Coherent Extension, Composition, and Merging Operators in Contract Models for System Design

ROBERTO PASSERONE, University of Trento
ÍÑIGO ÍNCER ROMEO and ALBERTO L. SANGIOVANNI-VINCENTELLI,
University of California, Berkeley

Contract models have been proposed to promote and facilitate reuse and distributed development. In this paper, we cast contract models into a coherent formalism used to derive general results about the properties of their operators. We study several extensions of the basic model, including the distinction between weak and strong assumptions and maximality of the specification. We then analyze the disjunction and conjunction operators, and show how they can be broken up into a sequence of simpler operations. This leads to the definition of a new contract viewpoint merging operator, which better captures the design intent in contrast to the more traditional conjunction. The adjoint operation, which we call separation, can be used to re-partition the specification into different viewpoints. We show the symmetries of these operations with respect to composition and quotient.

CCS Concepts: • **Computer systems organization** → **Embedded hardware**; **Embedded software**; • **Theory of computation** → *Logic and verification*;

Additional Key Words and Phrases: Interface, contract, merging, conformance, separation

ACM Reference format:
Roberto Passerone, Íñigo Íncer Romeo, and Alberto L. Sangiovanni-Vincentelli. 2019. Coherent Extension, Composition, and Merging Operators in Contract Models for System Design. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 86 (October 2019), 23 pages.
https://doi.org/10.1145/3358216

1 INTRODUCTION

The escalating size and complexity of embedded and cyber-physical systems makes abstraction and reuse indispensable tools for early system and software design and verification. In particular, determining the correctness of designs made of library parts that are developed independently implies the development of formal methods that can guarantee compositionality. To do this, system and embedded software design are resorting to models for both verification and automatic code generation. Viewpoints, i.e., orthogonal "sub-models" of components that come together (are

This article appears as part of the ESWEEK-TECS special issue and was presented at the International Conference on Embedded Software (EMSOFT) 2019.
Authors' addresses: R. Passerone, University of Trento, Dipartimento di Ingegneria e Scienza dell'Informazione, Via Sommarive 9, Trento, Italy, 38123; email: roberto.passerone@unitn.it; Í. Íncer Romeo and A. L. Sangiovanni-Vincentelli, University of California, Berkeley, Department of Electrical Engineering and Computer Sciences, Berkeley, CA, 94720; emails: {inigo, alberto}@eecs.berkeley.edu.

1539-9087/2019/10-ART86 $15.00
https://doi.org/10.1145/3358216

merged) to form a comprehensive description of a component, are essential to consider heterogeneity of models and specifications. In this context, assume-guarantee contracts can play an important role in developing a robust design methodology [6] for cyber-physical systems and in particular, for the development of provably correct synthesis methods for embedded software while considering explicitly the constraints and effects of the design environment. Theories that encompass both components and their environment have evolved from component models [17, 33] to interface models, type systems [10, 15, 21, 25, 26, 29, 34], and contract models [5, 7, 8, 13, 36, 37]. In this paper, we propose a number of extensions to these theories which are *transversal* to the form of specification or model. In particular, we examine the general form of an interface or contract, and study equivalent representations that facilitate the definition of the operators and refinement relations. We deal with refinement between components at the same level of abstraction, which we call *conformance* to distinguish from other forms of refinement. In particular, we are interested in conformance and its related conjunction operator applied across different *viewpoints*. Our objective is to derive fundamental results in a sufficiently general setting, and apply them to understand concrete aspects and issues that underlie and/or are faced by several models. To do this, we adapt the mathematical frameworks proposed by Benveniste et al. [6, 14], with simplifications to facilitate the exposition. Our intent is not to provide a new or more comprehensive interface or contract model, or a meta-model. Instead, we are interested in exploring the properties of several operators, and deal primarily with the underlying theory. Nevertheless, several examples are introduced to motivate and illustrate the methodology, and a use case shows the combined application of the operators, which leads to a more accurate result compared to previous work. Questions of efficiency and expressiveness, and the application to case studies, must instead be solved at the level of the concrete models, and constitute our future work.

The first part of the paper builds our theory by increasingly adding structure to a simple component model. We then introduce the actual interfaces, and show how to maximally extend the interface specification while retaining equivalence. The theory sheds light into several constructions employed when computing the composition operator or the normal form of many interface models found in the literature. We also study closure properties of various sub-classes of interface specifications. Subsequently, we apply our theory to discuss some subtle issues that we believe are not yet well understood and may lead to unexpected or downright incorrect results. We focus on these fundamental aspects:

- how weak forms of assumptions and guarantees, in the style of Damm [13], may coexist to provide a more flexible specification mechanism;
- how to effect viewpoint merging of contracts in a way that correctly deals with both assumptions and guarantees; and
- how to decompose contracts and separate viewpoints using the operations of quotient and separation.

Proofs of our results can be found in a technical report [35]. We lay the foundations of our theory in Section 2, while Section 3 presents the applications of our basic model, discussing a new interpretation of weak and strong assumptions and introducing merging and decomposition operators. Section 4 is devoted to a discussion of the related work, while Section 5 provides concluding remarks.

2 COMPONENT, INTERFACES AND CONTRACT MODELS

In this section we introduce various notions that range from simple components to full fledged interfaces. The term "contract" will be reserved for a particular form of interface. After the basic definitions, we explore the notion of conformance and of composition of interfaces. We build over

concepts developed for the Heterogeneous Rich Component model (HRC) [7, 14], where components, environments and the assumptions and promises of interfaces are simply Extended State Machines (ESMs), and are identified with the set of runs (or behaviors) that they accept, in the form of denotational semantics [5]. Our theory significantly extends these basic notions by precisely characterizing the role of the elements of an interface.

2.1 Components

Components are the basic building blocks of a model, and are identified denotationally in our settings simply as sets of possible behaviors. Components interact with one another through ports (signals, variables, method calls, etc.) identified through "names" taken from a master alphabet $\mathcal{A}$. A behavior (trace) is some object constructed over the alphabet of the component, such as a sequence of values assigned to the ports. We are not concerned with the specific form of the behavior, which depends on the particular model implementation. A component is therefore simply a set of behaviors.

Definition 2.1 (Component). Let $\Sigma \subseteq \mathcal{A}$ be an alphabet, and let $\mathcal{B}(\Sigma)$ be the set of all behaviors over alphabet Σ. A *component* M over alphabet Σ is a set of behaviors from $\mathcal{B}(\Sigma)$, i.e.,

$$M \subseteq \mathcal{B}(\Sigma).$$

A component may further partition its alphabet into a set of inputs I and outputs O, such that $I \cup O = \Sigma$.

Several styles can be used to concretely express component specifications. Damm et al. [13] use a pattern-based assertion language to symbolically encode a set as the behaviors that satisfy the assertion (see Section 3.6 for an example). Automata [7, 15] can be used as recognizers to select a set of sequences as the component behaviors. A specification could also be explicit. An example is the Tagged Signal Model (TSM) [27, 28], where an underlying set of partially ordered tags T is used to encode the interaction model (un-timed, precedence, synchronous, asynchronous, various notions of time, etc.), while ports take on values from a set V. An event on a port a is a triple (a, τ, v), where $\tau \in T$ is the "time" at which the event occurs, and v is its value. Time, here, does not necessarily correspond to "physical" time, and the tags may be used to simply encode precedence relations. Let E be the set of all events $E = \Sigma \times T \times V$. Then a behavior x is defined as a set of events $x \in E$, and the set of all behaviors is the powerset of E, i.e., $\mathcal{B}(\Sigma) = 2^E = 2^{\Sigma \times T \times V}$.

Two components may have the same set of behaviors, but be defined over different alphabets (one contained in the other). In this case, we consider the two components as distinct. Likewise, components over the same alphabet that partition the ports into different input and output sets are distinct.

The composition of two components is defined as the intersection of the corresponding sets of behaviors. This operation is easy to define when the components are defined over the same alphabet Σ. To be composable, the set O_1 of outputs of M_1 and the set O_2 of outputs of M_2 must be disjoint. The underlying semantics is that components mutually constrain their behaviors.

Definition 2.2 (Component Composition). Let M_1 and M_2 be components over alphabet Σ, and let $O_1 \cap O_2 = \emptyset$. Their composition $M = M_1 \parallel M_2$ is the component over alphabet Σ given by

$$M = M_1 \cap M_2.$$

The effect of composition is that components "synchronize" on their behaviors. This does not necessarily imply a synchronous model. For instance, partially ordered sets of tags can be employed in TSM to model asynchronous behaviors, using the same underlying notion of composition [4].

In practice, behaviors are retained after composition if the events on the shared ports share the same tags, preserving the partial order relation.

Projection takes a behavior x' over alphabet Σ' and computes a restriction x over an alphabet $\Sigma \subseteq \Sigma'$. We denote the operation as $x = \mathbf{proj}_{\Sigma',\Sigma}(x')$, where, obviously, Σ is the set of ports to be *retained*. The *inverse of the projection* operator can be used to extend the alphabet of a behavior from Σ to Σ'. Projection for components is the natural extension to sets of the projection on behaviors.

Definition 2.3 (Component Projection). Let M be a component over alphabet Σ', and let $\Sigma \subseteq \Sigma'$. The projection of M to Σ is defined as

$$\mathbf{proj}_{\Sigma',\Sigma}(M) = \{x \in \mathcal{B}(\Sigma) \mid \exists y \in M, x = \mathbf{proj}_{\Sigma',\Sigma}(y)\}.$$

For instance, projection in TSM is computed by retaining only the events on ports that must not be hidden. For a behavior $x \in \mathcal{B}(\Sigma')$ we define the projection onto alphabet $\Sigma \subseteq \Sigma'$ as

$$\mathbf{proj}_{\Sigma',\Sigma}(x) = \{(a, \tau, v) \in x \mid a \in \Sigma\}.$$

Projection works at one level of abstraction, and does not alter the notion of time given by the tags. Because projection hides ports, several different behaviors may project to the same one. The inverse of projection is therefore a set. This works naturally for components.

Definition 2.4 (Component Inverse Projection). Let M be a component over alphabet Σ, and let $\Sigma' \supseteq \Sigma$. The inverse projection of M to Σ' is defined as

$$\mathbf{proj}^{-1}_{\Sigma',\Sigma}(M) = \{x \in \mathcal{B}(\Sigma') \mid \mathbf{proj}_{\Sigma',\Sigma}(x) \in M\}.$$

In TSM, inverse projection populates the behaviors with all possible events in the additional ports to avoid constraining their behavior. When two components do not have the same alphabet, their composition is computed by composing the components obtained by their inverse projection to the union of the alphabets. Thus, for components M_1 and M_2 over alphabets Σ_1 and Σ_2, parallel composition is the component M over alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ given by

$$M = M_1 \parallel M_2 = \mathbf{proj}^{-1}_{\Sigma,\Sigma_1}(M_1) \parallel \mathbf{proj}^{-1}_{\Sigma,\Sigma_2}(M_2).$$

Components can be ordered based on the level of specification they contain. If a component M contains more information than a component M', we say that M *conforms to* M'. We may characterize conformance as a relation of substitutability: M conforms to M', written $M \preceq M'$, whenever M can be substituted for M' everywhere M' occurs. Thus, conformance is usually taken as behavior containment.

Definition 2.5 (Component Conformance). Let M and M' be components over the same alphabet Σ, such that $I_1 = I_2$ and $O_1 = O_2$. Then, M conforms to M', written $M \preceq M'$, whenever

$$M \subseteq M'.$$

With this definition, a component M' represents the potential behaviors of its refinements. Two components are *equivalent*, written $M \sim M'$ when they conform to each other.

For each alphabet Σ there exist a top and a bottom component in the conformance order, corresponding to the universe of behaviors $\top = \mathcal{B}(\Sigma)$ and to the empty set of behaviors $\bot = \emptyset$, respectively. Since the structure of components forms a lattice, it is natural to define the complement of a component as the set of behaviors *not* included in the component.

Definition 2.6 (Component Complementation). Let M be a component over alphabet Σ. The complement $\overline{M}$ of M is the component given by

$$\overline{M} = \mathcal{B}(\Sigma) - M.$$

Conjunction of components is achieved by taking their greatest lower bound, and is an operation derived from the conformance order. Theorem 2.7 below shows that, with the given conformance order, this operation again reduces to the intersection of the behaviors of the components, as in parallel composition. The semantics of conjunction, however, differs significantly from that of parallel composition. In conjunction, the "components" correspond to the specification of different *viewpoints* that relate to the *same* underlying object; conversely, composition puts together the specification of *different parts* of the system. Likewise, disjunction corresponds to union.

Theorem 2.7 (Component Conjunction and Disjunction). *Let M_1 and M_2 be components over alphabet Σ, such that $I_1 = I_2$ and $O_1 = O_2$. The conjunction $M = M_1 \sqcap M_2$ is the component over alphabet Σ given by*

$$M = M_1 \cap M_2.$$

The disjunction $M = M_1 \sqcup M_2$ is the component over alphabet Σ given by

$$M = M_1 \cup M_2.$$

2.2 Interfaces

An interface expresses both the requirements demanded on the environment of a component and its guarantees relative to the same environment. The requirements are the conditions under which a component may be used by its context. A violation of the requirements indicates that the component may fail or malfunction, because it is driven outside its intended range of use. Correspondingly, a component implementing an interface must correctly accept the allowed behaviors from the environment.

In the upcoming definition of interface, we will treat requirements and guarantees independently. This is not the way it is done normally in contracts, where requirements and guarantees are related. We do so for generality, and will show later how contracts are special cases of interfaces that enjoy certain maximality and symmetry properties.

Definition 2.8 (Interface). An interface $\mathcal{I}$ over alphabet Σ is a pair $(\mathcal{R}, \mathcal{G})$ where $\mathcal{R}$ (the requirements) and $\mathcal{G}$ (the guarantees) are components over Σ.

We distinguish between the admissible implementations and the admissible environments of an interface. An implementation or an environment is admissible when it conforms to the guarantees and the requirements of the interface, respectively.

Definition 2.9 (Implementations and Environments). A component M is an *implementation* of the interface $\mathcal{I} = (\mathcal{R}, \mathcal{G})$, written $M \models_G \mathcal{I}$, if and only if $M \preceq \mathcal{G}$. It is an *environment* of $\mathcal{I}$, written $E \models_R \mathcal{I}$, if and only if $E \preceq \mathcal{R}$.

Parallel composition is defined by working independently on requirements and guarantees. As usual, inverse projection is used when necessary to equalize the alphabets.

Definition 2.10. Let $\mathcal{I}_1 = (\mathcal{R}_1, \mathcal{G}_1)$ and $\mathcal{I}_2 = (\mathcal{R}_2, \mathcal{G}_2)$ be interfaces. Then,

$$\mathcal{I}_1 \parallel \mathcal{I}_2 = (\mathcal{R}_1 \parallel \mathcal{R}_2, \mathcal{G}_1 \parallel \mathcal{G}_2).$$

Because we handle requirements and guarantees independently, we have no interaction between the two, as it happens in the majority of the interface and contract models found in the literature. We will deal with this and show an example later when we consider how to propagate requirements when constructing maximal interfaces.

Conformance can be defined in a way similar to components, again working separately for requirements and guarantees. In the case of requirements, however, substitutability imposes that

they be *weakened* across conformance, rather than *strengthened*. With this choice, an interface conforms to another interface whenever it has fewer implementations and more environments.

Definition 2.11 (Interface Conformance and Equivalence). Let $\mathcal{I} = (\mathcal{R}, \mathcal{G})$ and $\mathcal{I}' = (\mathcal{R}', \mathcal{G}')$ be interfaces. We define

- $\mathcal{I} \preceq \mathcal{I}'$ if and only if $\mathcal{R}' \preceq \mathcal{R}$ and $\mathcal{G} \preceq \mathcal{G}'$.
- $\mathcal{I} \sim \mathcal{I}'$ if and only if $\mathcal{R}' \sim \mathcal{R}$ and $\mathcal{G} \sim \mathcal{G}'$.

These conditions can be restated in terms of implementations and environments.

Corollary 2.12. *An interface $\mathcal{I}$ conforms to an interface $\mathcal{I}'$, written $\mathcal{I} \preceq \mathcal{I}'$, if and only if, for all components M and E,*

$$M \models_G \mathcal{I} \Rightarrow M \models_G \mathcal{I}' \quad \text{and} \quad E \models_R \mathcal{I}' \Rightarrow E \models_R \mathcal{I}.$$

Because of our definitions, equivalence is the same as equality. Conformance is a pre-order relationship on interfaces. We can therefore derive the corresponding greatest lower bound (conjunction) and least upper bound (disjunction) operators from conformance.

Lemma 2.13 (Interface Bounds). *An interface $\mathcal{I}$ is a greatest lower bound of the interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$, written $\mathcal{I} = \mathcal{I}_1 \sqcap \mathcal{I}_2$, if and only if, for all components M and E,*

$$M \models_G \mathcal{I} \Leftrightarrow M \models_G \mathcal{I}_1 \wedge M \models_G \mathcal{I}_2 \quad \text{and}$$
$$E \models_R \mathcal{I} \Leftrightarrow E \models_R \mathcal{I}_1 \vee E \models_R \mathcal{I}_2.$$

It is a least upper bound, written $\mathcal{I} = \mathcal{I}_1 \sqcup \mathcal{I}_2$, if and only if, for all components M and E,

$$M \models_G \mathcal{I} \Leftrightarrow M \models_G \mathcal{I}_1 \vee M \models_G \mathcal{I}_2 \quad \text{and}$$
$$E \models_R \mathcal{I} \Leftrightarrow E \models_R \mathcal{I}_1 \wedge E \models_R \mathcal{I}_2.$$

The conjunction of two interfaces can be computed by taking the greatest lower bound of their guarantees and the least upper bound of their requirements. The condition is dual for the least upper bound.

Theorem 2.14. *Let $\mathcal{I}_1 = (\mathcal{R}_1, \mathcal{G}_1)$ and $\mathcal{I}_2 = (\mathcal{R}_2, \mathcal{G}_2)$ be interfaces. Then,*

$$\mathcal{I}_1 \sqcap \mathcal{I}_2 = (\mathcal{R}_1 \sqcup \mathcal{R}_2, \mathcal{G}_1 \sqcap \mathcal{G}_2) \quad \text{and}$$
$$\mathcal{I}_1 \sqcup \mathcal{I}_2 = (\mathcal{R}_1 \sqcap \mathcal{R}_2, \mathcal{G}_1 \sqcup \mathcal{G}_2).$$

Unlike components, because we handle requirements and guarantees differently, conjunction and parallel composition of interfaces are not the same.

2.3 Contracts

In Definition 2.8, the requirements and the guarantees are independent of each other. However, since they both refer to the same object, we expect that there be some relation between them. For instance, if the guarantees $\mathcal{G}$ ensure that all implementations do not have a behavior, then a requirement that excludes this behavior from the environment can be eliminated, since the assumption is automatically satisfied by the underlying component. In other words, if a behavior is not possible in the implementation, then it should be allowed in the environment, since under composition the behavior is in any case excluded by the implementation. The same holds between the guarantees of the environment (the requirements) and those of the implementation (the guarantees). Eliminating such inconsistencies leads to the notion of a *contract*.

We define the *maximal system* of an interface as the composition of its maximal implementation and its maximal environment, relative to conformance. Any composition of an implementation and an environment of the interface necessarily conforms to the maximal system.

Definition 2.15 (Maximal System). The maximal system of an interface $\mathcal{I} = (\mathcal{R}, \mathcal{G})$ is the component $S_{\mathcal{I}} = \mathcal{G} \parallel \mathcal{R}$.

Two interfaces are system equivalent when they have the same maximal system.

Definition 2.16 (System Equivalence). Two interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$ are *system equivalent*, written $\mathcal{I}_1 \equiv \mathcal{I}_2$, if and only if $S_{\mathcal{I}_1} = S_{\mathcal{I}_2}$.

If two interfaces are equivalent (Lemma 2.11), then they are also system equivalent.

Lemma 2.17. $\mathcal{I}_1 \sim \mathcal{I}_2 \Rightarrow \mathcal{I}_1 \equiv \mathcal{I}_2$.

Interfaces can be extended by increasing both their set of implementations and environments. This is unlike interface conformance (see Definition 2.11), where one is extended and the other shrunk.

Definition 2.18 (Extension). An interface $\mathcal{I}'$ is an extension of an interface $\mathcal{I}$, written $\mathcal{I} \rightsquigarrow \mathcal{I}'$, if and only if, for all components M and E,

$$M \models_G \mathcal{I} \Rightarrow M \models_G \mathcal{I}' \quad \text{and} \quad E \models_R \mathcal{I} \Rightarrow E \models_R \mathcal{I}'.$$

Corollary 2.19. $\mathcal{I} \rightsquigarrow \mathcal{I}' \iff \mathcal{G} \subseteq \mathcal{G}' \wedge \mathcal{R} \subseteq \mathcal{R}'$.

We are interested in extensions that do not alter the underlying semantics of an interface. A *completion* of an interface is an extension that preserves system equivalence.

Definition 2.20 (Completion). An interface $\mathcal{I}'$ is a *completion* of an interface $\mathcal{I}$, written $\mathcal{I} \mapsto \mathcal{I}'$, if and only if $\mathcal{I} \equiv \mathcal{I}'$ and $\mathcal{I} \rightsquigarrow \mathcal{I}'$.

Completion is a partial order (modulo equivalence). A maximal interface is one that cannot be further completed.

Definition 2.21 (Maximal Interface). An interface $\mathcal{I}$ is maximal if and only if it is maximal under completion, i.e., for all interfaces $\mathcal{I}'$, if $\mathcal{I} \mapsto \mathcal{I}'$, then $\mathcal{I} \sim \mathcal{I}'$.

Definition 2.21 says that an interface is maximal whenever it cannot be extended to another interface that has strictly more implementations and/or environments, while preserving the maximal system. This is useful to both simplify and enrich the interface, as the example at the end of this section will show. In practice, completion can be used to add behaviors to either the requirements or the guarantees, subject to the original specification, so that no behavior is left unassigned. A maximal interface is in fact complete, or saturated, in the following sense.

Theorem 2.22. *Let $\mathcal{B}(\Sigma)$ be the universe of behaviors over alphabet Σ. An interface $\mathcal{I}$ over Σ is maximal if and only if $\mathcal{G} \cup \mathcal{R} = \mathcal{B}(\Sigma)$.*

Corollary 2.23. *An interface $\mathcal{I}$ is maximal if and only if $\mathcal{G} = \mathcal{G} \cup \overline{\mathcal{R}}$.*

The possible implementations of a maximal interface are those that conform to $\mathcal{G} \cup \overline{\mathcal{R}}$, which is equivalent to $\mathcal{R} \Rightarrow \mathcal{G}$.

A given interface has, in general, several different maximal completions. At the two extremes, an interface may be completed by relaxing requirements only, or guarantees only.

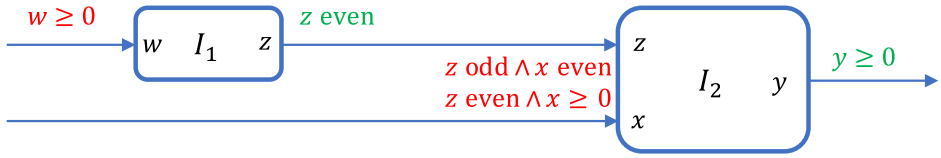

Fig. 1. Composition of simple interfaces.

Theorem 2.24. *Let $\mathcal{I} = (\mathcal{R}, \mathcal{G})$ be an interface. Then the interfaces $\mathcal{I}^{\mathcal{R}}$ and $\mathcal{I}^{\mathcal{G}}$ defined by*

$$\mathcal{I}^{\mathcal{R}} = (\mathcal{R} \cup \overline{\mathcal{G}}, \mathcal{G}) \quad \text{and} \quad \mathcal{I}^{\mathcal{G}} = (\mathcal{R}, \mathcal{G} \cup \overline{\mathcal{R}})$$

are maximal completions of $\mathcal{I}$.

Maximal completions are similar to the HRC canonical form. Here, $\mathcal{I}^{\mathcal{R}}$ has maximally relaxed requirements, and therefore more possible environments, while $\mathcal{I}^{\mathcal{G}}$ has maximally relaxed guarantees, and therefore more possible implementations. Other completions are also possible, which relax partly the requirements and partly the guarantees. When composing interfaces it is useful to complete the resulting interface by relaxing requirements. This is because, after composition, certain requirements of one component will be already fulfilled by the other. Thus, the environment need not provide those guarantees any more. We denote the operation of composition, followed by requirement relaxation, as $\|_{\mathcal{R}}$.

Definition 2.25 (Product - Requirement Relaxation). Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be interfaces. Then $\mathcal{I}$ is a product with requirement relaxation of $\mathcal{I}_1$ and $\mathcal{I}_2$, written $\mathcal{I} = \mathcal{I}_1 \|_{\mathcal{R}} \mathcal{I}_2$, if and only if

$$\mathcal{G} = \mathcal{G}_1 \cap \mathcal{G}_2 \quad \text{and} \quad \mathcal{R} = (\mathcal{R}_1 \cap \mathcal{R}_2) \cup \overline{(\mathcal{G}_1 \cap \mathcal{G}_2)}.$$

Likewise, if we are interested in composing environments, rather than components, it may be convenient to relax the guarantees of the interface.

Definition 2.26 (Product - Guarantee Relaxation). Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be interfaces. Then $\mathcal{I}$ is a product with guarantee relaxation of $\mathcal{I}_1$ and $\mathcal{I}_2$, written $\mathcal{I} = \mathcal{I}_1 \|_{\mathcal{G}} \mathcal{I}_2$, if and only if

$$\mathcal{G} = (\mathcal{G}_1 \cap \mathcal{G}_2) \cup \overline{(\mathcal{R}_1 \cap \mathcal{R}_2)} \quad \text{and} \quad \mathcal{R} = \mathcal{R}_1 \cap \mathcal{R}_2.$$

In general, the regular interface product of maximal interfaces is not maximal, so that maximal interfaces are not closed under interface product. However, by using requirement or guarantee relaxation we always obtain a maximal interface, as shown by the following lemma.

Lemma 2.27. *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be interfaces. Then $\mathcal{I}_1 \|_{\mathcal{R}} \mathcal{I}_2$ and $\mathcal{I}_1 \|_{\mathcal{G}} \mathcal{I}_2$ are maximal interfaces.*

Thus, one can always work with maximal interfaces, and use one of the above products as the definition of interface product to have closure.

Definitions 2.25 and 2.26 are useful because they propagate assumptions and guarantees through a composition. This is particularly important when dealing with inputs and outputs. In fact, once the input of one component is driven by an output of another component, then the environment has no way of driving that input (because composition is not defined when two components drive the same signal). In practice, that signal can be projected out after composition. However, unless we propagate assumptions and guarantees, we will obtain a sub-optimal interface, in the sense that there are possible system equivalent extensions that are lost.

Consider for example the stateless interface $\mathcal{I}_1 = (\mathcal{R}_1, \mathcal{G}_1)$ over alphabet $\{w, z\}$ and interface $\mathcal{I}_2 = (\mathcal{R}_2, \mathcal{G}_2)$ over $\{x, y, z\}$, with w, x, y, z reals, shown in Figure 1, defined as follows:

$$\begin{aligned}\mathcal{I}_1 \quad &: \quad \mathcal{R}_1 = \{w \geq 0\}, \mathcal{G}_1 = \{z \in \text{even}\}\\ \mathcal{I}_2 \quad &: \quad \mathcal{R}_2 = \{(z \in \text{odd} \wedge x \in \text{even}) \vee (z \in \text{even} \wedge x \geq 0)\}\\ &\quad\ \ \mathcal{G}_2 = \{y \geq 0\}\end{aligned}$$

Inverse projection and composition leads to $\mathcal{I} = (\mathcal{R}, \mathcal{G}) = \mathcal{I}_1 \parallel \mathcal{I}_2$:

$$\begin{aligned}\mathcal{R} &= \{w \geq 0, (z \in \text{odd} \wedge x \in \text{even}) \vee (z \in \text{even} \wedge x \geq 0)\}\\ \mathcal{G} &= \{y \geq 0, z \in \text{even}\}\end{aligned}$$

Because $\mathcal{I}_1$ guarantees that z is even, we can extend $\mathcal{I}$ by maximally relaxing the requirements. Computing the complement of the guarantees, taking the union with the requirements, and after logic simplification, we obtain:

$$\begin{aligned}\mathcal{R} &= (\mathcal{R}_1 \cap \mathcal{R}_2) \cup \overline{(\mathcal{G}_1 \cap \mathcal{G}_2)}\\ \overline{(\mathcal{G}_1 \cap \mathcal{G}_2)} &= \{y < 0 \vee z \in \text{odd}\}\\ \mathcal{R} &= \{w \geq 0, z \in \text{odd} \wedge x \geq 0\}\end{aligned}$$

Signal z can now be considered internal to the compound and can be projected away, leading to the final result:

$$\begin{aligned}\mathcal{R} &= \{w \geq 0, x \geq 0\}\\ \mathcal{G} &= \{y \geq 0\}.\end{aligned}$$

The operator has correctly accounted for the already discharged assumptions, resulting in the expected more permissive contract. We therefore define the following:

Definition 2.28 (Contract Model). The set of maximal interfaces equipped with conformance and product with requirement relaxation forms a *contract model* closed under composition.

Whenever a contract is defined which is not maximal, it is completed by relaxing the guarantees, to make the underlying implication explicit. This theory generalizes the contract theory introduced for the HRC model. There, a contract is defined as a pair (A, G), and is treated essentially as an implication. Difficulties, however, arise with the handling of conformance, there called *dominance*, due to the lack of uniqueness of its expression. Contracts are therefore restricted to only expressions in *canonical form*, as described previously in Theorem 2.24. If interpreted as an interface, the expression is equivalent to maximally relaxing the guarantees of the contract. Products are handled using the device of requirement relaxation, as shown in Definition 2.25. This is both convenient from a logical point of view, and helps maintain closure in the model when using only canonical forms. In addition, our theory shows that contracts are concerned with system equivalence, rather than with the stronger general equivalence.

In summary, contracts are maximal interfaces with requirement relaxation for composition, and guarantee relaxation for the initial canonical form. This favors flexibility of the implementation when contracts are defined, while it favors environments when contracts are put together in composition. In the previous sections we have generalized this theory and shown how these are only some of the possible choices to achieve maximality.

3 SYMMETRIES AND VIEWPOINTS

In Section 2 we have presented the basic contract theory, building from the simple notion of component and behavior containment to develop more complex objects able to express assumptions and guarantees. In this section, we consider whether the assumptions should or should not constrain the environment, a distinction that has been referred to in the literature as *strong* and *weak* assumptions [13]. We then more closely look at the operators of composition, and consider the

relation between parallel composition and the process of merging of viewpoints. We show how decomposing the operators leads to a coherent definition of merging, which closely captures the design intent.

3.1 Weak and Strong Assumptions

We have defined an interface, and therefore a contract, as a pair $\mathcal{I} = (\mathcal{R}, \mathcal{G})$ of requirements and guarantees, represented by component specifications. The contract works as an *implication*: if the requirements are satisfied, then also the guarantees must be satisfied. We can model an implication at the level of the component model by using its lattice structure.

Definition 3.1 (Implication). An *implication* C over alphabet Σ is a pair (A, G), where A (the assumption) and G (the promise) are components over Σ. The implication is equivalent to the component M given by

$$M = A \rightarrow G = G \sqcup \overline{A} = G \cup \overline{A}.$$

The difference between a contract and an implication is that a contract also constrains the environment, while an implication simply expresses a logical connective. By Definition 2.5, a component M over Σ satisfies an implication $C = (A, G)$ over Σ, written $M \models C$, if and only if

$$M \subseteq A \rightarrow G = G \cup \overline{A},$$

or, equivalently, whenever

$$M \parallel A \subseteq G.$$

Implications can be used as a specification mechanism for contracts to model concepts such as the weak and strong assumptions introduced by Damm et al. [13]. In particular, the guarantees $\mathcal{G}$ of a contract can be naturally expressed as an implication $\mathcal{G} = (A, G)$ over its alphabet. The requirements $\mathcal{R}$ may also be expressed as an implication $\mathcal{R} = (H, R)$, where H represent conditions on the implementations (i.e., assumptions made by the environment) under which the environment must satisfy its promises R (i.e., the contract requirements). When requirement and guarantees are specified as implications, a contract is of the form

$$\mathcal{I} = ((H, R), (A, G)).$$

The interest in this form lies in the explicit specification of different degrees of assumptions. Here, the requirements $\mathcal{R} = (H, R) = R \cup \overline{H}$ express assumptions that the environment *must* satisfy in order for the underlying component to function correctly. Their violation is therefore considered a failure of the system, and the assumption $\mathcal{R}$ is referred to as *strong* [13]. Conversely, the assumptions A of the second implication are the conditions under which the property G is guaranteed by the component. However, A does not constrain the environment: if violated, the implication simply does not hold and the promise is not guaranteed. The assumption A is therefore called *weak* [13]. Formalizing weak and strong assumptions this way clearly defines their scope, and does not require introducing any new theory to handle them correctly, as was instead proposed by Damm et al. [13]. In particular, we do not need the extra condition that "the strong assumptions of the parts are either discharged by the guarantees of other parts or covered by the strong assumption of the aggregate" [13], since composition will automatically take care of propagating assumptions through the device of requirement relaxation. We will further analyze these aspects, and provide examples, in Section 3.6 after we discuss merging.

3.2 Symmetric Interfaces

One can freely manipulate weak and strong assumptions using the results of the previous sections, and define products and logical operations. The difficulty with weak assumptions is that implications do not have unique representations, and may therefore lose their significance after going

through composition and merging. This can be overcome by considering the semantics of implications. The standard interpretation of the implication is that a component should provide the stated promise under the given assumption. A stronger interpretation also dictates that the environment should guarantee the assumption given the promise. This can be expressed by an interface that uses, as requirement, the dual of the implication used for the guarantees. An interface of this kind is said to be *symmetric*.

Definition 3.2 (Dual). Let $C = (A, G)$ be an implication. The dual of C is the implication $\hat{C}$ such that $\hat{C} = (G, A)$.

Definition 3.3 (Symmetric Interface). An interface $\mathcal{I} = (\mathcal{R}, \mathcal{G})$ is symmetric if and only if $\mathcal{R} = \hat{\mathcal{G}}$.

The interface is symmetric because, assuming $\mathcal{G} = (A, G)$, then $\mathcal{I}$ can be written as $((G, A), (A, G))$. Symmetric interfaces enjoy several properties. First, a symmetric interface is also maximal.

Theorem 3.4. *If $\mathcal{I}$ is a symmetric interface, then $\mathcal{I}$ is also maximal.*

In addition, a maximal interface has always an equivalent interface that is symmetric.

Theorem 3.5. *Let $\mathcal{I} = (\mathcal{R}, \mathcal{G})$ be a maximal interface. Then there exists a symmetric interface $\mathcal{I}' = ((G, A), (A, G))$ such that $\mathcal{I} \sim \mathcal{I}'$.*

Like maximal interfaces, symmetric interfaces are not closed under products. However, the use of requirement and/or guarantee relaxation (Definitions 2.25 and 2.26) will always yield a maximal interface, which has a symmetric equivalent. Thus, the operation of parallel composition is well defined up to equivalence.

Because maximal interfaces always have equivalent (not just system equivalent) symmetric interfaces (Theorem 3.5), then symmetric interfaces are sufficient as a representation if we plan to work only with maximal interfaces. The representation is sufficient because the pair of assumptions and promises (A, G) of an implication is a highly redundant representation, and one set, the component to which the implication is equivalent (see Definition 3.1), is enough. Thus, two sets are enough for the representation of an interface, provided we are only interested in system equivalence. Obviously, since any interface has a maximal system-equivalent extension, it also has a symmetric system equivalent extension, which justifies the use of symmetric interfaces as the only specification theory. Most interface and contract models implicitly rely on this result, which, to the best of our knowledge, was not reported before.

We can define a canonical form for symmetric interfaces similar to the one defined for implications.

Definition 3.6 (Symmetric Canonical Form). A symmetric interface $\mathcal{I} = (\hat{C}, C)$, with $C = (A, G)$ is in *canonical form* whenever $A = A \cup \overline{G}$ and $G = G \cup \overline{A}$.

This definition is one particular instance of Theorem 3.5. Therefore, every maximal interface has an equivalent symmetric interface in canonical form. When we restrict our formalism to using only symmetric interfaces in canonical form, then interface conformance can be checked on the individual assumptions and promises of C.

Theorem 3.7 (Conformance of Symmetric Interfaces in Canonical Form). *Let $\mathcal{I} = (\hat{C}, C)$, and $\mathcal{I}' = (\hat{C}', C')$ be symmetric interfaces in canonical form, with $C = (A, G)$ and $C' = (A', G')$. Then, $\mathcal{I} \preceq \mathcal{I}'$ if and only if*

$$G \subseteq G' \quad \text{and} \quad A' \subseteq A.$$

This result matches the expression for refinement of the majority of interface and contract models, which in fact are generally expressed as symmetric interfaces in canonical form.

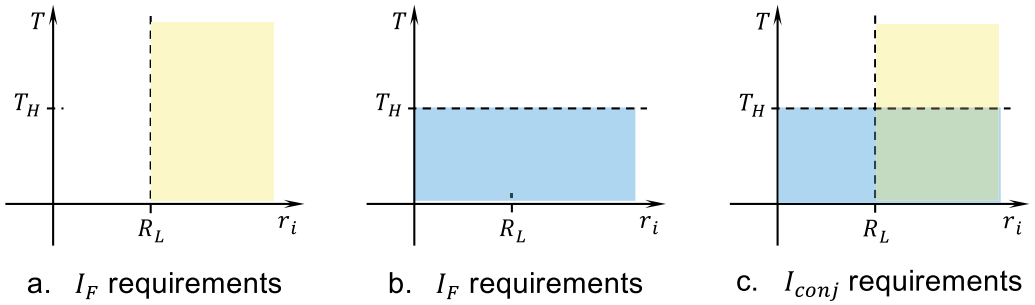

Fig. 2. Requirements: rate, temperature, and their union.

3.3 A Revised Notion of Contract Merging

Besides parallel composition, there is another binary operation we would like to carry out on pairs of contracts. Assume two contracts specify different aspects of the same component. One contract could specify, for example, how the component behaves functionally, and another could describe its timing or power characteristics. We call these aspects *viewpoints*. The operation of *viewpoint merging* consists in combining various contracts describing different aspects of the same component into a single contract object. A natural choice for merging, and one that is followed by most of the literature on contracts, is the conjunction operation [5, 6]. We here show, however, that conjunction has some undesirable effects.

Suppose a device guarantees to output data at certain rate R_o provided the data rate r_i of the input is higher than some minimum R_L. Suppose the same device guarantees it will consume less than P units of power if the temperature is bounded above by T_H. The contract rejects environments that do not satisfy such requirements. We can write the following functional and power contracts for this device:

$$\mathcal{I}_F = (r_i > R_L, r_o = R_o) \quad \text{and} \quad \mathcal{I}_P = (T < T_H, p < P).$$

We can take their conjunction after inverse projecting to equalize the alphabets. From a syntactic standpoint, inverse projection has no impact on the definition of the contract: if a variable is added to the requirements or guarantees, the proposition added must allow the variable to take any value in its domain, which is a true proposition. For example, the requirements of $\mathcal{I}_F$ can be extended with the proposition $0 < T < \infty$, which is always true since T is a non-negative real number. The extended requirements are shown graphically in Figure 2(a) and 2(b). Before carrying out conjunction, it is necessary to saturate both contracts. We obtain

$$\begin{aligned}\mathcal{I}_F &= (r_i > R_L,\ r_o = R_o \vee \neg(r_i > R_L)) \quad \text{and}\\ \mathcal{I}_P &= (T < T_H,\ p < P \vee \neg(T < T_H)).\end{aligned}$$

The conjunction is given by

$$\begin{aligned}\mathcal{I}_{\text{conj}} = (&r_i > R_L \vee T < T_H,\\ &(r_o = R_o \wedge p < P) \vee (r_o = R_o \wedge T \geq T_H) \vee\\ &(p < P \wedge r_i \leq R_L) \vee (T \geq T_H \wedge r_i \leq R_L).\end{aligned}$$

The resulting contract allows a satisfying component to guarantee either the guarantees of both viewpoints (i.e., $r_o = R_o \wedge p < P$) or the guarantees of only one of the viewpoints when the assumptions of only one of them holds (e.g., $r_o = R_o \wedge T \geq T_H$) or to guarantee nothing when none of the assumptions hold (i.e., $T \geq T_H \wedge r_i \leq R_L$). The requirements of the conjunction, shown in

Figure 2(c), are computed as the union and include the entire shaded area. These requirements appear too permissive, as a satisfying component must now be able to accept environments that produce rates below R_L, and must work at temperatures potentially higher than T_H. The problem lies in the *inverse projection combined with the union*. Ideally, we would like instead to consider only the intersection, which corresponds to only the green area in Figure 2(c). This must be regarded as a new operator, since the form of conjunction depends on the conformance order, and cannot simply be redefined (see Theorem 2.14).

The idea is that viewpoint merging should tell us what all viewpoints guarantee simultaneously since *the contract for each viewpoint demands that its assumptions be met*. Thus, contract *merging* is defined as product with guarantee relaxation:

Definition 3.8 (Merging). Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be contracts. Then $\mathcal{I}$ is the resulting of merging $\mathcal{I}_1$ and $\mathcal{I}_2$, written $\mathcal{I} = \mathcal{I}_1 \cdot \mathcal{I}_2$, if and only if

$$\mathcal{G} = (\mathcal{G}_1 \cap \mathcal{G}_2) \cup \overline{(\mathcal{R}_1 \cap \mathcal{R}_2)} \quad \text{and} \quad \mathcal{R} = \mathcal{R}_1 \cap \mathcal{R}_2.$$

Applying the merging operator to our example yields:

$$\begin{aligned}\mathcal{I}_F \cdot \mathcal{I}_P &= (r_i > R_L \wedge T < T_H, (r_o = R_o \vee \neg(r_i > R_L)) \wedge \\ &\qquad (p < P \vee \neg(T < T_H)) \vee r_i \leq R_L \vee T \geq T_H) \\ &= (R_L < r_i \wedge T < T_H, \\ &\qquad (r_o = R_o \wedge p < P) \vee T \geq T_H \vee r_i \leq R_L)\,.\end{aligned}$$

The guarantees of $\mathcal{I}_F \cdot \mathcal{I}_P$ now require that the guarantees of both contracts hold; moreover, this contract forces the environment to meet the requirements of both contracts. Thus, the merging operation correctly captures the intuitive notion of viewpoint merging. In what follows, we discuss the properties of the operator and how it fits coherently in the overall contract theory.

3.4 Composition, Merging, and the Contract Lattice

At this point, for a contract model we have two binary operations with semantic meaning: composition and merging. The contract model, however, is a partial order on conformance with well defined lattice operations. How are merging and composition related to the contract lattice? The following theorem tells us that merging is one of the components of the GLB.

Theorem 3.9. *Let $\mathcal{I} = (\mathcal{R}, \mathcal{G})$ and $\mathcal{I}' = (\mathcal{R}', \mathcal{G}')$ be contracts. Then the contract $\mathcal{I} \sqcap \mathcal{I}'$ is equal to the conjunction of the following three contracts:*

(1) $(\mathcal{R} - \mathcal{R}', \mathcal{G} \cup \overline{\mathcal{R} - \mathcal{R}'})$
(2) $(\mathcal{R}' - \mathcal{R}, \mathcal{G}' \cup \overline{\mathcal{R}' - \mathcal{R}})$
(3) $(\mathcal{R} \cap \mathcal{R}', \mathcal{G} \cap \mathcal{G}' \cup \overline{\mathcal{R} \cap \mathcal{R}'})$

We think of Theorem 3.9 as providing a factorization of the conjunction of two contracts into three contracts. The first two contracts of this factorization require the environment to support the environments of only one of $\mathcal{I}$ or $\mathcal{I}'$, and provide the guarantees only of one of the contracts. The third contract, on the other hand, which corresponds to merging, requires the environment to support the assumptions of both contracts simultaneously, and provides the guarantees of both contracts. The use of the first two contracts in the factorization is likely to be limited. The third contract, however, represents what the component does *when both viewpoints being conjoined are active simultaneously*. As we discussed, this corresponds exactly to the notion of *viewpoint merging*, hence the name of the operation.

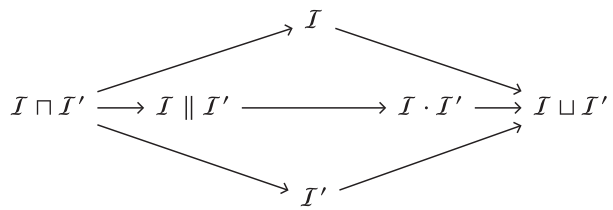

Fig. 3. Given contracts I and I', their operations of conjunction, disjunction, composition, and merging are ordered.

We now observe that composition is, dually to merging, a component of the factorization of the disjunction of two contracts.

THEOREM 3.10. *Let $\mathcal{I} = (\mathcal{R}, \mathcal{G})$ and $\mathcal{I}' = (\mathcal{R}', \mathcal{G}')$ be contracts. Then the contract $\mathcal{I} \sqcup \mathcal{I}'$ is equal to the disjunction of the following three contracts:*

(1) $(\mathcal{R} \cup \overline{\mathcal{G} - \mathcal{G}'}, \mathcal{G} - \mathcal{G}')$
(2) $(\mathcal{R}' \cup \overline{\mathcal{G}' - \mathcal{G}}, \mathcal{G}' - \mathcal{G})$
(3) $(\mathcal{R} \cap \mathcal{R}' \cup \overline{\mathcal{G} \cap \mathcal{G}'}, \mathcal{G} \cap \mathcal{G}')$

As with merging, composition is the third element of this factorization of the LUB. These two factorizations show that there is great duality between the notions of composition and merging. Both operations appear as a component of the factorizations of elementary operations of contracts, namely conjunction and disjunction, operations which are generated from the contract partial order. Figure 3 shows how merging, composition, conjunction, and disjunction are ordered.

3.5 Decomposition of Contracts and Separation of Viewpoints

In order to support a modular design process, the ability to *decompose* contracts into simpler ones is crucial; we call *quotient* the operation that allows us to carry out decomposition. The quotient is defined in terms of composition and conformance. Like composition, the quotient takes two contracts $\mathcal{I}$ and $\mathcal{I}_1$ with alphabets Σ and Σ_1, respectively, and computes a contract $\mathcal{I}'$ with alphabet $\Sigma' = \Sigma \cup \Sigma_1$. The output of the quotient, denoted $\mathcal{I} \,/\, \mathcal{I}_1$, has the property that its composition with $\mathcal{I}_1$ conforms to $\mathcal{I}$, and is maximal for this property in the conformance order:

$$\forall \mathcal{I}'. \quad \mathcal{I}' \parallel \mathcal{I}_1 \preceq \mathcal{I} \iff \mathcal{I}' \preceq \mathcal{I} \,/\, \mathcal{I}_1. \tag{1}$$

Suppose $\mathcal{I}$ corresponds to a top-level specification, and $\mathcal{I}_1$ to the specification of a component that will be used in the design. The quotient $\mathcal{I} \,/\, \mathcal{I}_1$ yields the specification of the functionality that $\mathcal{I}_1$ is missing for it to conform to $\mathcal{I}$. Thus, the quotient is key in the decomposition of specifications.

We use the language of categories to describe some transformations between contracts. Categories are composed of objects and arrows between these objects. For instance, in the category of sets, objects are sets, and arrows are functions between sets. Functors are transformations between categories; they map objects to objects and arrows to arrows. Since refinement is a partial order for contracts, we can speak about a category of contracts, in which objects are contracts and an arrow from contract $\mathcal{I}$ to contract $\mathcal{I}'$ exists if and only if $\mathcal{I} \preceq \mathcal{I}'$. We use the language of category theory to point out that some operations we have discussed have deep connections to each other (i.e., are not arbitrary definitions). For an in-depth treatment of category theory, see Mac Lane [30].

Let $\mathcal{I}_1$ be a contract. Let $C(\mathcal{I}) = \mathcal{I} \parallel \mathcal{I}_1$ be an endofunctor (i.e., a transformation of objects within the same category) in the category of contracts. Let $Q(\mathcal{I}) = \mathcal{I} \,/\, \mathcal{I}_1$ be another endofunctor. Then the definition (1) can be represented graphically as the universal property that the arrow α exists if and only if β exists in the diagram shown in Figure 4(a). This means that C is a left adjoint to the

(a) Composition and quotient (b) Merging and separation

Fig. 4. Let I_1 be a contract and define the functors $C\,I = I \parallel I_1$, $\mathcal{M}\,I = I \cdot I_1$, $\mathcal{Q}\,I = I \,/\, I_1$, and $\mathcal{S}\,I = I \div I_1$. Then C is a left adjoint of $\mathcal{Q}$, and $\mathcal{M}$ is a right adjoint of $\mathcal{S}$.

functor $\mathcal{Q}$. From this universal property of the quotient, we can derive a closed-form expression which permits its calculation:

Theorem 3.11 (Contract Quotient). *Let $\mathcal{I}$ and $\mathcal{I}_1$ be contracts. Then $\mathcal{I}_q$ is the quotient between contracts $\mathcal{I}$ and $\mathcal{I}_1$, written $\mathcal{I}_q = \mathcal{I} \,/\, \mathcal{I}_1$, if and only if*

$$\mathcal{G}_q = \mathcal{G} \cap \mathcal{R}_1 \cup \overline{(\mathcal{R} \cap \mathcal{G}_1)} \quad \textit{and} \quad \mathcal{R}_q = \mathcal{R} \cap \mathcal{G}_1.$$

There exists a dual operation related to merging. We call *separation* the operation that allows us to separate a viewpoint from a given merged contract. Given a merged contract $\mathcal{I}$ and a viewpoint $\mathcal{I}_1$, the separation $\mathcal{I} \div \mathcal{I}_1$ is defined as follows:

$$\forall \mathcal{I}'. \quad \mathcal{I} \preceq \mathcal{I}' \cdot \mathcal{I}_1 \iff \mathcal{I} \div \mathcal{I}_1 \preceq \mathcal{I}'. \tag{2}$$

In the category of contracts, if we let $\mathcal{M}$ be the endofunctor $\mathcal{M}(\mathcal{I}) = \mathcal{I} \cdot \mathcal{I}_1$ and $\mathcal{S}(\mathcal{I}) = \mathcal{I} \div \mathcal{I}_1$, we observe that the given universal property (2) means that, in Figure 4(b), arrow γ exists if and only if arrow δ exists. It follows that S is a left adjoint of the functor M. From the universal property of separation, we can obtain an explicit form that enables its computation:

Theorem 3.12 (Separation). *Let $\mathcal{I}$ and $\mathcal{I}_1$ be contracts. Then $\mathcal{I}_s$ is the separation of contracts $\mathcal{I}$ and $\mathcal{I}_1$, written $\mathcal{I}_s = \mathcal{I} \div \mathcal{I}_1$, if and only if*

$$\mathcal{G}_s = \mathcal{G} \cap \mathcal{R}_1 \quad \textit{and} \quad \mathcal{R}_s = \mathcal{R} \cap \mathcal{G}_1 \cup \overline{(\mathcal{G} \cap \mathcal{R}_1)}.$$

Note the duality between quotient and separation. Suppose we are given a high level specification I of a design, and the specification I' of a component which will be used in the design. The quotient represents the most relaxed missing specification, I/I', such that this missing specification in composition with I' refines I. Merging, on the contrary, works as follows: suppose we are given a specification I and suppose we are told that a specification I' is part of a *covering* of I (i.e., a set of specifications whose merging *is refined* by I); separation gives us the strictest specification which, when merged with I', is refined by I. In other words, quotient is used to find decompositions, while separation is used to find coverings (e.g., abstractions) of specifications.

To illustrate quotient and separation, suppose we have a top level specification $\mathcal{I} = (\mathcal{R}, \mathcal{G}_1 \cap \mathcal{G}_2 \cup \overline{\mathcal{R}})$. This top level specification requires environments to satisfy $\mathcal{R}$, and components to satisfy $\mathcal{G}_1 \cap \mathcal{G}_2 \cup \overline{\mathcal{R}}$. Suppose an existing component satisfies the contract $\mathcal{I}' = (\mathcal{R}', \mathcal{G}_1 \cup \overline{\mathcal{R}'})$, where $\mathcal{R} \subseteq \mathcal{R}'$, i.e., this component provides part of the requirements of the top-level contract. We expect the quotient to tell that we need another component which implements the guarantees $\mathcal{G}_2$. The quotient yields

$$\mathcal{I} \,/\, \mathcal{I}' = (\mathcal{R} \cap \mathcal{G}_1, \mathcal{G}_2 \cup \overline{\mathcal{R} \cap \mathcal{G}_1}),$$

as we expected. Note that the quotient allows its implementations to make use of the guarantees $\mathcal{G}_1$ as an assumption. In other words, the quotient allows its implementations to expect that the components satisfying the specification $\mathcal{I}'$ will do their job.

Table 1. Behavior of Composition, Quotient, Merging, and Separation with Respect to the Distinguished Elements of the Theory of Contracts

$\mathcal{I} \parallel \bot = \bot$	$\mathcal{I} \parallel \top = (\overline{\mathcal{G}}, \mathcal{G})$	$\mathcal{I} \parallel \mathbb{1} = \mathcal{I}$
$\mathcal{I} \cdot \bot = (\mathcal{R}, \overline{\mathcal{R}})$	$\mathcal{I} \cdot \top = \top$	$\mathcal{I} \cdot \mathbb{1} = \mathcal{I}$
$\mathcal{I} \,/\, \bot = \top$	$\mathcal{I} \,/\, \top = (\mathcal{R}, \overline{\mathcal{R}})$	$\mathcal{I} \,/\, \mathbb{1} = \mathcal{I}$
$\bot \,/\, \mathcal{I} = (\mathcal{G}, \overline{\mathcal{G}})$	$\top \,/\, \mathcal{I} = \top$	$\mathbb{1} \,/\, \mathcal{I} = (\mathcal{G}, \mathcal{R})$
$\mathcal{I} \div \bot = (\overline{\mathcal{G}}, \mathcal{G})$	$\mathcal{I} \div \top = \bot$	$\mathcal{I} \div \mathbb{1} = \mathcal{I}$
$\bot \div \mathcal{I} = \bot$	$\top \div \mathcal{I} = (\overline{\mathcal{R}}, \mathcal{R})$	$\mathbb{1} \div \mathcal{I} = (\mathcal{G}, \mathcal{R})$

Now consider the following application of separation. Instead of looking for decompositions of a specification, we look for coverings of a specification. Suppose we have a top-level specification $\mathcal{I}' = (\mathcal{R}, \mathcal{G} \cup \overline{\mathcal{R}})$ that we wish to implement, say, through synthesis. Suppose the implementation resulting from synthesis has the specification $\mathcal{I} = (\mathcal{R} \cap \mathcal{R}_e, \mathcal{G}' \cup \overline{\mathcal{R} \cap \mathcal{R}_e})$, where $\mathcal{G}' \cup \overline{\mathcal{R} \cap \mathcal{R}_e} \subseteq \mathcal{G} \cup \overline{\mathcal{R} \cap \mathcal{R}_e}$. That is, the implementation fails to be a refinement of the top-level specification because the implementation uses more assumptions. Note that this scenario is rather typical: top-level specifications often fail to include assumptions which are necessary for an implementation to work. These additional requirements are captured by $\mathcal{R}_e$. We wish to compute the smallest specification we need to add to $\mathcal{I}'$ so that $\mathcal{I}'$ merged with this missing specification covers $\mathcal{I}$. Computing separation yields

$$\mathcal{I} \div \mathcal{I}' = (\mathcal{R}_e \cup \overline{\mathcal{R}}, \mathcal{G}' \cap \mathcal{R} \cup \overline{(\mathcal{R}_e \cup \overline{\mathcal{R}})}).$$

We can abstract this separation result to the contract $(\mathcal{R}_e, \mathcal{G}' \cup \overline{\mathcal{R}_e})$ (verification that this is indeed an abstraction is left to the reader). This contract adds the missing requirements and enforces the stricter guarantees $\mathcal{G}'$. If the previous guarantees were acceptable, one can abstract this contract even further to $(\mathcal{R}_e, \mathcal{B}(\Sigma))$, where $\mathcal{B}(\Sigma)$ is the set of all behaviors under consideration.

The operations we have introduced have identities that characterize them in the contract lattice.

Definition 3.13 (Composition and Merging Identity). A composition identity, denoted $\mathbb{1}_c$, is a contract such that $\mathcal{I} \parallel \mathbb{1}_c = \mathbb{1}_c \parallel \mathcal{I} = \mathcal{I}$. Likewise, a merging identity, denoted $\mathbb{1}_m$, satisfies $\mathcal{I} \cdot \mathbb{1}_m = \mathbb{1}_m \cdot \mathcal{I} = \mathcal{I}$.

The definition of the identities does not imply their uniqueness. The following lemma settles this issue:

Lemma 3.14. *Let Σ be the union of all alphabets over which components are defined. The contract identities just introduced and the contracts $\bot$ and $\top$ have the following explicit forms: $\top = (\emptyset, \mathcal{B}(\Sigma))$, $\bot = (\mathcal{B}(\Sigma), \emptyset)$, and $\mathbb{1}_m = \mathbb{1}_c = (\mathcal{B}(\Sigma), \mathcal{B}(\Sigma))$. Since both identities are equal, we call $\mathbb{1} = \mathbb{1}_c = \mathbb{1}_m$ the identity.*

How do these distinguished elements behave with respect to the contract operations? These relations are shown in Table 1. We observe in this table that taking the quotient or separation from the identity results in a contract with flipped requirements and guarantees. This behavior motivates the following definition:

Definition 3.15 (Reciprocal). Let $I = (\mathcal{R}, \mathcal{G})$. Its reciprocal, denoted I^{-1}, is given by $I^{-1} = \mathbb{1} \,/\, \mathcal{I} = \mathbb{1} \div \mathcal{I} = (\mathcal{G}, \mathcal{R})$.

Finally, let $\mathcal{I} = (\mathcal{R}, \mathcal{G})$ be a contract defined over an alphabet Σ. The following table provides some identities pertaining merging and composition and their adjoint operations:

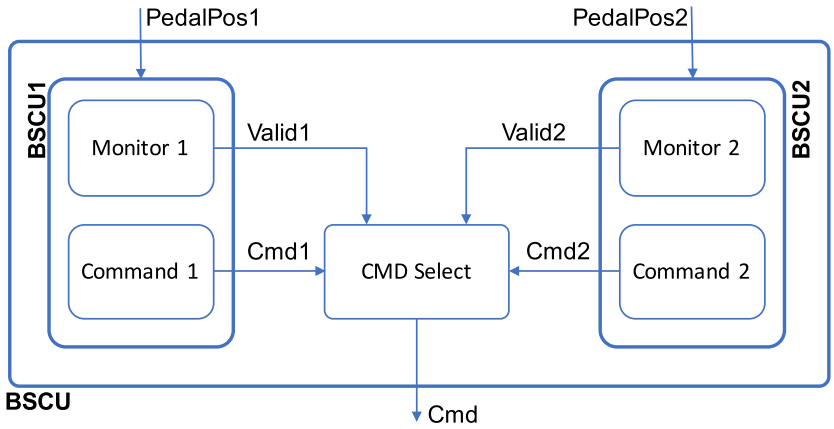

Fig. 5. The Break System Control Unit [13].

$$\begin{array}{lll} \mathcal{I} \parallel \mathcal{I} = \mathcal{I} & \mathcal{I} \,/\, \mathcal{I} = (\mathcal{R} \cap \mathcal{G}, \mathcal{B}(\Sigma)) & \mathcal{I} \,/\, \mathcal{I}' = \mathcal{I} \cdot \mathcal{I}'^{-1} \\ \mathcal{I} \cdot \mathcal{I} = \mathcal{I} & \mathcal{I} \div \mathcal{I} = (\mathcal{B}(\Sigma), \mathcal{R} \cap \mathcal{G}) & \mathcal{I} \div \mathcal{I}' = \mathcal{I} \parallel \mathcal{I}'^{-1} \end{array}$$

The operation of reciprocal allows us to create a contract representing the perspective of the environment in which the design operates (as the reciprocal flips requirements and guarantees). We obtained several identities showing how the reciprocal interacts with the other contract operations. It will be future work to better understand the role of the reciprocal in system design methodologies.

3.6 Multiviewpoint Design

We can use the device of merging to analyze systems described under different viewpoints, and derive stronger combined results. To illustrate the procedure, we employ the case study introduced by Damm et al. [13] and there solved by manually combining a timing and a safety specification. We show that the application of our operators produces a more accurate result, which refines the one derived there.

The example consists of a redundant wheel brake system composed of a dual *Break System Control Unit* (BSCU) and a Hydraulic actuator. We will be concerned mainly with the BSCU, which is shown schematically in Figure 5. The two units receive information regarding the position of the break pedal, and deliver a break command. The switch determines which of the two versions of the command to forward on the basis of the valid signals coming from the monitoring components. Without going into the details, the unit can sustain a single fault (at least one unit will have a valid signal), and must produce a command within 5 ms from a change in the pedal position. The specification makes use of the *Requirements Specification Language*, a natural-looking formal pattern-based assertion language. The timing and the safety analysis produce two contracts of the form $\mathcal{I} = ((H, R), (A, G))$, where H is the universe of behaviors (the requirements make no assumptions), R are the strong assumptions, A are the weak assumptions and G are the guarantees[1]. The contracts for the above properties are:

[1]In the original paper [13], the strong assumptions were denoted by A, the weak assumptions by B and the guarantees by G.

Safety contract
R: *fail*(PedalPos1) **and** *fail*(PedalPos2) **do not occur**
A: *No double fault*
G: Valid1 **or** Valid2

Timing contract
R: PedalPos1 == PedalPos2
A: Valid1 **or** Valid2
G: **Delay from** *change*(PedalPos1) **or***change*(PedalPos2) **to** *change*(Cmd) **within** [0, 5] ms

Merging the two contracts requires taking the intersection of the requirements after inverse projection, as well as the intersection of the guarantees, which we extend with the complement of the weak assumptions to obtain a canonical form. The addition of the complement of the requirements is not shown for brevity, as the semantics of the contract is not changed.

Merged contract
R: *fail*(PedalPos1) **and** *fail*(PedalPos2) **do not occur and** PedalPos1 == PedalPos2
A: null
G: ((Valid1 **or** Valid2) **or** $\overline{\textit{No double fault}}$) **and** ((**Delay…**) **or** $\overline{(\text{Valid1 } \textbf{or } \text{Valid2})}$)

By logical manipulation, simplification and extracting the weak assumptions, we obtain:

Merged contract
R: *fail*(PedalPos1) **and** *fail*(PedalPos2) **do not occur and** PedalPos1 == PedalPos2
A: (Valid1 **or** Valid2) **or** *No double fault*
G: ((Valid1 **or** Valid2) **and** (**Delay…**))

The result shows that under the assumption that the inputs are correct, and if there is no double fault, the system guarantees the stated delay between the change of the pedal position and the braking command. While the results in the original paper are only informally stated, they do appear to lack the first term of the weak assumption, making ours a more refined contract (it accepts more environments). The first term is necessary, as the original specification only expressed the forward implication (no double fault implies at least one valid signal is asserted), but not the converse. Having a coherent theory is therefore fundamental to ensure correctness, and to cover all corner cases. In this case, the risk is to have contract satisfaction despite a possible double failure. This may or may not be important in the context of the system, but we believe the designer should be aware of the possibility and take action as required.

Separation can be used to go back to the individual contracts. For instance, if we separate the timing viewpoint and project away its variables we reconstruct the safety contract:

Separating timing viewpoint from merged contract
R: *fail*(PedalPos1) **and** *fail*(PedalPos2) **do not occur**
A: *No double fault* **or** (Valid1 **or** Valid2)
G: (Valid1 **or** Valid2)

Again, the result is a refinement of the original contract, recovering the implicit single implication.

4 RELATED WORK

The notion of contract as used in our framework derives from the theory of abstract data types and was first suggested by Meyer in the context of the programming language Eiffel [32], following the original ideas introduced by Floyd and Hoare [19, 22] to assign logical meaning to sequential imperative programs in the form of triples of assertions. In his work, Meyer introduces *preconditions* and *postconditions* as assertions or specifications for the methods of a class, and *invariants* for the class itself. Similar ideas were already present in seminal work by Dijkstra [16] and Lamport [24] on *weakest preconditions* and *predicate transformers* for sequential and concurrent programs, and in more recent work by Back and von Wright, who introduce contracts in the *refinement calculus* [2]. Contracts are composed of *assertions* (higher-order state predicates) and *state transformers*. These contracts are of a very different nature, since there is no clear indication of the role (assumption or promise) a state predicate or a state transformer may play. This formalism is best suited to reason about discrete, un-timed process behavior.

Our work is based, in particular, on three models that were subsequently developed in the literature. The work of Dill on asynchronous trace structures was the first to differentiate between acceptable and non-acceptable uses of a component [17]. Safe substitutability is expressed as trace containment between the successes and failures of the specification and the implementation. The conditions obtained by Dill are equivalent to requiring that the implementation weaken the assumptions of the specification while strengthening the promises. The notion of refinement and composition are the same as for maximal interfaces, where composition is taken as the product with requirement relaxation. We have shown in this paper how this is only one possible choice for these operators, by working in a more general settings. Wolf later extended the same technique to a discrete synchronous model [40]. Finally, Process Spaces [33] is a more general model proposed by Negulescu following the work of Dill and Wolf, and is based on processes equivalent to our maximal interfaces. The notion of conformance corresponds to the one proposed here. While several operators are introduced, Process Spaces ignore the issues with inverse projection in conjunction, and weak and strong assumptions. Process Spaces define the operations of product and exclusive sum, which are syntactically the same as our operations of parallel composition and merging for contracts, respectively. Product is used to compose design elements, but no insight is given into the use of exclusive sum. In particular, the paper does not address multiple viewpoints for the same design element. Also, this work does not show how these operations relate to the lattice operations generated by the conformance order, as we do in Section 3.4. In process spaces, an operation called reflection is defined. This operation has the same closed form as the operation of reciprocal that we introduced. Nonetheless, while Negulescu *defines* the closed-form expression for reflection, in our development the closed-form of the reciprocal is derived from an expression involving the identity and the quotient. The classic Interface Automata [15] and HRC [7, 14] models are similar to synchronous trace structures, where failures are implicitly all the traces that are not accepted. Thus, the interface is maximal. Composition is defined on automata, rather than on traces, and requires a procedure similar to requirement relaxation in order to maintain maximality. The authors have also extended the approach to several other kinds of behaviors, including resources and asynchronous behaviors [10, 21]. We take these concepts a lot further in this paper to include several other operators.

One of the early foundational works that concerns interface and contract specification is due to Abadi and Lamport, who were first to thoroughly discuss and differentiate between the component guarantees and the environment assumptions [1]. In this work, the authors focus on the formulation of the specification as an *implication*, in the sense described in Section 3.1. In particular, while a specification is allowed to make assumptions, it is *not* interpreted as constraining

the environment, or else the specification is considered unrealizable. In this paper we formalize the difference between these two interpretations as weak and strong assumptions. The main result of Abadi and Lamport is a full set of proof rules that show when the parallel composition of components satisfies a given property, under a set of assumptions. These proof rules have later been reformulated in similar ways in several other contract models and tools [3, 12, 18, 20]. Composition, expressed as intersection of behaviors, takes primarily the view of the component. Our work builds on these concepts, and evolves in complementary directions. First, we emphasize an approach to composition that is balanced between component and environment, leading to the notions of maximal and symmetric interfaces. More importantly, while Abadi and Lamport touch upon the need for inverse projection during composition, they do not discuss the operation of conjunction and all that it entails. They also discusses the validity of the circular reasoning principle when liveness properties are considered. Our work is largely orthogonal to these aspects. Dill uses failures in infinite traces to express causality and liveness [17], a method that could be reflected in contracts. Also, proof rules similar to those proposed by Graf et al. [20] in HRC could be employed to determine how and when circular reasoning allows a component and its environment to be refined concurrently, each relying on the abstract description of its context, and therefore prove conformance.

Bauer et al. [3] present a meta-theory similar in spirit to our work. The objective is to provide a method with which to construct a *contract* framework given a *specification* (or component, with our terminology) framework with sufficient reasonable properties. The work focuses on the relation of refinement and defines operators for composition, conjunction and quotient. In particular, they show how to constructively define the composition operator. Their method is based on the use of canonical forms, and treats environments and implementations asymmetrically. We follow a similar approach, and start from a generic component model to build interface and contract models with increasing levels of structure. Our objectives, however, are complementary. We focus in particular on dissecting the interface models to thoroughly understand the role of each of its parts. Unlike the cited work, we treat environments and implementations on an equal ground, to include methodologies that favor both component optimization and component reuse. We show the pitfalls associated with applying inverse projection and union and propose an effective solution, using merging (see Section 3.3). In particular, we do so from a purely semantic point of view, instead of employing the traditional transition systems.

Chilton et al. [11] develop an algebraic theory of interface automata which is also useful to shed light on the properties of the operators and relations. The formalism is reminiscent of Dill's trace structures, and extends that work with additional operators, such as quotient. The authors also address issues of progress in the context of finite traces, unlike trace structures which use infinite traces. Of particular interest is the definition of refinement, which allows the refining component to have signatures with different sets of inputs and outputs. Consequently, conjunction can also be defined on components with different signatures. This facilitates a multiple viewpoint approach. However, the issue of inverse projection is not resolved, and the conjunction "yields the coarsest component that will work in any environment safe for at least one of its operands" [11]. Hence, a viewpoint that makes no assumption will necessarily wipe out the assumptions of all other viewpoints, as discussed in Section 3.3. We believe the merging operator that we introduce could be applied to their model to properly handle these cases.

Most recently, Benveniste et al. have introduced a meta-theory of contracts to frame several models in the same formalism [6], and discuss their operators. We also reason about the contract operators in a general settings, and cover aspects, such as completion and merging, which were not analyzed previously. Our aim, however, is not to construct another or better meta-theory.

Instead, the formalism that we use is largely derived from previous models (such as the mentioned HRC) and has been adapted to simplify the algebraic expression of our main results. The quotient operation we discussed is the same as that defined in terms of its universal property (1) in Benveniste et al. [6]. The closed-form expression for the quotient operation of contracts was introduced in [23]. As far as we know, the separation operation we discussed has no precedent in the literature.

Tripakis et al. [39] also study the connection between different kinds of interface specification. In particular, they show how to transform Relational Interfaces [38], which are not input complete (or receptive), into an equivalent set of input complete specifications, in order to avoid game-theoretic methods and have a more efficient analysis. We believe this procedure is akin to going from Interface Automata to Trace Structures. Similarly, Carmona and Kleijn [9] explore the issue of compatibility in a general multi-component settings. This work deals primarily with questions of receptiveness, progress and deadlock freedom. However, the authors do not develop a full interface or contract model, but express assumptions implicitly in terms of the actions which are enabled at each state of the components. An analysis of these aspects, which are orthogonal to the work presented in this paper, and their application to our context are part of our future work.

Damm et al. [13] introduce the distinction between weak and strong assumptions, which we extend using the concept of implication. Mangeruca et al. [31] use a similar notion, called *precondition*, to define the conditions under which the promises must hold, in a form similar to implications. The authors use this concept to define the *completeness* of a contract relative to the requirements, and avoid implementations that vacuously satisfy their contract. The formalism is also used to define extensions of the contract, by properly combining the promises and their preconditions. This differs from our notion of completion, and is used to help designers cope with evolving specifications. The authors also provide an operator to override a promise by another promise. These extensions naturally fit in our formalism when promises are defined as implications. Using our theory, the same operators can similarly be extended to cover also the environment requirements.

5 CONCLUSIONS

Starting from a traditional component model, we used a layered approach to incrementally construct interface and contract models with increasing level of structure. While doing so, we discussed expressiveness, equivalence and closure of the operators, including weak and strong assumptions, and a characterization of contract models as symmetric interfaces. We then focused on how to combine properly contracts related to different viewpoints, and introduced the operation of merging as a component of the greatest lower bound. We showed that parallel composition is symmetric in terms of the least upper bound. We then showed how the adjoint operations of quotient and separation can be used to decompose specifications. A use case shows that the formalization properly accounts for different viewpoints when merging specifications, avoiding potential errors.

In the near future, we plan to develop a thorough treatment of the questions of consistency and compatibility, and a study of "don't care" conditions, which could be useful in the context of synthesis. In this paper we have laid down the essential definitions and mechanisms that are required to express these conditions. In particular, "don't care" conditions could be useful not only during synthesis optimization, but also as a way to shift responsibilities between environment and implementation, thus facilitating the construction of compatible sets of components.

ACKNOWLEDGMENTS

This work was supported in part by NSF Contract CPS Medium 1739816.

REFERENCES

[1] Martín Abadi and Leslie Lamport. 1993. Composing specifications. *ACM Transactions on Programming Languages and Systems* 15, 1 (January 1993), 73–132.

[2] Ralph-Johan Back and Joakim von Wright. 2000. Contracts, games, and refinement. *Information and Communication* 156 (2000), 25–45.

[3] Sebastian S. Bauer, Alexandre David, Rolf Hennicker, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. 2012. Moving from specifications to contracts in component-based design. In *Proceedings of the* 15^{th} *International Conference on Fundamental Approaches to Software Engineering (FASE'12)*. Springer-Verlag, Tallinn, Estonia, 43–58.

[4] Albert Benveniste, Benoît Caillaud, Luca Carloni, Paul Caspi, and Alberto Sangiovanni-Vincentelli. 2008. Composing heterogeneous reactive systems. *ACM Transactions on Embedded Computing Systems* 7, 4 (2008), 43:1–43:36.

[5] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. 2008. Multiple viewpoint contract-based specification and design. In *Formal Methods for Components and Objects,* 6^{th} *International Symposium (FMCO'07), Amsterdam, The Netherlands, October 24–26, 2007, Revised Papers*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Lecture Notes in Computer Science, Vol. 5382. Springer Verlag, Berlin Heidelberg, 200–225.

[6] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto L. Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen. 2018. *Contracts for System Design*. Foundations and Trends in Electronic Design Automation, Vol. 12. now publishers.

[7] Luca Benvenuti, Alberto Ferrari, Leonardo Mangeruca, Emanuele Mazzi, Roberto Passerone, and Christos Sofronis. 2008. A contract-based formalism for the specification of heterogeneous systems. In *Proceedings of the Forum on Specification & Design Languages (FDL08)*. Stuttgart, Germany, 142–147.

[8] Luca Benvenuti, Alberto Ferrari, Emanuele Mazzi, and Alberto L. Sangiovanni Vincentelli. 2008. Contract-based design for computation and verification of a closed-loop hybrid system. In *Hybrid Systems: Computation and Control*, Magnus Egerstedt and Bud Mishra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 58–71.

[9] Josep Carmona and Jetty Kleijn. 2013. Compatibility in a multi-component environment. *Theoretical Computer Science* 484 (May 2013), 1–15.

[10] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Marielle Stoelinga. 2003. Resource interfaces. In *Proceedings of the Third Annual Conference on Embedded Software (EMSOFT'03) (Lecture Notes in Computer Science)*, Vol. 2855. Springer, 117–133.

[11] Chris Chilton, Bengt Jonsson, and Marta Kwiatkowska. 2014. An algebraic theory of interface automata. *Theoretical Computer Science* 549 (September 2014), 146–174.

[12] Alessandro Cimatti and Stefano Tonetta. 2015. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming* 97, Part 3 (2015), 333–348.

[13] Werner Damm, Hardi Hungar, Bernhard Josko, Thomas Peikenkamp, and Ingo Stierand. 2011. Using contract-based component specifications for virtual integration testing and architecture design. In *Design, Automation Test in Europe Conference Exhibition (DATE11)*. Grenoble, France, 1–6.

[14] Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Böde. 2005. Boosting re-use of embedded automotive applications through rich components. In *Foundations of Interface Technologies (FIT'05)*.

[15] Luca de Alfaro and Thomas A. Henzinger. 2001. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, 109–120.

[16] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (August 1975), 453–457.

[17] David L. Dill. 1989. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press.

[18] Iulia Dragomir, Iulian Ober, and Christian Percebois. 2015. Contract-based modeling and verification of timed safety requirements within SysML. *Software & Systems Modeling* (2015), 1–38.

[19] Robert W. Floyd. 1967. Assigning meaning to programs. In *Proceedings of Symposium on Applied Mathematics*, Vol. 19. 19–32.

[20] Susanne Graf, Roberto Passerone, and Sophie Quinton. 2014. Contract-based reasoning for component systems with rich interactions. In *Embedded Systems Development: From Functional Models to Implementations*, Alberto L. Sangiovanni-Vincentelli, Haibo Zeng, Marco Di Natale, and Peter Marwedel (Eds.). Embedded Systems, Vol. 20. Springer New York, Chapter 8, 139–154.

[21] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2005. Permissive interfaces. In *Proceedings of the* 13^{th} *Annual Symposium on Foundations of Software Engineering (FSE'05)*. ACM Press, 31–40.

[22] Charles A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[23] Íñigo Íncer Romeo, Alberto Sangiovanni-Vincentelli, Chung-Wei Lin, and Eunsuk Kang. 2018. Quotient for assume-guarantee contracts. In *16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*.

[24] Leslie Lamport. 1990. win and sin: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 396–428.

[25] Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. 2006. Interface input/output automata. In *14th International Symposium on Formal Methods, FM'06 (Lecture Notes in Computer Science)*, Vol. 4085. Springer, 82–97.

[26] Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. 2007. Modal I/O automata for interface and product line theories. In *Programming Languages and Systems, 16th European Symposium on Programming, (ESOP'07) (Lecture Notes in Computer Science)*, Vol. 4421. Springer, 64–79.

[27] Hoa Thi Thieu Le, Roberto Passerone, Uli Fahrenberg, and Axel Legay. 2016. A tag contract framework for modeling heterogeneous systems. *Science of Computer Programming* 115–116 (2016), 225–246.

[28] Edward A. Lee and Alberto Sangiovanni-Vincentelli. 1998. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems* 17, 12 (1998), 1217–1229.

[29] Edward A. Lee and Yuhong Xiong. 2004. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing Journal* 16, 3 (2004), 210–237.

[30] Saunders Mac Lane. 1998. *Categories for the Working Mathematician* (2nd ed.). Vol. 5. New York, NY: Springer. xii + 314 pages.

[31] Leonardo Mangeruca, Orlando Ferrante, and Alberto Ferrari. 2013. Formalization and completeness of evolving requirements using contracts. In *Proceedings of the* 8^{th} *IEEE International Symposium on Industrial Embedded Systems (SIES'13)*. Porto, Portugal, 120–129.

[32] Bertrand Meyer. 1992. Applying "design by contract". *IEEE Computer* 25, 10 (October 1992), 40–51.

[33] Radu Negulescu. 2000. Process spaces. In *CONCUR 2000 — Concurrency Theory*, Catuscia Palamidessi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 199–213.

[34] Ulrik Nyman. 2008. *Modal Transition Systems as the Basis for Interface Theories and Product Lines*. Ph.D. Dissertation. Aalborg University, Department of Computer Science.

[35] Roberto Passerone, Íñigo Íncer Romeo, and Alberto L. Sangiovanni-Vincentelli. 2019. *Contract model operators for composition and merging: extensions and proofs*. Technical Report DISI-19-004. Dipartimento di Ingegneria e Scienza dell'Informazione, University of Trento.

[36] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. 2011. A modal interface theory for component-based design. *Fundamenta Informaticae* 108, 1–2 (2011), 119–149.

[37] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, and Roberto Passerone. 2009. Why are modalities good for interface theories? In *Proceedings of the Ninth International Conference on Application of Concurrency to System Design (ACSD'09)*. Augsburg, Germany, 119–127.

[38] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. 2011. A theory of synchronous relational interfaces. *ACM Transactions on Programming Languages and Systems* 33, 4 (July 2011).

[39] Stavros Tripakis, Christos Stergiou, Manfred Broy, and Edward A. Lee. 2013. Error-completion in interface theories. In *Model Checking Software*, Ezio Bartocci and C. R. Ramakrishnan (Eds.). Lecture Notes in Computer Science, Vol. 7976. Springer Berlin Heidelberg, 358–375.

[40] Elizabeth S. Wolf. 1995. *Hierarchical Models of Synchronous Circuits for Formal Verification and Substitution*. Ph.D. Dissertation. Department of Computer Science, Stanford University.

Received April 2019; revised June 2019; accepted July 2019