A Generic Model of Contracts for Embedded Systems

Albert Benveniste, Benoit Caillaud, Roberto Passerone

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Generic Model of Contracts for Embedded Systems

Albert Benveniste, Benoît Caillaud, Roberto Passerone

N° 6214

June 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6214--FR+ENG

A Generic Model of Contracts for Embedded Systems*

Albert Benveniste†, Benoît Caillaud‡, Roberto Passerone§

Thème COM — Systèmes communicants
Projet S4

Rapport de recherche n° 6214 — June 2007 — 18 pages

Abstract: We present the mathematical foundations of the contract-based model developed in the framework of the SPEEDS project. SPEEDS aims at developing methods and tools to support "speculative design", a design methodology in which distributed designers develop different aspects of the overall system, in a concurrent but controlled way. Our generic mathematical model of contract supports this style of development. This is achieved by focusing on behaviors, by supporting the notion of "rich component" where diverse (functional and non-functional) aspects of the system can be considered and combined, by representing rich components via their set of associated contracts, and by formalizing the whole process of component composition.

Key-words: system design, component based design, contract based design, assume-guarantee reasoning

* This research has been developed in the framework of the European SPEEDS integrated project number 033471.

† IRISA / INRIA, Rennes, France, albert.benveniste@irisa.fr
‡ IRISA / INRIA, Rennes, France, benoit.caillaud@irisa.fr
§ University of Trento & PARADES EEIG, Trento & Rome, Italy, roberto.passerone@unitn.it

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Un modele générique de contrats pour les systèmes embarqués

Résumé : Ce rapport présente les fondements mathématiques du modèle de contrats conçu dans le cadre du projet SPEEDS. L'objectif du projet SPEEDS est de développer les outils et les méthodes supportant un "processus de conception spćulatif", dans lequel différentes équipes de conception peuvent contribuer à la conception d'un système de fçon concurrente, mais néanmoins controllée. Le modèle de contrats concerne les comportements du système projeté et permet une modélisation de celui-ci par assemblage de "composants riches", dont les différents aspects comportementaux sont décrits pas des ensembles contrats, regroupés par "points de vues".

Mots-clés : concepton système, conception par composants, conception par contrats, raisonnement hypiothèse-garantie

1 Introduction

Several industrial sectors involving complex embedded systems design have recently experienced drastic moves in their organization—aerospace and automotive being typical examples. Initially organized around large, vertically integrated companies supporting most of the design in house, these sectors were restructured in the 80's due to the emergence of sizeable competitive suppliers. OEMs performed system design and integration by importing entire subsystems from suppliers. This, however, shifted a significant portion of the value to the suppliers, and eventually contributed to late errors that caused delays and excessive additional cost during the system integration phase.

In the last decade, these industrial sectors went through a profound reorganization in an attempt by OEMs to recover value from the supply chain, by focusing on those parts of the design at the core of their competitive advantage. The rest of the system was instead centered around standard platforms that could be developed and shared by otherwise competitors. Examples of this trend are AUTOSAR in the automotive industry [1], and Integrated Modular Avionics (IMA) in aerospace [2]. This new organization requires extensive virtual prototyping and design space exploration, where component or subsystem specification and integration occur at different phases of the design, including at the early ones [3].

Component based development has emerged as the technology of choice to address the challenges that result from this paradigm shift. In the particular context of (safety critical) embedded systems with complex OEM/supplier chains, the following distinguishing features must be addressed. First, the need for high quality, zero defect, software systems calls for techniques in which component specification and integration is supported by clean mathematics that encompasse both static and *dynamic* semantics—this means that the behavior of components and their composition, and not just their port and type interface, must be mathematically defined. Second, system design includes various aspects—functional, timeliness, safety and fault tolerance, etc.—involving different teams with different skills using heterogeneous techniques and tools. Third, since the structure of the supply chain is highly distributed, a precise separation of responsibilities between its different actors must be ensured. This is addressed by relying on contracts. Following [4] a contract is a component model that sets forth the assumptions under which the component may be used by its environment, and the corresponding promises that are guaranteed under such correct use.

The semantic foundations that we present in this paper are designed to support this methodology by addressing the above three issues. At its basis, the model is a language-based abstraction where composition is by intersection. This basic model can then be instantiated to cover functional, timeliness, safety, and dependability requirements performed across all system design levels. No particular model of computation and communication is enforced, and continuous time dynamics such as those needed in physical system modeling is supported as well. On top of the basic model, we build the notion of a contract, which is central to our methodology, by distinguishing between assumptions and promises. This paper focuses on developing a generic compositional theory of contracts, providing relations of contract satisfaction and refinement called dominance, and the derivation of operators for the correct construction of complete

systems. In addition to traditional parallel composition, and to enable formal multi-viewpoint analysis, our model includes boolean meet and join operators that compute conjunction and disjunction of contracts. We also introduce a new operator, called fusion, that combines composition and conjunction to compute the least specific contract that satisfies a set of specifications, while at the same time taking their interaction into account.

The paper is organized as follows. The principles of our approach are presented in Section 2. Contracts and implementations are introduced in Section 3 and corresponding operations are studied in Section 4. The concept of rich component is formalized in Section 5, by introducing the contracts attached to it. In Section 6 we formalize the concept of designer responsibilities through the notion of controlled/uncontrolled port and we refine our theory of contracts accordingly. How we encompass the different viewpoints is sketched in Section 7 and related work is discussed in Section 8.

2 Principles of Assume/Guarantee Reasoning

The main element of our semantic model is a *Heterogeneous Rich Component*, or simply a component. A component consists of an *interface*, its *expected behavior*, and, optionally, one or more *implementations*. The interface is a set of ports and flows, used by the component to communicate with the rest of the system and with the environment. The expected behavior is described by one or several *assumption/promise* pairs, called *contracts*. Contracts can be combined together using three composition operators: greatest lower bound, parallel composition and fusion. The greatest lower bound is used to compose contracts referring to the same component and which use only variables and flows visible from the environment. Parallel composition is used to compute the contract resulting from the composition of several components. Fusion generalizes these two operators, and is capable of handling all cases. In particular, it is used to compose contracts whenever the greatest lower bound and parallel composition operators are inappropriate, for instance when contracts share local variables or flows. Thus, fusion is the implicit composition of contracts, whenever more than one contract is attached to a component. Implementations may be attached to a component, and are usually expressed as extended state machines, or as host tool models. We define several relations between components, contracts and implementations.

- The *compatibility* relation relates sets of components. A set of components are *incompatible* whenever for all environments, at least one of the assumption of at least one component is violated.
- Contract *dominance* relates assumptions and promises of two contracts. A contract dominates another when it has weaker assumptions and stronger promises.
- *Satisfaction* relates implementations to contracts. An implementation satisfies a contract whenever its behavior, modulo the assumptions, are consistent with the promises.
- *Refinement* relates implementations. An implementation refines another whenever it has fewer behaviors.

Throughout this paper we shall need an abstract notion of "assertion". The only facts we need to know about assertions can be summarized as follows:

- Each assertion E possesses a set of *ports* and a set of *variables* that are the vehicle for interaction.
- An assertion is identified with the set of runs it accepts. A run assigns a history to each variable and port of the assertion. We assume that a proper notion of "complement" for an assertion E is available, denoted by $\neg E$.
- When seen as sets of runs, assertions compose by intersection—note that such an operation is monotonic w.r.t. inclusion of sets of runs. When performing this composition, we assume that the appropriate inverse projections have been performed to equalize the sets of ports and variables. Products are equivalently denoted by $E_1 \times E_2$ or $E_1 \cap E_2$.

3 Rich Components, Contracts, Implementations

Definition 1 (Implementation). *An* implementation *is simply an assertion, that is, a set of runs.*

We denote implementations by the symbol M (for "machine"). Implementations are ordered according to the runs they contain. An implementation M *refines* an implementation M', written $M \preceq M'$ if and only if M and M' are defined over the same set of ports and variables, and

$$M \subseteq M'.$$

Products preserve implementation refinement.

A *contract* says that under certain assumptions, behaviors are guaranteed to be confined within a certain set.

Definition 2 (Contract). *A* contract C *is a pair* (A, G)*, where* A*, the* assumption*, and* G*, the* promise*, are assertions over the same alphabet.*

Whenever convenient, we shall denote the assumption and promise of contract C by A_C and G_C. The interpretation of a contract is made precise by the following definition.

Definition 3 (Satisfaction). *An implementation* M *satisfies a contract* $C = (A, G)$*, written* $M \models C$*, if and only if*

$$M \cap A \subseteq G.$$

Satisfaction can be checked using the following equivalent formulas, where $\neg A$ denotes the set of all runs that are not runs of A:

$$M \models C \iff M \subseteq G \cup \neg A \iff M \cap (A \cap \neg G) = \emptyset$$

There exists a unique maximal implementation satisfying a contract C, namely:

$$M_C = G \cup \neg A \tag{1}$$

Definition 4 (Rich Component). *A* rich component *is a tuple*

$$RC \quad = \quad \big(X, \{C\}, [M]\big) \tag{2}$$

In (2), X is the name *of the rich component, $\{C\}$ is a (possibly empty) set of contracts, and $[M]$ is an (optional) implementation such that $[M] \models \{C\}$, where the meaning of the latter property is postponed to Definition 16.*

Canonical forms Note that M_C is to be interpreted as the implication $A \Rightarrow G$. We have that $M \models (A, G)$, if and only if $M \models (A, M_C)$, if and only if $M \subseteq M_C$. Say that contract $C = (A, G)$ is in *canonical form* when $G = M_C$, or, equivalently, when $\neg A \subseteq G$ or when $\neg G \subseteq A$. Thus, every contract has an equivalent contract in canonical form, which can be obtained by replacing G with M_C. Hence, working with contracts in canonical form does not limit expressiveness. The operation of computing the canonical form is well defined, since the maximal implementation is unique, and it is idempotent.

$$\begin{array}{c} \textit{In the following, we assume that} \\ \textit{all contracts are in canonical form.} \end{array} \tag{3}$$

This assumption serves two purposes: (i) To have a unique representation of contracts, considered up to equivalence. (ii) To simplify the definition of contract composition operators.

Example 1 (Running example: control/monitoring unit). Throughout this paper, we develop the system of Figure 1 to illustrate our notion of contract and its use. It consists of a control unit interacting with a monitoring unit. The system is subject to two independent faults, f_1 for the control unit, and f_2 for the monitoring unit. The nominal behavior of the system (when $f_1 = \mathtt{F}$) is that

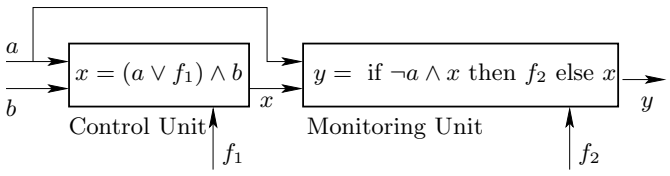

Figure 1: Running example: control/monitoring system.

it should deliver $y = a \wedge b$ at its output. When safe ($f_2 = \mathtt{F}$), the monitoring unit ensures that, if the control unit gets faulty ($f_1 = \mathtt{T}$), the overall system is shut down ($y = \mathtt{F}$) unless $a = \mathtt{T}$ holds. Thus the overall system requirement is to maintain the Top Level Exception TLE $= \neg a \wedge y$ false. This TLE may, however, get violated if the monitoring unit gets faulty too ($f_2 = \mathtt{T}$). These requirements are summarized by the two contracts C, for the nominal mode, and C' for the exception mode:

$$\begin{array}{rcll} C & = & (\,\neg f_1\,,\, y = a \wedge b\,) & : \text{ nominal mode} \\ C' & = & (\,\neg f_2\,,\, \neg\mathrm{TLE}\,) & : \text{ exception mode} \end{array} \tag{4}$$

This separation of concerns into nominal and exception mode is similar to the separation of viewpoints (functional, timed, safety, etc) when handling components via their contracts. ◇

4 Operations on contracts

4.0.1 Boolean algebra

As usual, it will be extremely useful to have an algebra of contracts, providing ways of expressing more complex contracts from simpler ones. The following relation of dominance formalizes substituability for contracts and induces a boolean algebra of contracts, which provides such a logic of contracts.

Definition 5 (Dominance). *Say that contract $C = (A, G)$ dominates contract $C' = (A', G')$, written $C \preceq C'$, if and only if $A \supseteq A'$ and $G \subseteq G'$.*

Dominance amounts to relaxing assumptions and reinforcing promises. Note that $C \preceq C'$ and $C' \preceq C$ together imply $C = C'$. Hence, dominance is a partial order relation. Furthermore,

$$C \preceq C' \quad \Longrightarrow \quad M_C \models C' \tag{5}$$

but the converse is not true. Property (5) implies the following result:

Lemma 6. *If $M \models C$ and $C \preceq C'$, then $M \models C'$.*

The following theorem defines the boolean algebra over contracts, implied by $\preceq$. Its proof is straightforward and left to the reader.

Theorem 7 (Boolean algebra of contracts). *Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts. Then, the greatest lower bound of C_1 and C_2, written $C = C_1 \sqcap C_2$, is given by $C = (A, G)$ where $A = A_1 \cup A_2$ and $G = G_1 \cap G_2$. Note that the so defined pair (A, G) is in canonical form.*

Similarly, the least upper bound of C_1 and C_2, written $C = C_1 \sqcup C_2$, is given by $C' = (A', G')$ where $A = A_1 \cap A_2$ and $G = G_1 \cup G_2$. Note that the so defined pair (A, G) is in canonical form.

The minimal and maximal contracts are $\bot = (\mathcal{R}, \emptyset)$ and $\top = (\emptyset, \mathcal{R})$, respectively, where $\mathcal{R}$ denotes the set of all runs.

Finally, the complement of C is the contract $\neg C$ such that $\neg C = (\neg A, \neg G)$; it satisfies $\neg C \sqcap C = \bot$ and $\neg C \sqcup C = \top$.

Example 2 (Running example: greatest lower bound). The two contracts of (4) represent two viewpoints attached to a same component, corresponding to the nominal and exception modes, respectively. These two contracts involve the same set of ports. Combining them is by computing their greatest lower bound. Putting these two contracts in canonical form and then taking their greatest lower bound yields:

$$C \sqcap C' = \left(\neg(f_1 \wedge f_2)\,, \left[\begin{array}{ll} & \neg f_1 \Rightarrow y = a \wedge b \\ \wedge & \\ & \neg f_2 \Rightarrow \neg\mathrm{TLE} \end{array} \right] \right)$$

This contract assumes that no double failure occurs. Its promise is the conjunction of the promises of C and C'. Expanding the promise of this global contract leads to a cumbersome formula, hardly understandable to the user, so we discard it. ◇

4.0.2 Parallel composition

Contract composition formalizes how contracts attached to different rich components should be combined to represent a single, compound, rich component. Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts. First, composing these two contracts amounts to composing their promises. Regarding assumptions, however, the situation is more subtle. Suppose first that the two contracts possess disjoint sets of ports and variables. Intuitively, the assumptions of the composite should be simply the conjunction of the assumptions of the rich components, since the environment should satisfy all the assumptions. In general, however, part of the assumptions A_1 will be already satisfied by composing C_1 with C_2, acting as a partial environment for C_1. Therefore, G_2 can contribute to relaxing the assumptions A_1. And vice-versa. Whence the following definition:

Definition 8 (Parallel composition of contracts). *Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts. Define $C_1 \,||\, C_2$ to be the contract $C = (A, G)$ such that:*

$$\begin{aligned} A &= (A_1 \cap A_2) \cup \neg(G_1 \cap G_2), \\ G &= G_1 \cap G_2. \end{aligned}$$

Note that the so defined contract is in canonical form.

The following result expresses the compositionality of the implementation relation:

Lemma 9. *$M_1 \models C_1$ and $M_2 \models C_2$ together imply $M_1 \times M_2 \models C_1 \,||\, C_2$.*

Proof: The assumption of the lemma means that $M_i \subseteq M_{C_i}$, for $i = 1, 2$. Since the two contracts are in canonical form, we have $M_{C_i} = G_i$ and the result follows directly from Definition 8. □The following lemma relates greatest lower bound and parallel composition, it relies on the fact that we work with contracts in canonical form:

Lemma 10. *For any two contracts, $C_1 \sqcap C_2 \preceq C_1 \parallel C_2$.*

Proof: Both sides of this relation possess identical promises. Thus the only thing to prove relates to the assumptions thereof. From Definition 8 and Theorem 7, the assumption of $C_1 \sqcap C_2$ is equal to $A_1 \cup A_2$, whereas the assumption of $C_1 \parallel C_2$ is equal to $(A_1 \cap A_2) \cup \neg(G_1 \cap G_2)$. Since the two contracts are in canonical form, we have $\neg G_i \subseteq A_i, i = 1, 2$, and thus $\neg(G_1 \cap G_2) = \neg G_1 \cup \neg G_2 \subseteq A_1 \cup A_2$. Therefore, the assumption of $C_1 \parallel C_2$ is contained in $A_1 \cup A_2$, which is the assumption of $C_1 \sqcap C_2$. This proves the lemma. □

Example 3 (Running example: compositional reasoning). In Example 2, we have shown how to combine the two nominal and exception viewpoints, for the overall system of Figure 1. The system further decomposes into a control and monitoring unit. We would like to associate contracts to each of these components, for each of their viewpoint. Composing these contracts, we should recover the system's overall contract.

Since the system is the parallel composition of control and monitoring units, we may reasonably expect that the parallel composition of contracts, for each of

these components, should be used. However, we are also combining viewpoints for these two components and this sould be performed by the greatest lower bound. So, which is the correct answer? The new notion of contract *fusion* we shall introduce in the following section will provided the adequate answer. Prior to introducing this notion, we need to investigate what it means to eliminate ports in contracts. ◇

4.0.3 Eliminating ports in contracts

Elimination in contracts requires handling assumptions and promises differently.

Definition 11 (Elimination). *Let $C = (A, G)$ be a contract and let p be any port. Define the* elimination of p in C by:

$$[C]_p \quad = \quad (\forall p\, A, \exists p\, G)$$

where A and G are seen as predicates.

Note that we do not require that p be a port of C. Definition 11 is motivated by the following lemma:

Lemma 12. *We have $C \preceq [C]_p$. Furthermore, let M be an implementation such that $M \models C$ and p is not a port of M. Then, $M \models [C]_p$.*

Proof: By definition, $M \models C$ implies $M \cap A \subseteq G$. Eliminating p, with $\forall$ on the left hand side and $\exists$ on the right hand side, yields $[\forall p\,(M \cap A)] \subseteq [\exists p\, G]$ and the lemma follows from the fact that $\forall p\,(M \cap A) = M \cap (\forall p\, A)$ if p is not a port of M. □The following lemma relates elimination and greatest lower bounds:

Lemma 13. *For any two contracts C_1 and C_2 and any port p, we have:*

$$[C_1 \sqcap C_2]_p \quad \preceq \quad [C_1]_p \sqcap [C_2]_p \tag{6}$$
$$[C_1 \parallel C_2]_p \quad \preceq \quad [C_1]_p \parallel [C_2]_p \tag{7}$$

Proof: We have $\forall p\,(A_1 \cup A_2) \supseteq (\forall p\, A_1 \cup \forall p\, A_2)$ and $\exists p\,(G_1 \cap G_2) \subseteq (\exists p\, G_1 \cap \exists p\, G_2)$, which proves (6) as well as the promise part of (7). Regarding the assumption part of (7), we need to prove

$$A_{[C_1 \parallel C_2]_p} \quad \supseteq \quad A_{([C_1]_p \parallel [C_2]_p)} \tag{8}$$

where

$$\begin{aligned} A_{[C_1 \parallel C_2]_p} &= \forall p\,\big((A_1 \cap A_2) \cup \neg(G_1 \cap G_2)\big) \\ A_{([C_1]_p \parallel [C_2]_p)} &= (\forall p\, A_1 \cap \forall p\, A_2) \cup \neg(\exists p\, G_1 \cap \exists p\, G_2) \end{aligned}$$

We have $\forall p\,((A_1 \cap A_2) \cup \neg(G_1 \cap G_2)) \supseteq (\forall p\,(A_1 \cap A_2)) \cup (\forall p \neg(G_1 \cap G_2)) = (\forall p\, A_1 \cap \forall p\, A_2) \cup \neg(\exists p\,(G_1 \cap G_2)) \supseteq (\forall p\, A_1 \cap \forall p\, A_2) \cup \neg(\exists p\, G_1 \cap \exists p\, G_2)$. Which proves (8) and the lemma. □Elimination trivially extends to finite sets of ports, we denote it by $[C]_P$, where P is the considered set of ports.

5 Set of contracts associated to a rich component

We are now ready to address the case of synchronizing viewpoints when local ports are shared between viewpoints. More precisely, we shall formally define what it means to consider a set of contracts associated to a same rich component.

Definition 14 (Fusion). *Let $(C_i)_{i\in I}$ be a finite set of contracts and Q a finite set of ports. We define* the fusion of $(C_i)_{i\in I}$ with respect to Q *by*

$$[\![(C_i)_{i\in I}]\!]_Q \quad = \quad \bigsqcap_{J\subseteq I} \big[\, \|_{j\in J} C_j \,\big]_Q \tag{9}$$

where J ranges over the set of all subsets of I.

The following particular cases of Definition 14 are of interest:

Lemma 15.

1. *When $Q = \emptyset$, the fusion reduces to the greatest lower bound:*

$$[\![(C_i)_{i\in I}]\!]_\emptyset \quad = \quad \sqcap_{i\in I} C_i \tag{10}$$

In particular, $M \models [\![(C_i)_{i\in I}]\!]_\emptyset$ implies $M \models C_i$ for each $i \in I$.

2. *Assume that, for $i = 1, 2$:*

$$A_i \quad \supseteq \quad G_1 \cap G_2 \tag{11}$$

holds. Then:

$$[\![(C_i)_{i\in\{1,2\}}]\!]_\emptyset \quad = \quad C_1 \parallel C_2 \tag{12}$$

3. *Assume that, for $i = 1, 2$:*

$$\forall Q\,(A_i \cup \neg G) \quad \supseteq \quad \forall Q\,(A_1 \cup A_2) \tag{13}$$

holds, where $C_1 \parallel C_2 = (A, G)$. Then:

$$[\![(C_i)_{i\in\{1,2\}}]\!]_Q \quad = \quad [C_1 \parallel C_2]_Q \tag{14}$$

Condition (11) expresses that each rich component is a valid environment for the other rich components; in other words, the two contracts are attached to two rich components that together constitute a valid closed system. Condition (13) expresses that the restriction to Q of each component is a valid environment for the restriction to Q of the other component. This situation corresponds to two rich components interacting through ports belonging to Q, which are subsequently hidden from outside. **Proof:** We successively prove the three statements.

Statement 1 results immediately from Lemma 10.

To prove (12) in Statement 2, note that the two expressions only differ by their assumptions, since the promises of greatest lower bound and parallel composition are identical. For the assumptions, let $C_1 \parallel C_2 = (A, G)$ and $C_1 \sqcap C_2 = (A', G)$. We have $G = G_1 \cap G_2$, $A = (A_1 \cap A_2) \cup \neg G$, and $A' =$

$(A_1 \cup A_2)$. From (11) we get $\neg G \supseteq \neg(A_1 \cap A_2)$. Therefore, $A = (A_1 \cap A_2) \cup \neg G \supseteq (A_1 \cap A_2) \cup \neg(A_1 \cap A_2) = \mathcal{R} \supseteq A'$.

Regarding Statement 3, (13) implies $\forall Q\,((A_1 \cap A_2) \cup \neg G) \supseteq \forall Q\,(A_1 \cup A_2)$. Whence (14) follows. □The lesson is that fusion boils down to parallel composition for contracts attached to two different sub-components of a same compound component, whereas contracts attached to a same component and involving the same set of ports fuse via the operation of greatest lower bound. The general case lies in between and is given by formula (9). Finally, the various relations that we have established between greatest lower bound, parallel composition, and elimination, allows us to simplify the actual evaluation of the fusion in general. Corresponding heuristics to guide this remain to be developed.

Definition 4 for rich components can now be completed.

Definition 16 (Rich Component, completed). *Let* $RC = \big(X, \{C\}, [M]\big)$ *be a rich component. Say that*

$$[M] \models \{C\} \quad \textit{iff} \quad [M] \models [\![(C_i)_{i\in I}]\!]_Q,$$

where I *indexes set* $\{C\}$*, and set* Q *collects the ports of* $\{C\}$ *that are local to* RC*.*

Example 4 (Running example: fusion of contracts). We shall perform a composability study for the two contracts C and C', and then for their fusion $[\![C, C']\!]$.

Study of C Consider the following two contracts, for the control and monitoring unit, respectively $C_1 = (\,\neg f_1\,,\,[x = a \wedge b]\,)$ and $C_2 = (\,\neg\varphi\,,\;y = x\,)$, where $\varphi = \neg a \wedge x$. Contract C_1 states that, if not faulty, the control unit guarantees that $\neg\varphi$ holds, i.e., invariant $a \vee \neg x$ holds. Contract C_2 states that the monitoring unit guarantees that, if not faulty, $y = x$ holds unless φ does not hold. Putting these two contracts in canonical form and then computing their fusion yields

$$\begin{aligned}
C_1 &= (\,\neg f_1\,,\,\neg f_1 \Rightarrow [x = a \wedge b]\,)\\
C_2 &= (\,\neg\varphi\,,\,\neg\varphi \Rightarrow\; y = x\,)\\
[C_1]_x &= (\,\neg f_1\,,\,\mathtt{T}\,)\\
[C_2]_x &= (\,\mathtt{F}\,,\,\mathtt{T}\,)\\
[C_1 \parallel C_2]_x &= (\,\neg f_1 \wedge P\,,\,\neg f_1 \Rightarrow [y = a \wedge b]\,)
\end{aligned}$$

where P is some predicate (which we don't care about), from which we obtain, provided that C is put in canonical form,

$$[\![C_1, C_2]\!]_x = [C_1]_x \sqcap [C_2]_x \sqcap [C_1 \parallel C_2]_x = [C_1 \parallel C_2]_x = C$$

Study of C' Now, let us focus on the other contract C', by proposing the following two local contracts, for the control and monitoring unit, respectively $C'_1 = (\,\mathtt{F}\,,\,\mathtt{T}\,)$, and $C'_2 = (\,\neg f_2\,,\,[y = x \wedge a]\,)$. The first contract is trivial, and the second one states the invariant promised if the monitoring unit is not faulty. We first have

$$[C'_1]_x = C'_1 \text{ and } [C'_2]_x = (\,\neg f_2\,,\,\neg f_2 \Rightarrow \neg\mathrm{TLE}\,). \tag{15}$$

Second, $G'_1 \cap G'_2 = \neg f_2 \Rightarrow [y = x \wedge a]$, whence $\exists x : (G'_1 \cap G'_2) = \neg f_2 \Rightarrow \neg\text{TLE}$. Next, $(A'_1 \cap A'_2) \cup \neg(G'_1 \cap G'_2) = \neg(G'_1 \cap G'_2)$, which equals $\neg(\neg f_2 \Rightarrow [y = x \wedge a])$, whence

$$\begin{aligned} \forall x : ((A'_1 \cap A'_2) \cup \neg(G'_1 \cap G'_2)) &= \neg(\exists x : (G'_1 \cap G'_2)) \\ &= \text{TLE} \wedge \neg f_2 \end{aligned} \tag{16}$$

Finally, (15)–(16) together prove that $[\![C'_1, C'_2]\!]_x = C'$.

Study of $[\![C, C']\!]$ The remarkable point is that composability works both across components, and viewpoints/modes, i.e., we have $[\![C, C']\!] = [\![C_1, C_2, C'_1, C'_2]\!]_x$. ◇

6 The asymmetric role of ports

So far we have ignored the role of ports and the corresponding splitting of responsibilities between the implementation and its environment, see the discussion in the introduction. Such a splitting of responsibilities avoids the competition between environment and implementation in setting the value of ports and variables.

Intuitively, an implementation can only provide promises on the value of the ports it controls. On ports controlled by the environment, instead, it may only declare assumptions. Therefore, we will distinguish between two kinds of ports for implementations and contracts: those that are *controlled* and those that are *uncontrolled*. The latter property is formalized via the following notion of *receptiveness*:

Definition 17 (Receptiveness). *For E an assertion, and $P' \subseteq P$ a subset of its ports, E is said to be* P'-receptive *if and only if for all runs σ' restricted to ports belonging to P', there exists a run in σ of E such that σ' and σ coincide over P'.*

In words, E accepts any history offered to the subset P' of its ports. Note that:

Lemma 18. *If E is P'-receptive, then so is $E \cup E'$ for any E' having no extra ports or variables than those of E.*

In some cases, different viewpoints associated with a same rich component need to interact through some common ports. This motivates providing a scope for ports, by partitioning them into ports that are *visible* (outside the underlying component) and ports that are *local* (to the underlying component).

Definition 19 (Profile). *A* profile *is a 4-tuple $\pi = (\mathbf{vis}, \mathbf{loc}, \mathbf{u}, \mathbf{c})$, partitioning P as*

$$\begin{array}{rcccl} P & = & \mathbf{vis} \uplus \mathbf{loc} & = & \{\mathit{visible}\} \uplus \{\mathit{local}\} \\ P & = & \mathbf{u} \uplus \mathbf{c} & = & \{\mathit{uncontrolled}\} \uplus \{\mathit{controlled}\} \end{array}$$

We are now ready to refine our theory of contracts by taking the asymmetric role of ports into account.

Definition 20 (Implementation). *An* implementation *is a pair* $M = (\pi, E)$*, where* $\pi = (\mathbf{vis}, \mathbf{loc}, \mathbf{u}, \mathbf{c})$ *is a* profile *over a set* P *of ports, and* E *is a* $\mathbf{u}$*-receptive assertion over* P*.*

The last requirement formalizes the fact that an implementation has no control over the values of ports set by the environment. Implementations refine as follows:

Definition 21 (Implementation Refinement). *For* M *and* M' *two implementations, say that* M *refines* M'*, written* $M \preceq M'$*, if and only if* $\pi = \pi'$ *and* $E \subseteq E'$*.*

In defining parallel composition for implementations, we need to take into account controlled ports. Each implementation is responsible for its set of controlled ports, and, in our theory, such responsibility should not be shared. This motivates the following definition for our parallel composition of implementations associated with different rich components (whence the requirement $\mathbf{loc}_1 \cap \mathbf{loc}_2 = \emptyset$ in this definition):

Definition 22 (Parallel composition of implementations). *Let* M_1 *and* M_2 *be two implementations such that* $\mathbf{loc}_1 \cap \mathbf{loc}_2 = \emptyset$*. Then,* $M = M_1 \,||\, M_2$ *is defined if and only if* $\mathbf{c}_1 \cap \mathbf{c}_2 = \emptyset$*. In that case,* $E = E_1 \times E_2$*, and:*

$$\mathbf{vis} = \mathbf{vis}_1 \cup \mathbf{vis}_2 \qquad \mathbf{c} = \mathbf{c}_1 \cup \mathbf{c}_2$$
$$\mathbf{loc} = \mathbf{loc}_1 \cup \mathbf{loc}_2 \qquad \mathbf{u} = (\mathbf{u}_1 \cup \mathbf{u}_2) - (\mathbf{c}_1 \cup \mathbf{c}_2)$$

Theorem 23. *Implementation composition is monotonic relative to implementation refinement.*

Proof: Since profiles refine via identity, this results boils down to the well known monotonicity w.r.t. sets of runs. □

Definition 24 (Contract). *A* contract *is a triple* $C = (\pi, A, G)$*, where* $\pi = (\mathbf{vis}, \mathbf{loc}, \mathbf{u}, \mathbf{c})$ *is a profile over a set* P *of ports, and* A *and* G *are two assertions over* P*, respectively called the* assumptions *and* promises *of* C*.* C *is called* consistent *if* G *is* $\mathbf{u}$*-receptive, and* compatible *if* A *if* $\mathbf{c}$*-receptive.*

As pointed out in (3), *contracts are in canonical form,* meaning that $G \supseteq \neg A$. If this is not the case, we simply replace G by its most permissive version $G \cup \neg A$, which cannot per se break consistency, thanks to Lemma 18. The sets A and G are not required to be receptive. However, if G is not $\mathbf{u}$-receptive, then the promises constrain the uncontrolled ports of the contract. This is against our policy of separation of responsibilities, since we stated that uncontrolled ports should remain entirely under the responsibility of the environment. Corresponding contracts are therefore called *inconsistent.*

Definition 25 (Satisfaction). *An implementation* M satisfies *contract* C*, written* $M \models C$*, iff* $\pi_M = \pi_C$ *and* $E_M \subseteq G_C$*.*

By Lemma 18, if contract C is consistent, then $M_C = G_C \cup \neg A_C$ is still the *maximal implementation* satisfying C. We now turn to the relation of dominance and its consequences.

Definition 26 (Contract Dominance). *A contract $C = (\pi, A, G)$ dominates a contract $C' = (\pi, A', G')$, written $C \preceq C'$, if and only if $\pi = \pi'$, $A \supseteq A'$, and $G \subseteq G'$.*

Theorem 27 (Boolean algebra of contracts). *Let $C_1 = (\pi_1, A_1, G_1)$ and $C_2 = (\pi_2, A_2, G_2)$ be contracts such that $\pi_1 = \pi_2 = \pi$. Then $C =(\pi, A_1 \cup A_2, G_1 \cap G_2)$ is the greatest lower bound of C_1 and C_2, written $C = C_1 \sqcap C_2$. Similarly, $C' = (\pi, A_1 \cap A_2, G_1 \cup G_2)$ is the least upper bound of C_1 and C_2, written $C = C_1 \sqcap C_2$. Finally, the complement of $C = (\pi, A, G)$ is $\neg C = (\pi, \neg A, \neg G)$.*

Proof: This is a direct consequence of Theorem 7 □

Finally, it remains to define the parallel composition of contracts. Having done this we can directly borrow the definition 14 of fusion, for contracts enhanced with profiles.

Definition 28 (Parallel composition of contracts). *Let $C_1 = (\pi_1, A_1, G_1)$ and $C_2 = (\pi_2, A_2, G_2)$ be contracts. The parallel composition, or product, of C_1 and C_2, written $C = C_1 \,||\, C_2$, is defined if and only if $\mathbf{c}_1 \cap \mathbf{c}_2 = \emptyset$, and in that case is the contract $C = (\pi, A, G)$ defined by:*

$$\begin{array}{rcl} \mathbf{vis} & = & \mathbf{vis}_1 \cup \mathbf{vis}_2, \\ \mathbf{loc} & = & (\mathbf{loc}_1 \cup \mathbf{loc}_2) - (\mathbf{vis}_1 \cup \mathbf{vis}_2), \\ \mathbf{c} & = & \mathbf{c}_1 \cup \mathbf{c}_2, \\ \mathbf{u} & = & (\mathbf{u}_1 \cup \mathbf{u}_2) - (\mathbf{c}_1 \cup \mathbf{c}_2), \\ A & = & (A_1 \cap A_2) \cup \neg(G_1 \cap G_2), \\ G & = & G_1 \cap G_2. \end{array}$$

Unlike Definition 22, we do not require here that $\mathbf{loc}_1 \cap \mathbf{loc}_2 = \emptyset$. The reason is that we wish to encompass the composition of different viewpoints attached to a same rich component. (For contracts attached to different rich components, however, we do have $\mathbf{loc}_1 \cap \mathbf{loc}_2 = \emptyset$.)

With parallel composition, we can formalize the notion of contract compatibility. Recall that a contract is *compatible* whenever A is $\mathbf{c}$-receptive. If not, then there exists a sequence of values on the controlled ports that are refused by all acceptable environments. However, by our definition of satisfaction, implementations are allowed to output such sequence. Unreceptiveness, in this case, implies that a hypothetical environment that wished to prevent a violation of the assumptions should actually prevent the behavior altogether, something it cannot do since the port is controlled by the contract. Therefore, unreceptive assumptions denote the existence of an incompatibility internal to the contract, that cannot be avoided by any environment. This justifies the following definition.

Definition 29 (Compatibility). *Two contracts C_1 and C_2 are* compatible *if and only the assumption of their parallel composition is receptive with resepct to the controlled ports.*

Assumptions may become unreceptive as a result of a parallel composition even if they are not so individually. This is because the set of controlled ports after a composition is strictly larger than before the composition. In particular, ports that were uncontrolled may become controlled, because they are controlled by the other contract.

Note that consistency and compatibility may not be preserved by Boolean operations. For example, one obtains an inconsistent contract when taking the greatest lower bound of two contracts, one of which promises that certain behaviors will never occur in response to a certain input, while the other promises that the remaining behaviors will not occur in response to the same input. Both contracts have legal responses to the input, but their intersection is empty, thus making the combination unreceptive. In this case, inconsistency is due to two contracts making inconsistent promises.

7 Addressing Multiple Viewpoints

An important question is: can our abstract notion of "assertion" encompass the different functional and non-functional viewpoints of system design? Since assertions are just sets of runs, we can, in particular, accomodate hybrid automata following [5]. So seemingly, we can in particular support functional, timeliness, safety, as these can be modeled by specific subclasses of hybrid automata.

A closer investigation reveals that we need to deal with classes of models that are stable under parallel composition (defined by intersection), union, and complement. Taking complements is a delicate issue: hybrid automata are not closed under complementation; in fact, no model class is closed under complementation beyond deterministic automata. To account for this fact, various countermeasures can be considered.

First, the designer has the choice to specify either E or its complement $\neg E$ (e.g., by considering observers). However, the parallel composition of contracts requires manipulating both E and its complement $\neg E$, which is the embarrasing case. To get compact formulas, our theory was developed using canonical forms for contracts, systematically. Not enforcing canonical forms provides room for flexibility in the representation of contracts, which can be used to avoid manipulating both E and $\neg E$ at the same time. A second idea is to redefine an assertion as a *pair* $(E, \bar{E})$, where $\bar{E}$ is an approximate complement of E, e.g., involving some abstraction. In doing so, one of the two characteristic properties of complements, namely $E \cap \bar{E} = \emptyset$ or $E \cup \bar{E} = \top$, do not hold. However, either necessary of sufficient conditions for contract dominance can be given. The above techniques are the subject of ongoing work and will be reported elsewhere.

8 Related Work

The notion of contract derives from the theory of abstract data types, first suggested by Meyer in the context of the programming language Eiffel [6]. In his work, Meyer introduces *preconditions* and *postconditions* as assertions for the methods of a class, and *invariants* for the class itself. Preconditions correspond to the assumptions under which the method operates, while postconditions express the promises at method termination, provided that the assumptions are satisfied. Invariants must be true at all states of the class regardless of any assumption. To guarantee safe substitutability, a subclass is only allowed to weaken the assumptions and to strengthen the promises.

Similar ideas were in fact, already present in earlier work by Dill, although phrased in less explicit terms [7]. Dill proposes an asynchronous model based on sets of sequences and parallel composition (trace structures). Behaviors (traces) can be either accepted as *successes*, or rejected as *failures*. The failures, which are still possible behaviors of the system, correspond to unacceptable inputs from the environment, and are therefore the complement of the preconditions. Safe substitutability is expressed as trace containment between the successes and failures of the specification and the implementation. Wolf later extended the same technique to a discrete synchronous model [8]. More recently, De Alfaro and Henzinger have proposed Interface Automata which are similar to synchronous trace structures, where failures are implicitly all the traces that are not accepted by an automaton representing the component [9]. Composition is defined on automata, rather than on traces, and requires a procedure to restrict the state space that is equivalent to the process called autofailure manifestation of Dill and Wolf. A more general approach along the lines proposed by Dill and Wolf is the work by Negulescu with Process Spaces [10], and by Passerone with Agent Algebra [11], both of which extend the algebraic approach to generic behaviors introduced by Burch [12].

Our notion of contract supports *speculative design* in which distributed teams develop partial designs concurrently and synchronize by relying on the notions of rich component [4] and associated contracts. We define assumptions and promises in terms of behaviors, and use parallel composition as the main operator for decomposing a design. This choice is justified by the reactive nature of embedded software, and by the increasing use of component models that support structured concurrency. We developed our theory on the basis of assertions, i.e., languages of generic "runs". To achieve the generality of a (mathematical) metamodel we have complemented this by developing a concrete model for such assertions, that encompasses the different viewpoints of the design [13].

In our approach, behaviors are decomponsed into assumptions and promises, as in Process Spaces, a representation that is more intuitive than, albeit equivalent to, the one based on the successes and failures of asynchronous trace structures. Unlike Process Spaces, however, we explicitly consider inputs and outputs, which we generalize to the concept of controlled and uncontrolled signals. This distinction is essential in our framework to determine the exact role and responsibilities of users and suppliers of components. This is concretized in our framework by a notion of compatibility which depends critically on the particular partition of the signals into inputs and outputs. We also extend the use of receptiveness of asynchronous trace structures, which is absent in Process Spaces, to define formally the condition of compatibility of components for open systems.

Our refinement relation between contracts, which we call *dominance* to distinguish it from refinement between implementations of the contracts, follows the usual scheme of weakening the assumption and strengthening the guarantees. The order induces boolean operators of conjunction and disjunction, which resembles those of asynchronous trace structures and Process Spaces. In addition, we also define a new *fusion* operator that combines the operation of composition and conjunction for a set of contracts. This operator is introduced to make it easier for the user to express the interaction between contracts related to different viewpoints of a component.

The model that we present in this paper is based on execution traces, and is therefore inherently limited to representing linear time properties. The branching structure of a process whose semantics is expressed in our model is thus abstracted, and the exact state in which non-deterministic choices are taken is lost. Despite this, the equivalence relation that is induced by our notion of dominance between contracts is more distinguishing than the traditional trace containment used when executions are not represented as pairs (assumptions, promises). This was already observed by Dill, with the classic example of the vending machine [7], see also Brookes et al. on refusal sets [14]. There, every accepted sequence of actions is complemented by the set of possible *refusals*, i.e., by the set of actions that may not be accepted after executing that particular sequence. Equivalence is then defined as equality of sequences with their refusal sets. Under these definitions, it is shown that the resulting equivalence is stronger than trace equivalence (equality of trace sets), but weaker than observation equivalence [15, 16]. A precise characterization of the relationships with our model, in particular with regard to the notion of composition, is deferred to future work.

9 Conclusion

We have presented mathematical foundations for the contract-based model developed in the framework of the SPEEDS project. Our generic mathematical model of contract supports "speculative design". This is achieved by focusing on behaviors, by supporting the notion of rich component where diverse (functional and non-functional) aspects of the system can be considered and combined, by representing rich components via their set of associated contracts, and by formalizing the whole process of component composition through the general mechanism of contract fusion. These foundations support the Heterogeneous Rich Component (HRC) metamodel under development in SPEEDS [13]. Future work includes the development of effective algorithms to handle contracts, coping with the problems raised by complementation.

Acknowledgements: This research has been developed in the framework of the SPEEDS integrated European project number 033471. We would like to thank all SPEEDS project participants for the fruitful discussions we had with them, and for the suggestions they made to improve the research report.

References

[1] W. Damm, "Embedded system development for automotive applications: trends and challenges." in *EMSOFT*, S. L. Min and W. Yi, Eds. ACM, 2006, p. 1.

[2] H. Butz, "The Airbus approach to open Integrated Modular Avionics (IMA): technology, functions, industrial processes and future development road map," in *International Workshop on Aircraft System Technologies, Hamburg*, March 2007.

[3] A. Sangiovanni-Vincentelli, "Reasoning about the trends and challenges of system level design," *Proc. of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.

[4] W. Damm, "Controlling speculative design processes using rich component models." in *ACSD*. IEEE Computer Society, 2005, pp. 118–119.

[5] T. A. Henzinger, "The theory of hybrid automata." in *LICS*, 1996, pp. 278–292.

[6] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, October 1992.

[7] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, ser. ACM Distinguished Dissertations. MIT Press, 1989.

[8] E. S. Wolf, "Hierarchical models of synchronous circuits for formal verification and substitution," Ph.D. dissertation, Department of Computer Science, Stanford University, October 1995.

[9] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, 2001, pp. 109–120.

[10] R. Negulescu, "Process spaces and the formal verification of asynchronous circuits," Ph.D. dissertation, University of Waterloo, Canada, 1998.

[11] R. Passerone, "Semantic foundations for heterogeneous systems," Ph.D. dissertation, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720, May 2004.

[12] J. R. Burch, "Trace algebra for automatic verification of real-time concurrent systems," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, August 1992.

[13] SPEEDS. http://www.speeds.eu.com.

[14] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, "A theory of communicating sequential processes," *Journal of the Association for Computing Machinery*, vol. 31, no. 3, pp. 560–599, July 1984.

[15] J. Engelfriet, "Determinacy → (observation equivalence = trace equivalence)," *Theoretical Computer Science*, vol. 36, pp. 21–25, 1985.

[16] S. D. Brookes, "On the relationship of CCS and CSP," in 10^{th} *ICALP*, ser. Lecture Notes in Computer Science, vol. 154. Springer-Verlag, 1983.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399