Error-Completion in Interface Theories*

Stavros Tripakis[1], Christos Stergiou[1], Manfred Broy[2], and Edward A. Lee[1]

[1] University of California, Berkeley
[2] TU Munich

Abstract. Interface theories are compositional theories where components are represented as abstract, formal interfaces which describe the component's input/output behavior. A key characteristic of interface theories is that interfaces are non-input-complete, meaning that they allow specification of *illegal* inputs. As a result of non-input-completeness, interface theories use game-theoretic definitions of composition and refinement, which are both conceptually and computationally more complicated than standard notions of composition and refinement that work with input-complete models. In this paper we propose a lossless transformation, called error-completion, which allows to transform a non-input-complete interface into an input-complete interface while preserving and allowing to retrieve completely the information on illegal inputs. We show how to perform composition of relational interfaces on the error-complete domain. We also show that refinement of such interfaces is equivalent to standard implication of their error-completions.

1 Introduction

Interface theories such as the theory of *interface automata* are compositional theories proposed by Alfaro and Henzinger in the early 2000s [9,10], and since then studied extensively (e.g., see [7,11,20,22,12]). Generally speaking, an interface theory provides the following:

- A notion of *interface* which is an abstract, formal description of a component's interface behavior. Different notions of interfaces exist in the literature, e.g., in [9] interfaces are automata, while in [22] they are static or dynamic logical formulas.
- One or more *composition operators* which allow to compose interfaces and form new interfaces. Different notions of composition are available depending on the theory, e.g., asynchronous composition in [9], synchronous composition in [22].
- A *refinement relation* which is a binary relation between interfaces.

* This work was supported in part by the iCyPhy Center (supported by IBM and United Technologies), the CHESS Center (supported by awards NSF #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), NRL #N0013-12-1-G015, Bosch, National Instruments, and Toyota), and by the NSF Expeditions in Computing project *ExCAPE: Expeditions in Computer Augmented Program Engineering*.

– A set of theorems, typically including:
  - *Preservation of refinement by composition*, e.g., if A' refines A and B' refines B, then the composition of A' and B' refines the composition of A and B.
  - *Preservation of certain properties by refinement*, e.g., if A' refines A and A satisfies, say, a safety property p, then A' also satisfies p.

Theorems such as the above support incremental design methodologies and reconfigurability. For instance, if we have shown that a certain system consisting of the composition of A and B satisfies p, and later A needs to be replaced by A', proving that A' refines A is sufficient to ensure that p will not be compromised by such a replacement, i.e., it will continue to hold on the new system composed of A' and B.

A key characteristic of interface theories such as interface automata [9] and relational interfaces [22] is that interfaces in these theories are generally *non-input-complete*, that is, they may specify that certain inputs are *illegal*.[3] This is in contrast with other compositional theories such as I/O automata [16], FOCUS [5,6], and reactive modules [2], where specifications are assumed to be input-complete. As argued in [22], non-input-completeness is essential to obtain a theory which allows a lightweight verification methodology, akin to type-checking. In particular, non-input-completeness allows to define semantic or *behavioral* notions of interface compatibility. These go beyond syntactic compatibility notions like correct port matching.

As a result of non-input-completeness, and the fact that components are generally non-deterministic (meaning that for a given input they may produce different outputs) the definitions of composition and refinement in interface theories are *game-theoretic* in nature.

Although game-theoretic notions such as demonic composition and alternating refinement are relatively well-understood, they are more complex than the corresponding standard notions, and generally involve computing strategies in a two-player game [3,10,8,22]. It makes sense, then, to ask whether there exists a transformation from non-input-complete to input-complete interfaces, which allows to reduce the above operations into standard composition and refinement.

In this paper we answer the above question in the affirmative for the setting of relational interfaces of [22]. In particular, we propose a *lossless* transformation called *error-completion*. Given a (generally non-input-complete) interface ϕ, error-completion returns an input-complete interface $\mathsf{EC}(\phi)$, with an additional boolean output variable which captures illegal inputs. The main results of the paper are:

– We show that $\mathsf{EC}(\phi)$ does not lose any information contained in ϕ, by providing an inverse transformation EC^{-1} and showing that $\mathsf{EC}^{-1}(\mathsf{EC}(\phi)) \equiv \phi$ for all ϕ.

[3] We use the term *input-complete* following [22]. Other terms used in the literature are *input-enabled*, *receptive*, or *total*.

– We show that serial and parallel composition of relational interfaces can be performed in the error-complete domain, and the result can be transformed backwards using EC^{-1} to obtain the equivalent composition in the original domain.
– We show that the (alternating) refinement relation $\phi_1 \sqsubseteq \phi_2$ is equivalent to the standard implication $\mathsf{EC}(\phi_1) \rightarrow \mathsf{EC}(\phi_2)$.

We point out that error-completion is discussed in [22]. However, the definition of error-completion given in [22] is not satisfactory because, as already observed in [22], it does not allow to reduce refinement checking of general relational interfaces to checking standard implication on their error-completed versions. In this paper we propose a new definition of error-completion which achieves this, among other properties. We also provide an in-depth discussion of error-completion and possible alternatives.

The rest of the paper is organized as follows. Section 2 summarizes the theory of [22]. Section 3 describes error-completion. Section 4 discusses possible extensions. Section 5 presents related work. Section 6 concludes the paper.

2 Preliminaries

In this section we summarize the relational interface framework developed in [22].

2.1 Relational Interfaces

Let V be a finite set of variables. A property over V is a first-order logic formula ϕ such that any free variable of ϕ is in V. We write $\mathcal{F}(V)$ for the set of all properties over V. Assuming that every variable is associated with a certain domain, an assignment over V is a function mapping every variable in V to a certain value in the domain of that variable. The set of all assignments over V is denoted by $\mathcal{A}(V)$.

Assume a component with inputs X and outputs Y. We identify states with observational histories, i.e., a state of the component is an element of $\mathcal{A}(X \cup Y)^*$.

Definition 1 (Relational interface). *A* relational interface *(RI) is a tuple (X, Y, f) where X and Y are two finite and disjoint sets of input and output variables, respectively, and f is a function from states to* contracts*, i.e., for every $s \in \mathcal{A}(X \cup Y)^*$, $f(s) \in \mathcal{F}(X \cup Y)$.*

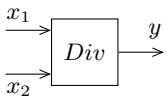
Note that we allow X or Y to be empty. If $X = \emptyset$ then the interface is a *source*. If $Y = \emptyset$ then the interface is a *sink*.

In order to simplify the presentation we will restrict the definitions and the rest of the formalization to the case of stateless interfaces, i.e. interfaces that specify the same contract for each state or input-output history. We also often omit the term relational and speak simply of interfaces, for the sake of brevity.

Definition 2 (Stateless interface). *An interface $I = (X, Y, f)$ is* stateless *iff for all $s, s' \in \mathcal{A}(X \cup Y)^*$, $f(s) = f(s')$.*

For the sake of simplicity, we will specify a stateless interface as a triple (X, Y, ϕ), where $\phi \in \mathcal{F}(X \cup Y)$.

An example of a stateless relational interface is shown in Figure 1. This interface, called Div, is the interface of a component that is supposed to perform division. The component has two inputs x_1 and x_2 and produces the result $y = \frac{x_1}{x_2}$ on its output. There are different properties of this component that one might want to capture in its interface Div. Two possible contracts for Div, ϕ_1 and ϕ_2, are shown in Figure 1. Both specify that input x_2 has to be different than 0. Note that the first contract ϕ_1 completely determines the behavior of the component; it is an example of a *deterministic* contract: given legal inputs, outputs are unique. Contract ϕ_2, on the other hand, only provides guarantees about the sign of the output.



$$\phi_1 : x_2 \neq 0 \wedge x_1 = y \cdot x_2$$
$$\phi_2 : x_2 \neq 0 \wedge (y \geq 0 \equiv (x_1 \geq 0 \wedge x_2 > 0))$$

Fig. 1. Component Div outputs at y the division of its inputs x_1/x_2

The theory does not separate requirements on inputs from guarantees on the outputs. A single formula on input and output variables captures the behavioral specification of a stateless interface. We can however extract the requirements a contract makes on the inputs by existentially quantifying over the output variables.

Definition 3 (Input requirement). *Given a contract $\phi \in \mathcal{F}(X \cup Y)$, the input requirement of ϕ is the formula* $\mathsf{in}(\phi) := \exists Y : \phi$.

A note on notation: if ϕ is a formula over a set of variables V, and $U \subseteq V$, with $U = \{u_1, u_2, ..., u_k\}$, then $\exists U : \phi$ is shorthand notation for $\exists u_1, u_2, ..., u_n : \phi$. Note that U is allowed to be empty. If $U = \emptyset$, then $(\exists U : \phi) \equiv \phi$.

$\mathsf{in}(\phi)$ is a property over X only, and represents the requirements that the contract places on the component inputs. For example, for the division component with contract $\phi \equiv (x_2 \neq 0 \wedge x_2 \cdot y = x_1)$, the input requirement is $\mathsf{in}(\phi) \equiv (\exists y : x_2 \neq 0 \wedge x_2 \cdot y = x_1) \equiv x_2 \neq 0$. Note that if ϕ belongs to a source (that is, if $X = \emptyset$), and ϕ is satisfiable, then $\mathsf{in}(\phi) \equiv \mathsf{true}$. If ϕ belongs to a sink (i.e., if $Y = \emptyset$), then $\mathsf{in}(\phi) \equiv \phi$. In all cases, $\phi \rightarrow \mathsf{in}(\phi)$.

Definition 4 (Input-completeness). *An interface $I = (X, Y, \phi)$ is* input-complete *iff $\mathsf{in}(\phi)$ is valid.*

Going back to the examples of Figure 1, note that $\mathsf{in}(\phi_1) \equiv \mathsf{in}(\phi_2) \equiv x_2 \neq 0$. Therefore, both ϕ_1 and ϕ_2 are not input-complete. If, however, the contract was specified as $\phi_3 \equiv (x_2 \neq 0 \rightarrow x_2 \cdot y = x_1)$, then $\mathsf{in}(\phi_3)$ would be true. ϕ_3 is thus an example of an input-complete contract.

Definition 5 (Well-formedness). *An interface* $I = (X, Y, \phi)$ *is* well-formed *iff* ϕ *is satisfiable.*

At this point it is worth making the following remark. Syntactically, relational interfaces are represented by formulas. Semantically, they are relations between input and output assignments, that is, subsets of $\mathcal{A}(X \cup Y)$ (hence the term *relational*). Clearly, different formulas correspond to the same relation. For example, both $x \wedge \neg x$ and false represent the same relation (in this case the empty set). What we are mainly interested in is the semantics, not the syntax. For formulas ϕ_1 and ϕ_2, we can check whether they represent the same relation by checking whether they are equivalent, $\phi_1 \equiv \phi_2$.

Based on the above discussion, the canonical non-well-formed interface can be represented by false.

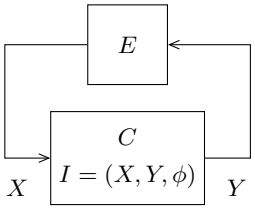


Fig. 2. Component C specified by interface I in feedback with environment E.

A relational interface $I = (X, Y, \phi)$ can be seen as specifying a game between a component and its environment. In Figure 2, the component and the environment are represented by blocks C and E respectively. The game proceeds in a sequence of rounds. At the end of each round, an assignment $a \in \mathcal{A}(X \cup Y)$ is chosen. Typically, the environment plays first and chooses an assignment for the inputs X of the component, $a_X \in \mathcal{A}(X)$. If a_X does not satisfy $\mathsf{in}(\phi)$ then this is not a legal input and the environment loses. Otherwise, the component plays by choosing an assignment for the outputs Y, $a_Y \in \mathcal{A}(Y)$. If (a_X, a_Y) does not satisfy ϕ then this is not a legal output for this input, and the component loses the game. Otherwise, the round is complete, and the game moves to the next round.

2.2 Composition

We can compose two interfaces I_1 and I_2 in series by connecting some of the output variables of I_1 to some of the input variables of I_2. Variables that have the same name are implicitly connected. As it was argued in [22], composition by conjunction of the interface contracts is not sufficient, and instead a "demonic" definition of serial composition needs to be used.

Definition 6 (Serial composition). *Let* $I_1 = (X_1, Y_1, \phi_1)$ *and* $I_2 = (X_2, Y_2, \phi_2)$ *be two interfaces.* I_1 *and* I_2 *are said to be* composable *if* $X_1 \cap X_2 = X_1 \cap Y_2 =$

$Y_1 \cap Y_2 = \emptyset$. *If* I_1 *and* I_2 *are composable, then we can define the serial composition of* I_1 *and* I_2*, denoted* $I_1 \rightsquigarrow I_2$*, as the interface* $I = (X, Y, \phi_1 \rightsquigarrow \phi_2)$ *where* $X = X_1 \cup (X_2 \setminus Y_1)$, $Y = Y_2 \cup Y_1$ *and*

$$\phi_1 \rightsquigarrow \phi_2 := \phi_1 \wedge \phi_2 \wedge \forall Y_1 : \big(\phi_1 \rightarrow \mathsf{in}(\phi_2)\big) \tag{1}$$

It is often convenient to automatically hide the connected outputs $Y_1 \cap X_2$ *right after the composition. For that purpose, we introduce the additional operator of* serial composition with hiding*, denoted* $I_1 \rightsquigarrow^* I_2$*, which defines interface* $I' = (X, Y', \phi_1 \rightsquigarrow^* \phi_2)$*, where* X *is as above,* $Y' = Y_2 \cup (Y_1 \setminus X_2)$*, and*

$$\phi_1 \rightsquigarrow^* \phi_2 := \exists (Y_1 \cap X_2) : (\phi_1 \rightsquigarrow \phi_2) \tag{2}$$

Note that in the definition above, Y_1 and X_2 could also be disjoint, which means that no connections exist between I_1 and I_2. This can be used to model the *parallel composition* of I_1 and I_2, which then becomes a special case of serial composition.

Four serial composition examples are shown in Figure 3. In all of them, a component C with different guarantees is connected to a component D that is expecting its second input, x_2, to be different than zero.

In case (a), the contract of C guarantees that its output x_2 will be non-zero, and indeed, the composite interface contract is equal to true and thus well-formed.

In cases (b) and (c), where the contracts of C are $x_2 = 0$ and true, the resulting contract of the composition is false, i.e., $C \rightsquigarrow D$ is not well-formed in these two cases. In case (b) this is not surprising since we know that $x_2 = 0$ is an illegal input for D. In case (c), the contract of C is too weak, therefore it cannot be guaranteed that the input to D will be legal.

Case (d) presents a more interesting example. In this case the interface of C is $(\{z\}, \{x_2\}, x_2 \geq z)$. The requirement that $x_2 \neq 0$ induces a new requirement in the resulting contract, namely, that input z be strictly positive. This is the weakest requirement on z that allows to ensure $x_2 \neq 0$.

Definition 7 (Compatibility). *Let* $I_1 = (X_1, Y_1, \phi_1)$ *and* $I_2 = (X_2, Y_2, \phi_2)$ *be two composable interfaces. We say that* I_1 *and* I_2 *are* compatible *if* $I_1 \rightsquigarrow I_2$ *is well-formed.*

In Figure 3, in examples (a) and (d), the interfaces of C and D are compatible, whereas in examples (b) and (c) they are not.

We remark that we view compatibility as a key differentiating aspect of input-complete theories and interface theories. Compatibility is a local correctness property, akin to type checking. As the examples of Figure 3 illustrate, we can speak of compatibility between components without proving any property about the entire system. We view this as more lightweight than full system verification. In addition, example (d) illustrates how composition can be used to induce new input constraints, which is akin to type inference.

The difference between $\rightsquigarrow$ and standard composition, i.e., conjunction, lies in the last conjunct of Formula (1), namely $\forall Y_1 : \big(\phi_1 \rightarrow \mathsf{in}(\phi_2)\big)$. The latter is

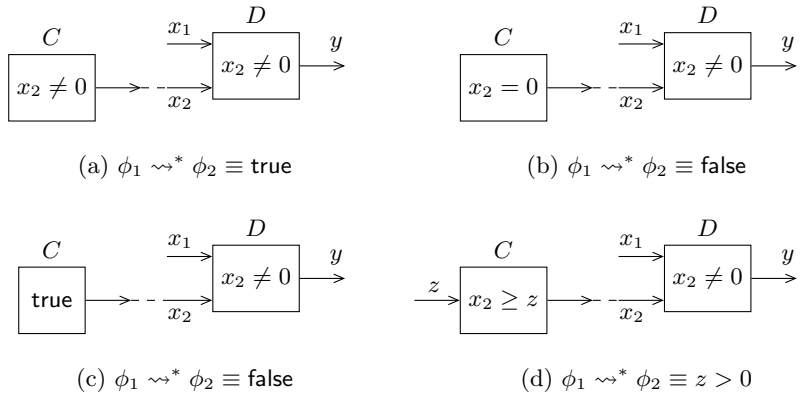


Fig. 3. Four examples of serial composition of relational interfaces.

a condition on the free inputs of the composite interface (because $\phi_1 \to \mathsf{in}(\phi_2)$ is a formula on $X_1 \cup Y_1 \cup X_2$). This conjunct states that, for a given input to the composite interface, any outputs that satisfy ϕ_1 will be legal inputs for ϕ_2. It can be easily seen that if ϕ_2 is input-complete, then this conjunct evaluates to true, so $\phi_1 \rightsquigarrow \phi_2$ becomes equivalent to $\phi_1 \wedge \phi_2$. The same holds when ϕ_1 is deterministic, so standard composition is a special case of $\rightsquigarrow$.

Theorem 1 (Special cases of composition [22]). *Let $I_1 = (X_1, Y_1, \phi_1)$ and $I_2 = (X_2, Y_2, \phi_2)$ be two composable interfaces. If I_2 is input-complete or I_1 is deterministic, then $\phi_1 \rightsquigarrow \phi_2 \equiv \phi_1 \wedge \phi_2$.*

2.3 Refinement

Definition 8 (Refinement). *We say that an interface $I' = (X', Y', \phi')$ refines an interface $I = (X, Y, \phi)$, written $I' \sqsubseteq I$, iff $X' \subseteq X$, $Y' \supseteq Y$, and the following formula is valid:*

$$\mathsf{in}(\phi) \to \big(\mathsf{in}(\phi') \wedge (\phi' \to \phi)\big) \tag{3}$$

The condition can be written as the conjunction of two conditions:

$$\mathsf{in}(\phi) \to \mathsf{in}(\phi') \tag{4}$$

$$(\mathsf{in}(\phi) \wedge \phi') \to \phi \tag{5}$$

The first condition guarantees that any input assignment that is legal in I will also be legal in I'. The second states that for every input assignment that is legal in I, all output assignments that can be possibly computed by I' from that input, can also be produced by I.

Theorem 2 (Refinement preserves well-formedness [22]). *Let I, I' be stateless interfaces such that $I' \sqsubseteq I$. If I is well-formed, then I' is well-formed.*

Theorem 3 (Composition preserves refinement [22]). *Let $I_1, I_2, I_1', $ and I_2' be interfaces such that $I_1' \sqsubseteq I_1$ and $I_2' \sqsubseteq I_2$. Then $I_1' \rightsquigarrow I_2' \sqsubseteq I_1 \rightsquigarrow I_2$.*

We can conclude from Theorems 2 and 3 that refinement preserves compatibility:

Corollary 1 (Refinement preserves compatibility [22]). *Let I_1, I_2 be compatible interfaces. Let I_1', and I_2' be interfaces such that $I_1' \sqsubseteq I_1$, $I_2' \sqsubseteq I_2$. Then I_1' and I_2' are also compatible.*

3 Error-Completion

Error-completion is a lossless transformation from (possibly non-input-complete) relational interfaces to input-complete relational interfaces. The idea is to capture illegal inputs using an extra boolean output variable. This has already been proposed in [22]. However, the way in which error-completion is defined in [22] is too strict, and does not allow us to reduce checking refinement of RIs to checking implication of their error-completed versions. We explain this further below.

In this paper we provide a less restrictive version of error-completion:

Definition 9 (Error-completion). *Let $I = (X, Y, \phi)$ be an interface. Let e be a new output variable, such that $e \notin X \cup Y$. The* error-completion *of ϕ is the formula $\mathsf{EC}(\phi)$ over $X \cup Y \cup \{e\}$, defined as follows:*

$$\mathsf{EC}(\phi) := \mathsf{in}(\phi) \rightarrow (\phi \wedge \neg e) \tag{6}$$

It is easy to verify that $\mathsf{EC}(\phi)$ is input-complete, for any ϕ. Also note that if ϕ is input complete, then $\mathsf{EC}(\phi) \equiv (\phi \wedge \neg e)$.

In the example of the division component in Figure 1 where $\phi \equiv (x_2 \neq 0 \wedge x_1 = y \cdot x_2)$, the error-completion of ϕ is:

$$\mathsf{EC}(\phi) \equiv x_2 \neq 0 \rightarrow (x_1 = y \cdot x_2 \wedge \neg e)$$

Definition 10 (Inverse transformation). *Let $I = (X, Y, \phi)$ be an interface and let $\phi_e = \mathsf{EC}(\phi)$ be the error-completion of ϕ. We can retrieve ϕ from ϕ_e using the following transformation:*

$$\mathsf{EC}^{-1}(\phi_e) := (\exists e : \phi_e) \wedge (\forall Y \cup \{e\} : \phi_e \rightarrow \neg e) \tag{7}$$

It can be shown that the two conjuncts of the definition of EC^{-1} correspond to $\phi \vee \neg\mathsf{in}(\phi)$ and $\mathsf{in}(\phi)$, respectively. Intuitively $\exists e : \phi_e$ adds all illegal inputs to the domain of ϕ and $\forall Y \cup \{e\} : \phi_e \rightarrow \neg e$ removes them. Formally, EC^{-1} is a left inverse of EC:

Lemma 1. *Any formula ϕ over $X \cup Y$ is equivalent to $\mathsf{EC}^{-1}(\mathsf{EC}(\phi))$, i.e.:*

$$\phi \equiv \mathsf{EC}^{-1}(\mathsf{EC}(\phi)) \tag{8}$$

Proof. If we expand the definitions of EC and EC^{-1}, $\mathsf{EC}(\mathsf{EC}^{-1}(\phi))$ is equal to:

$$\mathsf{EC}(\mathsf{EC}^{-1}(\phi)) \equiv (\exists e : (\mathsf{in}(\phi) \to (\phi \wedge \neg e))) \wedge (\forall Y \cup \{e\} : (\mathsf{in}(\phi) \to (\phi \wedge \neg e)) \to \neg e).$$

We examine the two conjuncts separately. The first conjunct is:

$$\begin{aligned} \exists e : (\mathsf{in}(\phi) \to (\phi \wedge \neg e)) &= \mathsf{in}(\phi) \to (\phi \wedge \exists e : \neg e) \\ &= \mathsf{in}(\phi) \to \phi. \end{aligned}$$

Let Φ be the formula $(\mathsf{in}(\phi) \to (\phi \wedge \neg e)) \to \neg e$.
For $e = \mathsf{true}$, Φ is equal to $(\mathsf{in}(\phi) \to \mathsf{false}) \to \mathsf{false} \equiv \mathsf{in}(\phi)$.
For $e = \mathsf{false}$, Φ is equal to $(\mathsf{in}(\phi) \to \phi) \to \mathsf{true} \equiv \mathsf{true}$.
Therefore the second conjunct is equivalent to $\forall Y : \mathsf{in}(\phi)$ or $\mathsf{in}(\phi)$ since the latter does not depend on Y variables.
Going back to $\mathsf{EC}^{-1}(\mathsf{EC}(\phi))$, we get:

$$\mathsf{EC}^{-1}(\mathsf{EC}(\phi)) \equiv (\mathsf{in}(\phi) \to \phi) \wedge \mathsf{in}(\phi) \equiv \phi \wedge \mathsf{in}(\phi) \equiv \phi.$$

□

It can be easily shown that any function that has a left inverse is injective. Therefore, $\phi_1 \not\equiv \phi_2$ implies $\mathsf{EC}(\phi_1) \not\equiv \mathsf{EC}(\phi_2)$.

Lemma 1 is an important result, as it proves that the transformations $\mathsf{EC}, \mathsf{EC}^{-1}$ are *lossless*. In addition, as we shall show next, Lemma 1 allows to prove that EC forms a bijection between relational interfaces and an appropriate subclass of error-complete interfaces.

3.1 Meaningful ECI

We have seen that EC^{-1} is a left inverse of EC. Note, however, that it is not the case that EC^{-1} is a right inverse of EC, that is, $\mathsf{EC}(\mathsf{EC}^{-1}(\phi_e))$ is *not* always equivalent to ϕ_e. For example, if $\phi_e := (y = e)$ where y is an output, then:

$$\mathsf{EC}^{-1}(\phi_e) \equiv (\exists e : y = e) \wedge (\forall Y \cup \{e\} : (y = e) \to \neg e) \equiv \mathsf{true} \wedge \mathsf{false} \equiv \mathsf{false}$$

while

$$\mathsf{EC}(\mathsf{EC}^{-1}(\phi_e)) \equiv \mathsf{EC}(\mathsf{false}) \equiv \mathsf{in}(\mathsf{false}) \to (\mathsf{false} \wedge \neg e) \equiv \mathsf{false} \to \mathsf{false} \equiv \mathsf{true}.$$

The same can be shown for less elementary contracts. For instance, if $\phi_e := (x = 0 \to \neg e) \wedge (x = 1 \to e)$, then $\mathsf{EC}^{-1}(\phi_e) = (x = 0)$ but $\mathsf{EC}(x = 0) \neq \phi_e$.
In fact we can prove that ϕ_e is generally stronger than $\mathsf{EC}(\mathsf{EC}^{-1}(\phi_e))$.

Lemma 2. *Any formula ϕ_e over $X \cup Y \cup \{e\}$ is equivalent or stronger than $\mathsf{EC}(\mathsf{EC}^{-1}(\phi_e))$:*

$$\phi_e \to \mathsf{EC}(\mathsf{EC}^{-1}(\phi_e)) \tag{9}$$

As the above results show, even though, by Lemma 1, EC is injective, it is not surjective. This means that there are error-complete interfaces which do not correspond to any meaningful relational interfaces. However, note that it was never our intention to handle arbitrary error-complete interfaces, that is, arbitrary formulas over $X \cup Y \cup \{e\}$. Instead, what we are interested in is a transformation from contracts over $X \cup Y$ to input-complete contracts over $X \cup Y \cup \{e\}$. We are thus only interested in the subclass of error-complete interfaces which are obtained from relational interfaces via EC. That is, we are only interested in the image of EC. We call this subclass the class of *meaningful error-complete interfaces* (MECI). MECI is a strict subset of the set of all input-complete interfaces over $X \cup Y \cup \{e\}$, which we denote by ECI.

Definition 11 (Meaningful error-complete interfaces). *Let ϕ_e be a formula over $X \cup Y \cup \{e\}$. ϕ_e is said to be* meaningful *iff there exists a formula ϕ over $X \cup Y$ such that $\mathsf{EC}(\phi) \equiv \phi_e$.*

Theorem 4. *Let ϕ_e be a formula over $X \cup Y \cup \{e\}$. ϕ_e is meaningful iff $\mathsf{EC}(\mathsf{EC}^{-1}(\phi_e)) \equiv \phi_e$.*

Proof. Suppose $\mathsf{EC}(\mathsf{EC}^{-1}(\phi_e)) \equiv \phi_e$. By definition of EC^{-1}, $\mathsf{EC}^{-1}(\phi_e)$ is a formula over $X \cup Y$. Therefore, setting $\phi := \mathsf{EC}^{-1}(\phi_e)$, we have $\mathsf{EC}(\phi) \equiv \phi_e$, thus ϕ_e is meaningful.

In the other direction, suppose ϕ_e is meaningful. Then there is a formula ϕ over $X \cup Y$ such that $\mathsf{EC}(\phi) \equiv \phi_e$. $\mathsf{EC}(\phi) \equiv \phi_e$ implies $\mathsf{EC}^{-1}(\mathsf{EC}(\phi)) \equiv \mathsf{EC}^{-1}(\phi_e)$. By Lemma 1, $\mathsf{EC}^{-1}(\mathsf{EC}(\phi)) \equiv \phi$, therefore, $\phi \equiv \mathsf{EC}^{-1}(\phi_e)$. This implies $\mathsf{EC}(\phi) \equiv \mathsf{EC}(\mathsf{EC}^{-1}(\phi_e))$. Since $\mathsf{EC}(\phi) \equiv \phi_e$, we get $\phi_e \equiv \mathsf{EC}(\mathsf{EC}^{-1}(\phi_e))$. □

Theorem 4 is an important result which shows that EC, restricted to the class MECI, is a bijection. This is illustrated in Figure 4. Note that we interpret the spaces RI, ECI, MECI, and so on, as containing *semantic* rather than syntactic objects, that is, relations rather than formulas. Alternatively, and equivalently, a point in each of these spaces represents the equivalence class of all equivalent formulas.

3.2 Composition in the ECI domain

Beyond merely having a lossless transformation from relational interfaces to error-complete interfaces and back, we are interested in using the error-completion to perform operations on relational interfaces more efficiently. In this section we show how error-completion can be used to compute serial composition of relational interfaces by avoiding the universal quantification formula $\forall Y_1 : \big(\phi_1 \rightarrow \mathsf{in}(\phi_2)\big)$ used in the definition of $\rightsquigarrow$. The idea is to *delay* computing the game-theoretic demonic composition as much as possible. In that sense, we can perform serial composition on the error-complete domain, and use the inverse transformation EC^{-1} (which introduces the universal quantification) to return to the non-input-complete domain whenever necessary. To achieve this, we define a serial composition operator $\overset{e}{\rightsquigarrow}$ on error-completions:

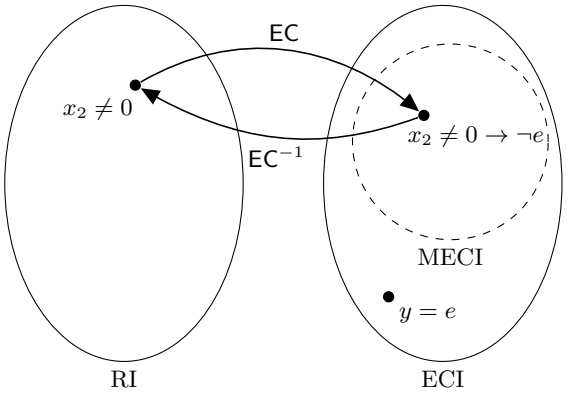


Fig. 4. Meaningful error-complete interfaces

Definition 12. *Let $I_1 = (X_1, Y_1, \phi_1)$, $I_2 = (X_2, Y_2, \phi_2)$ be two composable interfaces. Let e_1, e_2 be two fresh variables, i.e., $e_1, e_2 \notin X_1 \cup Y_1 \cup X_2 \cup Y_2$. Let ψ_1, ψ_2 be two predicates over $X_1 \cup Y_1 \cup \{e_1\}$ and $X_2 \cup Y_2 \cup \{e_2\}$ respectively, such that $\psi_1 = \mathsf{EC}(\phi_1)[e/e_1]$ and $\psi_2 = \mathsf{EC}(\phi_2)[e/e_2]$ where $\xi[e/e_i]$ denotes the formula ξ' obtained by ξ by replacing e with e_i. We define the composition of ψ_1 and ψ_2 as:*

$$\psi_1 \overset{e}{\rightsquigarrow} \psi_2 := \exists e_1, e_2 : \big(\psi_1 \wedge \psi_2 \wedge e = (e_1 \vee e_2)\big) \tag{10}$$

The operator $\overset{e}{\rightsquigarrow}$ is illustrated in Figure 5 for the simple case of single-input/single-output components.

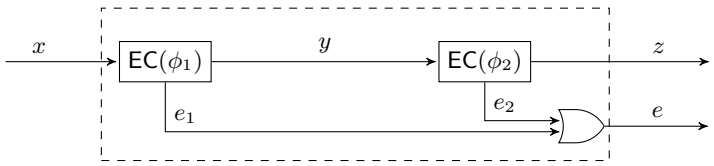


Fig. 5. Illustration of serial composition in the error-complete domain.

We can now state a main result which allows to compute serial composition losslessly on the error-complete domain:

Theorem 5. *Let $I_1 = (X_1, Y_1, \phi_1)$, $I_2 = (X_2, Y_2, \phi_2)$ be two composable interfaces. Then the following is true:*

$$\mathsf{EC}^{-1}\big(\mathsf{EC}(\phi_1) \overset{e}{\rightsquigarrow} \mathsf{EC}(\phi_2)\big) \equiv \phi_1 \rightsquigarrow \phi_2 \tag{11}$$

Examples Let us look at the composition examples of Figure 3 and see how serial composition is performed in the error-complete domain. We first look at example (c) and compute the error-completion of the contracts of the two components:

$$\psi_1 := \mathsf{EC}(\mathsf{true})[e/e_1] \equiv \mathsf{true} \to (\mathsf{true} \wedge \neg e_1) = \neg e_1$$

$$\begin{aligned}\psi_2 := \mathsf{EC}(x_2 \neq 0)[e/e_2] &\equiv \mathsf{in}(x_2 \neq 0) \to (x_2 \neq 0 \wedge \neg e_2)\\ &\equiv x_2 \neq 0 \to (x_2 \neq 0 \wedge \neg e_2)\\ &\equiv x_2 \neq 0 \to \neg e_2\end{aligned}$$

The serial composition contract in the error-complete domain is:

$$\begin{aligned}\psi_1 \overset{e}{\rightsquigarrow} \psi_2 &\equiv \exists e_1, e_2 : \mathsf{true} \wedge (x_2 \neq 0 \to \neg e_2) \wedge (e = e_1 \vee e_2)\\ &\equiv \exists e_1, e_2 : (x_2 \neq 0 \to \neg e_2) \wedge (e = e_1 \vee e_2)\\ &\equiv \mathsf{true}\end{aligned}$$

If we apply the EC^{-1} operator, we indeed get back the serial composition contract we had computed before:

$$\begin{aligned}\mathsf{EC}^{-1}(\mathsf{true}) &\equiv (\exists e : \mathsf{true}) \wedge (\forall x_2 \forall e : \mathsf{true} \to \neg e)\\ &\equiv (\forall x_2 \forall e : \neg e)\\ &\equiv \mathsf{false}\end{aligned}$$

In case (d) of Figure 3, the error-completion of the contract of the division component, ψ_2, is the same as before, and for component C we get:

$$\begin{aligned}\psi_1 := \mathsf{EC}(x_2 \geq z)[e/e_1] &\equiv \mathsf{in}(x_2 \geq z) \to (x_2 \geq z \wedge \neg e_1)\\ &\equiv (\exists x_2 : x_2 \geq z) \to (x_2 \geq z \wedge \neg e_1)\\ &\equiv x_2 \geq z \wedge \neg e_1\end{aligned}$$

The serial composition contract in the error-complete domain is:

$$\begin{aligned}\psi := \psi_1 \overset{e}{\rightsquigarrow} \psi_2 &\equiv \exists e_1, e_2 : (x_2 \geq z \wedge \neg e_1) \wedge (x_2 \neq 0 \to \neg e_2) \wedge (e = e_1 \vee e_2)\\ &\equiv (x_2 \geq z \wedge \neg e) \vee (x_2 \geq z \wedge x_2 = 0 \wedge e)\\ &\equiv x_2 \geq z \wedge (\neg e \vee (x_2 = 0 \wedge e))\end{aligned}$$

We examine the two conjuncts of EC^{-1} separately:

$$\begin{aligned}\exists e : \psi &\equiv \exists e : (x_2 \geq z \wedge (\neg e \vee (x_2 = 0 \wedge e)))\\ &\equiv x_2 \geq z \wedge \exists e : (\neg e \vee (x_2 = 0 \wedge e)) \equiv x_2 \geq z\end{aligned}$$

$$\begin{aligned}\forall x_2 \forall e : \psi \to \neg e &\equiv \forall x_2 \forall e : (x_2 \geq z \wedge (\neg e \vee (x_2 = 0 \wedge e))) \to \neg e\\ &\equiv \forall x_2 : (x_2 \geq z \wedge (\mathsf{false} \vee (x_2 = 0 \wedge \mathsf{true}))) \to \mathsf{false}\\ &\equiv \forall x_2 : \neg(x_2 \geq z \wedge x_2 = 0)\\ &\equiv \neg\exists x_2 : (x_2 \geq z \wedge x_2 = 0)\\ &\equiv z > 0\end{aligned}$$

Thus $\mathsf{EC}^{-1}(\psi_1 \overset{e}{\rightsquigarrow} \psi_2) \equiv x_2 \geq z \wedge z > 0$, and if we hide the connected input x_2 we get $\exists x_2 : (x_2 \geq z \wedge z > 0) \equiv z > 0$ which is equal to $\phi_1 \rightsquigarrow^* \phi_2$.

3.3 Refinement in the ECI domain

In the previous section we showed how to perform composition on the error-complete domain. In this section we show that checking refinement of relational interfaces can be reduced to checking standard implication on their error-completions.

Theorem 6. *Let* $I_1 = (X_1, Y_1, \phi_1)$, $I_2 = (X_2, Y_2, \phi_2)$ *be two interfaces such that* $X_1 \subseteq X_2$ *and* $Y_1 \supseteq Y_2$. *Then* $I_1 \sqsubseteq I_2$ *iff* $\mathsf{EC}(\phi_1) \rightarrow \mathsf{EC}(\phi_2)$ *is valid.*

Proof. (only if) We repeat Formula (3) for convenience:

$$\mathsf{in}(\phi_2) \rightarrow \big(\mathsf{in}(\phi_1) \wedge (\phi_1 \rightarrow \phi_2)\big)$$

We need to show that if Formula (3) is valid then

$$\big(\mathsf{in}(\phi_1) \rightarrow (\phi_1 \wedge \neg e)\big) \rightarrow \big(\mathsf{in}(\phi_2) \rightarrow (\phi_2 \wedge \neg e)\big) \tag{12}$$

is also valid. To show that Formula (12) is valid, consider a valuation a that satisfies $\big(\mathsf{in}(\phi_1) \rightarrow (\phi_1 \wedge \neg e)\big) \wedge \mathsf{in}(\phi_2)$. We need to show that a also satisfies $\phi_2 \wedge \neg e$. Because a satisfies $\mathsf{in}(\phi_2)$ and Formula (3) is valid, a also satisfies $\mathsf{in}(\phi_1) \wedge (\phi_1 \rightarrow \phi_2)$. Because a satisfies $\mathsf{in}(\phi_1)$ and also $\big(\mathsf{in}(\phi_1) \rightarrow (\phi_1 \wedge \neg e)\big)$, it also satisfies $\phi_1 \wedge \neg e$. And because it satisfies ϕ_1 and $\phi_1 \rightarrow \phi_2$ it also satisfies ϕ_2. Thus, it satisfies $\phi_2 \wedge \neg e$.

(if) We need to show that if Formula (12) is valid then Formula (3) is also valid. It suffices to show that if the negation of Formula (3) is satisfiable then the negation of Formula (12) is also satisfiable.

The negation of Formula (3) is

$$\mathsf{in}(\phi_2) \wedge \big(\neg\mathsf{in}(\phi_1) \vee (\phi_1 \wedge \neg\phi_2)\big) \tag{13}$$

The negation of Formula (12) is

$$\big(\mathsf{in}(\phi_1) \rightarrow (\phi_1 \wedge \neg e)\big) \wedge \mathsf{in}(\phi_2) \wedge (\neg\phi_2 \vee e) \tag{14}$$

Suppose a satisfies Formula (13). Notice that a is an assignment over variables in $X_2 \cup Y_1$. In particular, a does not assign a value to e. There are two cases:

1. a satisfies $\mathsf{in}(\phi_2) \wedge \neg\mathsf{in}(\phi_1)$: We extend assignment a to assignment a' over $X_2 \cup Y_1 \cup \{e\}$, such that a' sets e to true and keeps the values of a for all other variables. Clearly, a' satisfies the last conjunct $\neg\phi_2 \vee e$ of Formula (14). Also, because a satisfies the first two conjuncts of Formula (14) and because these conjuncts do not refer to e, a' satisfies them as well. Therefore, a' satisfies Formula (14).
2. a satisfies $\mathsf{in}(\phi_2) \wedge \phi_1 \wedge \neg\phi_2$: As before we extend a to a' but now a' sets e to false. It can be seen that this a' satisfies the consequent of the first conjunct and the last two conjuncts of Formula (14) and thus satisfies Formula (14).

Thus, in both cases Formula (14) is satisfiable. □

We can now fulfill our promise to explain why the error-completion transformation defined in [22] is not satisfactory. The definition given in [22] is:

$$\mathsf{EC}_{\mathtt{strict}}(\phi) := (\neg \mathsf{in}(\phi) \wedge e) \vee (\phi \wedge \neg e) \tag{15}$$

Unfortunately, Theorem 6 does not hold if we replace EC by $\mathsf{EC}_{\mathtt{strict}}$. Intuitively, this is because $\mathsf{EC}_{\mathtt{strict}}$ is too strict. It requires that the error variable is true when an input is given that does not satisfy $\mathsf{in}(\phi)$. This demand goes against the contravariant definition of refinement: a refinement of ϕ can accept more inputs than ϕ. To give a concrete example, consider two interfaces I, I' with contracts $\phi \equiv x > 0$ and $\phi' = \mathsf{true}$ respectively. true accepts more inputs than $x > 0$, therefore we have $I' \sqsubseteq I$. Indeed if we consider the error-completions $\psi := \mathsf{EC}(\phi)$ and $\psi' := \mathsf{EC}(\phi')$ we get:

$$\psi \equiv x > 0 \rightarrow \neg e \quad \text{and} \quad \psi' \equiv \neg e$$

and it is true that $\psi' \rightarrow \psi$.

However, if we consider the strict error-completions we get:

$$\psi_{\mathtt{strict}} := \mathsf{EC}_{\mathtt{strict}}(\phi) \equiv (x > 0 \wedge \neg e) \vee (x \leq 0 \wedge e)$$
$$\psi'_{\mathtt{strict}} := \mathsf{EC}_{\mathtt{strict}}(\phi') \equiv \neg e$$

and it is not the case that $\psi'_{\mathtt{strict}} \rightarrow \psi_{\mathtt{strict}}$.

4 Discussion

4.1 Extension to stateful

We first discuss how the ideas of error-complete interfaces can be applied in the case of stateful interfaces. For the sake of brevity, we do this by means of an example. Nonetheless, we are confident that the results in the paper extend without problem to the general case of stateful interfaces.

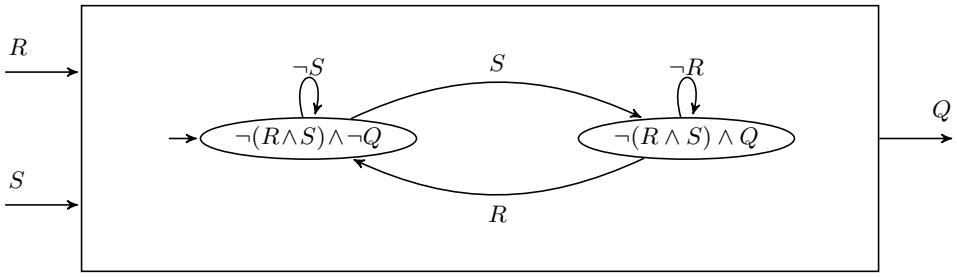


Fig. 6. Stateful interface example

Stateful interfaces can be represented as extended automata whose states are annotated with contracts. Figure 6 shows the stateful interface of an SR flip-flop. An SR flip-flop has two inputs, S for set, R for reset, and an output Q which is equal to the current flip-flop state. When neither S nor R are present, the flip-flop maintains its state. When S is present and R is not, the output Q is set to true. When R is present and S is not, the output Q is set to false. The combination of both S and R being present is illegal; in that case, in real implementations, the output depends on gate propagation delays, and hence it is considered an error state. In the game interpretation of interfaces, after the environment decides on the values of R and S, the interface will move to the correct state and produce an output Q that satisfies the contract of that state.

Performing error-completion on stateful interfaces is straightforward: it amounts to perform error-completion on the contract of every state. In the case of the SR flip-flop interface shown above, it amounts to adding a new boolean output variable e and modifying its two contracts as follows:

$$\begin{aligned}\mathsf{EC}(\neg(R \wedge S) \wedge \neg Q) &\equiv \mathsf{in}(\neg(R \wedge S) \wedge \neg Q) \rightarrow (\neg(R \wedge S) \wedge \neg Q \wedge \neg e) \\ &\equiv \neg(R \wedge S) \rightarrow (\neg(R \wedge S) \wedge \neg Q \wedge \neg e) \\ &\equiv \neg(R \wedge S) \rightarrow (\neg Q \wedge \neg e)\end{aligned}$$

$$\mathsf{EC}(\neg(R \wedge S) \wedge Q) \equiv \neg(R \wedge S) \rightarrow (Q \wedge \neg e)$$

4.2 Value-completion

An alternative way to achieve a notion of error-completion is to introduce error *values* in the domains of the original output variables, without adding new error variables. Let us discuss this alternative.

Let ϕ be a formula over input and output variables $X \cup Y$. In this subsection, we assume that Y is non-empty. For each variable v, let D_v denote the domain of v, i.e., the set of all possible values that v can take. Let $\perp$ be a new value, not in D_v, for any v. Let $D_v^{\perp} := D_v \cup \{\perp\}$. The *value-completion* of ϕ is a formula $\mathsf{VC}(\phi)$ over $X \cup Y$, where every output variable y is assumed to range over domain $D_y^{\perp}$. $\mathsf{VC}(\phi)$ is defined as follows:

$$\mathsf{VC}(\phi) := \mathsf{in}(\phi) \rightarrow (\phi \wedge \bigwedge_{y \in Y} y \neq \perp) \tag{16}$$

As with $\mathsf{EC}(\phi)$, it is easy to verify that $\mathsf{VC}(\phi)$ is input-complete, for any ϕ.

For example, the contract $\phi \equiv (x_2 \neq 0 \wedge x_2 \cdot y = x_1)$ has the following value-completion:

$$\mathsf{VC}(\phi) \equiv x_2 \neq 0 \rightarrow (x_2 \cdot y = x_1 \wedge y \neq \perp)$$

Consider a value-complete formula ϕ_b. Interpreting ϕ_b as a relation, we can define the inverse operation VC^{-1} which yields a formula over $X \cup Y$, where each output variable y ranges over D_y. $\mathsf{VC}^{-1}(\phi_b)$ will contain all valuations

(a_X, a_Y) over $\mathcal{A}(X \cup Y)$, such that $(a_X, a_Y) \in \phi_b$ and if there exists a'_Y such that $(a_X, a'_Y) \in \phi_b$, then for all $y \in Y$, $a'_Y(y) \neq \bot$. Formally:

$$\mathsf{VC}^{-1}(\phi_b) := \{(a_X, a_Y) \in \mathcal{A}(X \cup Y) \mid (a_X, a_Y) \in \phi_b \wedge \forall a'_Y : (a_X, a'_Y) \in \phi_b \rightarrow \forall y \in Y : a'_Y(y) \neq \bot \}$$

We can show that VC^{-1} is a left inverse of VC:

$$\mathsf{VC}^{-1}(\mathsf{VC}(\phi)) \equiv \phi$$

In a similar way that we used to define MECI, we can now define a space of meaningful value-complete interfaces, or MVCI, which is a subclass of VCI, the set of all value-complete interfaces. Doing so, we obtain a bijection, VC, between RI and MVCI. We also have the bijection EC between RI and MECI. Therefore, there exists a bijection between MECI and MVCI, which means that MECI and MVCI are isomorphic. This is illustrated in Figure 7, which expands our previous Figure 4.

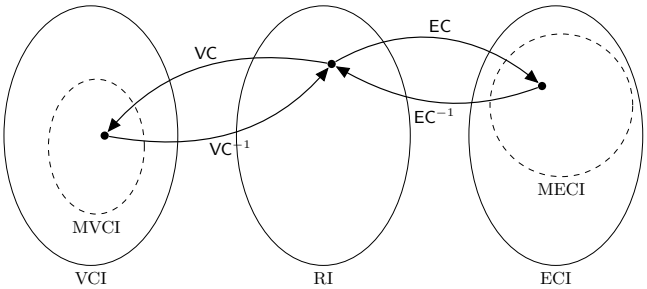


Fig. 7. Meaningful value-complete and error-complete interfaces

5 Related Work

Component-based design is one of the holy grails of computer science, and as a result, a large number of compositional specification and design frameworks exist in the literature. See, for instance, [21,1,15,17,4], and the related work discussion in [22]. As mentioned in the introduction, our work follows the approach of interface theories [9,10], where specifications can be non-input-complete. This is in contrast with other compositional theories such as I/O automata [16], FOCUS [5,6], and reactive modules [2], where specifications are assumed to be input-complete.

Input-completion of finite automata is a folklore technique. Given a finite automaton with a partial transition function, input-completion consists in adding one extra, non-accepting, state to the automaton, and directing all missing transitions to that state. This results in an equivalent automaton which accepts the

same language as the original one. Moreover, the resulting automaton has a total transition function, thus can be seen as being "input-complete". Input-completion can be adapted to interface automata [10] in a straightforward way: add an error state, direct all missing inputs to that state, and add a self-loop for any possible (input or output, and assuming no internal) action to the error state. This transformation appears to correctly reduce the alternating refinement relation between interface automata to a standard simulation relation, however, we were unable to find a reference in the literature to corroborate this.

In the context of viewing programs as predicates or relations [19,14], the question arises whether these relations should be total or partial. This question naturally arises in sequential programs that contain "while" loops, and where modeling program (non-)termination is a concern. The question has received a lot of attention in the literature (see [18] for a survey) and has also generated some controversy [13]. In this paper we accept as a fact that partial relations (i.e., non-input-complete interfaces) are useful, so non-input-completeness is our starting point. An extensive argument on the usefulness of non-input-completeness can be found in [22], which also introduces the framework of relational interfaces used in this paper.

Finally, as already mentioned in the introduction, the error-completion operator introduced in [22] is different from the one defined in this paper, which we believe is the right one.

6 Conclusions and Future Work

We presented a set of transformations $\mathsf{EC}, \mathsf{EC}^{-1}$, which allow to transform non-input-complete relational interfaces into input-complete ones, so that composition and refinement can be computed using standard methods on the error-complete domain. We emphasize that we do not propose error-complete interfaces as a new interface theory. We merely suggest them as convenient equivalent representations of non-input-complete relational interfaces, which can make computation of composition and refinement easier.

Regarding future work, a number of algorithmic issues need to be resolved to make the relational interface theory (with or without error-completion) practical, including effective procedures for formula simplification and quantifier elimination. Another interesting question is raised in [22]: can feedback composition be defined for general interfaces, rather than for a subclass of stateful interfaces? This question remains open. Another issue is how to extend the relational interface theory to liveness properties. Finally, value-completion also deserves a more thorough study. In particular, it is not entirely obvious how to perform value-completion on sink components, that is, those with no outputs.

References

1. J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

2. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
3. R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *CONCUR'98*, volume 1466 of *LNCS*. Springer, 1998.
4. R-J. Back and J. Wright. *Refinement Calculus*. Springer, 1998.
5. M. Broy. Compositional refinement of interactive systems. *J. ACM*, 44(6):850–891, 1997.
6. M. Broy and K. Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer, 2001.
7. A. Chakrabarti, L. de Alfaro, T. Henzinger, and F. Mang. Synchronous and bidirectional component interfaces. In *CAV*, LNCS 2404, pages 414–427. Springer, 2002.
8. L. de Alfaro. Game models for open systems. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 192–213. Springer, 2004.
9. L. de Alfaro and T. Henzinger. Interface automata. In *Foundations of Software Engineering (FSE)*. ACM Press, 2001.
10. L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *EMSOFT'01*. Springer, LNCS 2211, 2001.
11. L. Doyen, T. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *8th ACM & IEEE International conference on Embedded software, EMSOFT*, pages 79–88, 2008.
12. M. Geilen, S. Tripakis, and M. Wiggers. The Earlier the Better: A Theory of Timed Actor Interfaces. In *14th Intl. Conf. Hybrid Systems: Computation and Control (HSCC'11)*. ACM, 2011.
13. E.C.R. Hehner and D.L. Parnas. Technical correspondence. *Commun. ACM*, 28(5):534–538, 1985.
14. C. A. R. Hoare. Programs are predicates. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 141–155, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.
15. B. Liskov. Modular program construction using abstractions. In *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 354–389. Springer, 1979.
16. N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.
17. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
18. G. Nelson. A generalization of dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4):517–561, 1989.
19. D.L. Parnas. A generalized control structure and its formal definition. *Commun. ACM*, 26(8):572–581, 1983.
20. J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone. A modal interface theory for component-based design. *Fundam. Inf.*, 108(1-2):119–149, January 2011.
21. J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
22. S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. A theory of synchronous relational interfaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(4), July 2011.