters **R** and **S** is the greatest process-filter $\mathbf{T} = \mathbf{R} \sqcap \mathbf{S}$ that accepts all processes whose behaviors are at once behaviors in some process accepted by **R** and behaviors in some process accepted by **S**. The process-filter disjunction $\mathbf{R} \sqcup \mathbf{S}$ of two process-filters **R** and **S** is the smallest process-filter $\mathbf{T} = \mathbf{R} \sqcup \mathbf{S}$ that accepts all processes whose behaviors are at once behaviors in some process accepted by **R** or behaviors in some process accepted by **S**. A *contract* $\mathbf{C} = (\mathbf{A},\mathbf{G})$ is a pair of process-filters and $\mathbb{C} = \Phi \times \Phi$ is the set of contracts. The refinement relation ($\preccurlyeq$) is a partial order on contracts. ($\mathbb{C}$, $\preccurlyeq$) is a distributive lattice of supremum ($\{\mho\}$,$\mathbb{P}^\star$) and infimum ($\mathbb{P}^\star$,$\{\mho\}$): it is a Heyting algebra. We define the nominal contract $p_{\succcurlyeq}$ of a process p as the contract that assumes any behavior and guarantees all the possible behaviors of p. Two contracts $\mathbf{C}_1$ and $\mathbf{C}_2$ have a greatest lower bound $\mathbf{C}_1 \Downarrow \mathbf{C}_2$ and a least upper bound $\mathbf{C}_1 \Uparrow \mathbf{C}_2$. The greatest lower bound $\mathbf{C} = (\underset{\sim}{\mathbf{A}},\mathbf{G})$ of two contracts $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ and $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ is defined by: $\mathbf{A} = \mathbf{A}_1 \sqcup \mathbf{A}_2$ and $\mathbf{G} = ((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2))$.

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 is instantiated to the context of the synchronous language Signal, yet 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.

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

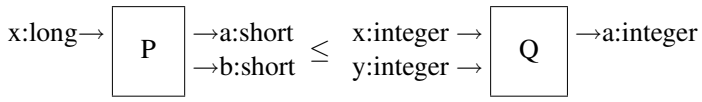
4.2 A type system for contracts and processes

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 τ is π for the type of a process and γ 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. The role of the typing hypothesis Γ is to hold an association $x : T$ of names to types (we write $\Gamma(x)$ the type of name x in Γ). The role of the typing constraints Σ is to register inferred refinement relations of the form $C \preceq D$.

$S, T ::= t \mid \tau(I,C) \mid S \times T \mid \Lambda(x : S).T$ (type) $\quad \tau ::= \gamma \mid \pi$ (kind) $\quad \Gamma ::= \emptyset \mid \Gamma \cup x : T \quad \Sigma ::= \emptyset \mid \Sigma \cup C \preceq D$ (context)

4.3 Contract refinement

We wish to define a subtyping relation on types t to extend the refinement relation of the contract algebra to the type algebra. In that aim, we wish to apply the subtyping principle $S \leq T$ 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. For example, consider 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, such that P refines Q.Then the type of a module M encapuslating P is a subtype of a module N encapsulating Q. M can replace N.



4.4 Subtyping as refinement

Accordingly, we say that s is a subtype of t under the hypothesis Σ, written $\Sigma \supset s \leq t$ iff Σ contains (or implies) the relation $s \leq t$. The inductive definition of the subtyping relation $\leq$ starts with the subtyping axioms for datatypes. Then, it is elaborated with algebraic rules, the rules for declarations and, finally, one rule for each kind of module type. In particular, a module type $S = \tau(I,C)$ is a subtype of $T = \tau(J,D)$, written $\Sigma \supset S \leq T$, iff the inputs in J subtype those in I, if the outputs in I subtype those in J, and if the contract C refines D. In the rule for functors, we write $V[y/x]$ for substituting the name x by y in V (we assume that y does not occur in V).

$$T \leq T \quad \textbf{event} \leq \textbf{boolean} \quad \textbf{short} \leq \textbf{integer} \leq \textbf{long}$$

$$\frac{\Sigma \supset I \leq J \quad x \notin vars(I)}{\Sigma \supset I \leq (x : \textbf{input}\ b) \cup J} \qquad \frac{\Sigma \supset I \leq J \quad x \notin vars(J)}{\Sigma \supset (x : \textbf{output}\ b) \cup I \leq J}$$

$$\frac{\Sigma \supset b \leq c}{\Sigma \supset \textbf{input}\ c \leq \textbf{input}\ b} \qquad \frac{\Sigma \supset b \leq c}{\Sigma \supset \textbf{output}\ b \leq \textbf{output}\ c}$$

$$\frac{\tau \leq \gamma \quad \pi \leq \tau \quad \Sigma \supset I \leq J \quad (C \preceq D) \in \Sigma \quad \tau \leq \tau'}{\Sigma \supset \tau(I,C) \leq \tau'(J,D)}$$

$$\frac{\Sigma \supset S \leq T \quad \Sigma \supset T \leq U}{\Sigma \supset S \leq U} \qquad \frac{\Sigma \supset I \leq J \quad \Sigma \supset S \leq T}{\Sigma \supset (x : S) \cup I \leq (x : T) \cup J}$$

$$\frac{\Sigma \supset S \leq U \quad T \leq V}{\Sigma \supset S \times T \leq U \times V} \qquad \frac{\Sigma \supset U \leq S \quad \Sigma \supset T \leq V[x/y]}{\Sigma \supset \Lambda(x : S).T \leq \Lambda(y : U).V}$$

Alternatively, we can interpret the relation $\Sigma \supset C \preceq D$ as a mean to register the refinement constraint between C and D in Σ. It corresponds to a proof obligation in the target language whose meaning is defined by the semantic relation $[\![C]\!] \preceq [\![D]\!]$ in the contract algebra, and whose validity may for instance be proved by model checking. The set Σ of refinement relations $C \preceq D$ is obtained by the structural decomposition of subtyping relations of the form $s \leq t$ into either elementary axioms, e.g. **event** $\leq$ **boolean**, or proof obligations $C \preceq D$.

4.5 Greatest-lower and lowest-upper bounds

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 and least upper bound of two contracts are extended to module type by induction on the structure of types. The intersection and union operators are extended to combine the set of input and output declarations of a module. The side condition $^{(*)}$ is that $x \notin dom(J)$.

$$\begin{array}{rl}
\tau(I,C) \sqcap \tau(J,D) = \tau(I \sqcap J, C \Downarrow D) & (I \cup (x : S)) \sqcap (J \cup (x : T)) = (I \sqcap J) \cup (x : S \sqcap T) \\
S \times T \sqcap U \times V = (S \sqcap U) \times (T \sqcap V) & (I \cup (x : S)) \sqcup (J \cup (x : T)) = (I \sqcup J) \cup (x : S \sqcup T) \\
\Lambda(x : S).T \sqcap \Lambda(y : U).V = \Lambda(x : (S \sqcup U)).(T \sqcap V[y/x]) & (I \cup (x : \textbf{input}\ b)) \sqcap J = (I \sqcap J)^{(*)} \quad \emptyset \sqcap J = \emptyset \\
\tau(I,C) \sqcup \tau(J,D) = \tau(I \sqcup J, C \Uparrow D) & (I \cup (x : \textbf{output}\ b)) \sqcap J = (I \sqcap J) \cup (x : \textbf{output}\ b)^{(*)} \\
S \times T \sqcup U \times V = (S \sqcup U) \times (T \sqcup V) & (I \cup (x : \textbf{input}\ b)) \sqcup J = (I \sqcup J) \cup (x : \textbf{input}\ b)^{(*)} \\
\Lambda(x : S).T \sqcup \Lambda(y : U).V = \Lambda(x : (S \sqcap U)).(T \sqcup V[y/x]) & (I \cup (x : \textbf{output}\ b)) \sqcup J = (I \sqcup J)^{(*)} \quad \emptyset \sqcup J = J
\end{array}$$

4.6 Type inference in the module language

Our aim is to check the correctness of program construction. Hence, type inference shall produce a consistent type assignment to contract and process names in the module language and generate a proof obligation in the form of an observer function [12] (used for describing some properties) in the target programming language. The type inference system is defined by the sequent $\Gamma/\Sigma \vdash exp$ where Γ is the typing environment, Σ the typing constraints and exp is an expression of the module language. It embeds the type system of the target language: we assume that the relation $\Gamma \vdash p$ tells that p is well-typed in the target language under the hypothesis Γ. The sequent establishes a structural correspondence between expressions and types. It is defined by induction on the structure of expressions in a similar manner as that proposed for Standard ML in [17]. The operator $\tau \cdot T$ promotes the type T to the kind τ. It is defined by $\tau \cdot (\tau'(I,C)) = \tau \cdot (I,C)$ and by $\tau \cdot (\Lambda(x : S).T) = \Lambda(x : S).(\tau \cdot T)$. It is used to check that the definition of a module type is a contract and to promote the type of a module.

$$\frac{\Gamma/\Sigma \vdash s : S \quad \Gamma/\Sigma \vdash t : T}{\Gamma/\Sigma \vdash s \times t : S \times T} \qquad \frac{\Gamma/\Sigma \vdash s : S \quad \Gamma/\Sigma \vdash t : T}{\Gamma/\Sigma \vdash \Lambda(x : s).t : \Lambda(x : S).T} \qquad \frac{\Gamma/\Sigma \vdash t : T}{\Gamma/\Sigma \vdash t\,x : (x : T)} \qquad \frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma/\Sigma \vdash dec' : J}{\Gamma/\Sigma \vdash dec, dec' : I \cup J}$$

$$\frac{\Gamma/\Sigma \vdash p \quad \Gamma/\Sigma \vdash q}{\Gamma/\Sigma \vdash \textbf{assume}\ p\ \textbf{guarantee}\ q : (p,q)} \qquad \frac{\Gamma/\Sigma \vdash ag : C \quad \Gamma/\Sigma \vdash ag' : D}{\Gamma/\Sigma \vdash ag\ \textbf{and}\ ag' : C \Downarrow D} \qquad \frac{\Gamma/\Sigma \vdash x : \gamma(x_1 : T_1 .. x_n : T_n, C) \quad (\Gamma(y_i) \leq T_i)_{i=1}^n}{\Gamma/\Sigma \vdash x(y_{1..n}) : C[y_i/x_i]_{i=1}^n}$$

$$\frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma \cup I \vdash ag : C}{\Gamma/\Sigma \vdash \textbf{contract}\ dec; ag\ \textbf{end} : \gamma(I,C)} \qquad \frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma \cup I \vdash p}{\Gamma/\Sigma \vdash \textbf{process}\ dec; p\ \textbf{end} : \pi(I,((),p))} \qquad \frac{\Gamma/\Sigma \vdash exp : S \quad \Gamma/\Sigma \vdash exp' : T}{\Gamma/\Sigma \vdash (exp\ \textbf{and}\ exp') : S \sqcap T}$$

$$\frac{\Gamma(x) = T}{\Gamma/\Sigma \vdash x : T} \qquad \frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma \cup I/\Sigma \vdash exp : T}{\Gamma/\Sigma \vdash \textbf{functor}\ (dec)\ exp\ \textbf{end} : \Lambda I.T} \qquad \frac{\Gamma/\Sigma \vdash x : \Lambda(z : S).T \quad \Gamma/\Sigma \vdash y : U \quad \Sigma \subset U \leq S}{\Gamma/\Sigma \vdash x(y) : T[y/z]}$$

$$\frac{\Gamma/\Sigma \vdash exp : T \quad \gamma \cdot T \leq T}{\Gamma/\Sigma \vdash \textbf{module type}\ x = exp : (x : T)} \qquad \frac{\Gamma/\Sigma \vdash exp : S \quad \Gamma/\Sigma \vdash t : T \quad \Sigma \subset S \leq T}{\Gamma/\Sigma \vdash \textbf{module}\ x : t = exp : (x : \pi \cdot T)} \qquad \frac{\Gamma/\Sigma \vdash def : I \quad \Gamma \cup I/\Sigma \vdash exp : T}{\Gamma/\Sigma \vdash \textbf{let}\ def\ \textbf{in}\ exp : T}$$

4.7 Correctness

The correctness of our module system is stated by showing that a program is well-typed if the constraints implied by Γ/Σ are consistent. We say that the environment ρ is well-typed with Γ/Σ, written $\rho : \Gamma/\Sigma$, iff all definitions in ρ are well-typed with Γ under the constraints Σ. Notice that, in this implementation of the type system, the generated constraints Σ define the proof obligations that are needed for checking that the specification *exp* is well-typed. To establish this theorem we define the semantics $[\![exp]\!]_\rho$ of a term *exp* in the module system by induction on the structure of *exp*. It is a set of processes p of the model of computation that satisfy their specifications.

Theorem 1. *If* $\rho : \Gamma/\Sigma$ *for a satisfiable* Σ *and* $\Gamma/\Sigma \vdash exp : T$ *then* $[\![exp]\!]_\rho \subseteq [\![T]\!]_\rho$.

Proof sketch The proof of Theorem 1 is by induction on the structure of expressions *exp* (just as for a regular type system). For each syntactic category in the grammar of *exp*, it considers a satisfiable set of constraints Σ, a well-typed environment $\rho : \Gamma/\Sigma$ and assumes a proof tree for $\Gamma/\Sigma \vdash exp : T$. Induction hypotheses are made to assert that the sub-expressions $exp_{i\in 1..n}$ of *exp* satisfy the expected sub-goals $[\![exp_i]\!]_\rho \subseteq [\![T_i]\!]_\rho$ in order to deduce that $[\![exp]\!]_\rho \subseteq [\![T]\!]_\rho$ by definition of the denotational semantics.

5 Discussion

We illustrate the distinctive features of our contract algebra by reconsidering the specification of the four-stroke engine and its translation into observers in the target language of our choice: the multi-clocked synchronous (or polychronous) data-flow language Signal [7]. The separation of environmental assumptions and system guarantees is facilitated by the (unpaired) possibility to naturally express the complementary of a process-filter. Had we used automata to model $\mathbf{A}_{Intake}$, as in most of the related work, it would probably have been more difficult to model the engine not being in the intake mode: this would have required the definition of the complementary of an automaton and, most importantly, this could not have been done compositionaly. In our algebra, the complementary of the intake property is simply defined by $\widetilde{\mathbf{A}_{Intake}} = cam\ modulo\ 360^\circ \geq 90$. In general, the generic structure of observers specified in contracts will find a direct instance and compositional translation into the synchronous multi-clocked model of computation of Signal [16]. Indeed, a subtlety of the Signal language is that an observer not only talks about the value, true or false, of a signal, but also about its status, present or absent. Thanks to its encoding in three-valuated logic, an event (e.g. intake is present and true) can directly be associated with its complementary (e.g. intake is absent or false) without separating the status (the clock) and the value (the stream). Hence, the signal A_{intake} is present and true iff cam signal is present and its value is between 0 and 89 degrees.

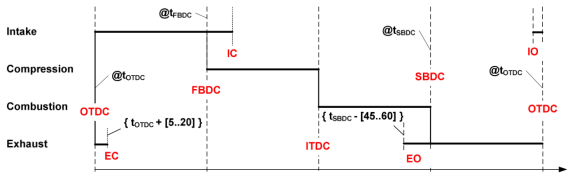
$\mathbf{A}_{intake} = true\ when\ (0 \leq cam\ modulo\ 360 < 90)$ $\quad$ $\mathbf{G}_{intake} = (true\ when\ intake)\ default\ false$

The complementary of these assumptions is simply defined to be true iff the cam is absent or out of bounds: $\widetilde{\mathbf{A}_{Intake}} = (false\ when\ A_{intake})\ default\ true$. 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. Had we used a stricly synchronous model of computation, as in [18], it would have been necessary to know the clock of the system in order to define that of the complementary by a sample of it. Beside its Boolean structure, which allows for logical reasoning and normalization of contracts, our algebra supports the capability to compositionally refine contracts. For instance, consider a more precise model of the 4-stroke engine found

in [4]. To additionally require the engine to reach the EC state (Exhaust closes) between 5 and 20 degrees, while in the intake mode, one will simply compose the initial contract with an additional constraint with : $A_{EC} = true\ when\ (5 <= cam\ modulo\ 360 < 21)$ and $G_{EC} = true\ when\ EC\ default\ false$. Similarly, event OTDC (Overlap Top Dead Center) occurs at the beginning of the cycle. The instant t_{OTDC} is a time observation of this event and the occurrence of the event EC is constrained by $\{t_{OTDC} + [5..20]\}$, hence $t_{OTDC} + 5 \leq t_{EC} \leq t_{OTDC} + 20$. We shall refine the specification of the engine to incorporate these additional constraints. This is done as follows:




```
module type better_engine = engine
  and contract
        input integer CAM ;
        output event EC, IC, EO, IO;
            phase(5,20)(CAM,EC)
        and phase(130,150)(CAM,IC)
        and phase(210,225)(CAM,EO)
        and phase(344,350)(CAM,IO)
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 above engine map. Nonetheless, an implementation of the above engine 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 engine controller to verify that it implements the expected engine map. 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. In Signal, process `phase` consists of one equation that is executed when its output `trigger` is needed (its clock is active). If the signal `cam` is present and within the specified bounds `min-max`, then `trigger` is present. Otherwise, `cam` is absent or simply out of bounds, the trigger is absent. The signals `cam` and `trigger` are respectively the input and output of the process whereas the names `min-max` are functor parameters of the process.

```
process phase = {integer min, max;} (? integer cam; ! event trigger;)
                (| trigger := when min<=(cam mod 360)<max |);
```

In the process `engine`, four instances of the `phase` equation are defined to specify the output signals that are relevant to a specific phase of the engine. Notice that these signals are not, *a priori*, synchronized one with the other: they are concurrent. This is done to favor a compositional specification of the system. Refinement precisely allows to iteratively build a sequentially executable specification by, e.g., synchronizing the signals `OTD`, `FBD`, `ITD` and `SBD`. This choice in favor of compositional modeling (polychrony) versus executability (synchrony) allows us to handle the additional specification of the *better engine* in a compositional manner, showing that the Signal language and our module system share the same concurrent/compositional design philosophy.

```
process engine =
 (? integer CAM; ! event OTD, FBD, ...
 (| OTD := phase {  0,  90} (CAM)
  | FBD := phase { 90, 180} (CAM)
  | ITD := phase {180, 270} (CAM)
  | SBD := phase {270, 360} (CAM)
  |);
```

```
process betterengine =
  (? boolean CAM ! event OTDC, FBDC, ITDC, ...
  (| (OTDC, FBDC, ITDC, SBDC) := engine (CAM)
   | EC := phase {  5,  20} (CAM)
   | IC := phase {130, 150} (CAM)
   | EO := phase {210, 225} (CAM)
   | IO := phase {344, 350} (CAM) |);
```

Contracts can be used to express exclusion properties. For instance, when the engine is in the intake mode, one should not start compression. We have $\mathbf{A}_{excl}{=}OTDC$ and $\mathbf{G}_{excl}{=}\neg FBDC$.

```
module type exclude = contract output event intake, compression;
                      assume intake guarantee (not compression default intake)
                      end;
```

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 define a contract to guarantee that when the ignit button is pushed, the starter eventually stops.

```
module type StarterOff = contract input event ignit; output boolean starter;
                           assume ignit guarantee eventually not starter
                         end;
```

6 Implementation

The module system described in this paper, embedding data-flow equations defined in syntax, has been implemented in Java. 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 [20], 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 [19] 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.

7 Related work

The use of contracts has been advocated for a long time in computer science [21, 13] and, more recently, has been successfully applied in object-oriented software engineering [22]. 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. The *assumption* specifies hypothesis which has to be satisfied by the component in order to provide the *guarantee*.

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 [25, 10], 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 [1], 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 allow 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, the filtering equivalence relation (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 the context of the EC project Speeds [6], a model of assume-guarantee contracts is proposed which extends the notion of interface automata with modalities and multiple views. This consists of labelling transitions that may be fired and other that must. By contrast to our domain-theoretical approach, the Speeds approach starts from an abstracted notion of modules whose only structure is a partial order of refinement. The proposed approach also leaves the role of variables in contracts unspecified, at the cost of some algebraic relations such as inclusion.

In [18], 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 system). 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 stream-processing data-flow language. The proposed type system allows reasoning on properties of programs abstracted by input-output types and causality graphs. In a similar way as Broy et al., we aim at using our module system to associate programs with compilation contracts, consisting of the necessary (and sufficient) synchronization and scheduling relations for modular code generation.

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 [15], 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.

Another example is the language Synergy [9], that combines two paradigms: object-oriented modeling for robust and flexible design, and synchronous execution for precise modeling of reactive behavior. In [14], a method of encapsulation based on the object-oriented paradigm and behavioral inheritance for the description of synchronous models is proposed. [24] describes an ML-like model for decomposing complex synchronous structures in parameterizable modules.

Our main contribution is to define a type system starting from a domain theoretical algebra for assume-guarantee reasoning consisting of 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 [3]) can directly be used to express assumptions about the context of a process and guarantees provided by that process.

8 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-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) (**A**,**G**) if either it is rejected by **A** (i.e., if **A** represents assumptions on the environment, they are not satisfied for p) or it is accepted by **G**. The structure of process-filters is a Boolean algebra and allows for reasoning on contracts with great flexibility to abstract, refine and combine them. In addition

to that, and unlike the related work, 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. This is all again due to the central notion of process-filter. 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] Alfaro L. and Henzinger T. A. Interface automata. In *ESEC / SIGSOFT FSE*, pp. 109–120, 2001.

[2] Alpern, B., Schneider, F. Proving boolean combinations of deterministic properties. In Proceedings of the Second Symposium on Logic in Computer Science. IEEE Press, 1987.

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

[4] André C., Mallet F., and Peraldi-Frati M-A. A multiform time approach to real-time system modeling; application to an automotive system. In *International Symposium on Industrial Embedded Systems*, pp. 234–241, 2007.

[5] Bartetzko D., Fischer C., Mueller 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] Benveniste A., Caillaud B., and Passerone R. A generic model of contracts for embedded systems. In *INRIA RR n. 6214*, 2007.

[7] Benveniste A., Le Guernic P., and Jacquemot C. Synchronous programming with events and relation: the SIGNAL language and its semantics. In *Science of Computer Programming*, volume v.16, 1991.

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

[9] Budde R., Poigné A., and Sylla K.-H. Synergy - an object-oriented synchronous language. *In Havelund, K., Rosu, G. eds: Runtime Verification*, Volume 153 of Electronic Notes in Theoretical Computer Science(1):99–115, 2006.

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

[11] 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.

[12] Halbwachs N. and Raymond P. Validation of Synchronous Reactive Systems: From Formal Verification to Automatic Testing. In *Advances in Computing Science-Proceedings of the ASIAN'99 conference*, 1999.

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

[14] Kerbœuf M. and Talpin J.-P. Encapsulation and behavioural inheritance in a synchronous model of computation for embedded system services adaptation. In *Journal of languages, algebra and progr. Special issue on process algebra and syst. arch*. Elsevier, 2004.

[15] 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.

[16] 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.

[17] Leroy X. A modular module system. Journal of Functional Programming, v. 10(3). Cambridge Academic Press, 2000.

[18] 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.

[19] 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.

[20] 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.

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

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

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

[24] Nowak D., Talpin J.-P., Gautier T., and Le Guernic P. An ML-like module system for the synchronous language SIGNAL. In *European Conference on Parallel Processing (EUROPAR'97)*, August 1997.

[25] Ziemann P. and Gogolla M. An extension of OCL with temporal logic. In *Critical Systems Development with UML-Proceedings of the UML'02 workshop*, 2002.