Preprint accepted in ACM TOPLAS (Transactions on Programming Languages and Systems), March 1, 2011.

A Theory of Synchronous Relational Interfaces

Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, Edward A. Lee

Compositional theories are crucial when designing large and complex systems from smaller components. In this work we propose such a theory for synchronous concurrent systems. Our approach follows so-called interface theories, which use game-theoretic interpretations of composition and refinement. These are appropriate for systems with distinct inputs and outputs, and explicit conditions on inputs that must be enforced during composition. Our interfaces model systems that execute in an infinite sequence of synchronous rounds. At each round, a contract must be satisfied. The contract is simply a relation specifying the set of valid input/output pairs. Interfaces can be composed by parallel, serial or feedback composition. A refinement relation between interfaces is defined, and shown to have two main properties: (1) it is preserved by composition, and (2) it is equivalent to substitutability, namely, the ability to replace an interface by another one in any context. Shared refinement and abstraction operators, corresponding to greatest lower and least upper bounds with respect to refinement, are also defined. Input-complete interfaces, that impose no restrictions on inputs, and deterministic interfaces, that produce a unique output for any legal input, are discussed as special cases, and an interesting duality between the two classes is exposed. A number of illustrative examples are provided, as well as algorithms to compute compositions, check refinement, and so on, for finite-state interfaces.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms: Algorithms, Design, Languages, Theory, Verification

Additional Key Words and Phrases: Compositionality, Interfaces, Refinement, Substitutability

1. INTRODUCTION

Compositional methods, which allow one to assemble smaller components into larger systems both efficiently and correctly, are not simply a desirable feature

This report is a revised version of [Tripakis et al. 2009a; 2009b]. This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #CCR-0225610 (ITR), #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota. This work was also supported by the COMBEST and ArtistDesign projects of the European Union, and the Swiss National Science Foundation.
Tripakis, Blickly and Lee are with the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. Henzinger is with the Institute of Science and Technology Austria. Authors' emails: {stavros,blickly,eal}@eecs.berkeley.edu, tah@ist.ac.at. Corresponding author's address: Stavros Tripakis, 545Q, DOP Center, Cory Hall, EECS Department, University of California, Berkeley, CA 94720-1772, USA.

in system design: they are a must for designing large and complex systems. A compositional theory provides means for reasoning formally about components and their compositions. It also typically provides sufficient and/or necessary conditions for *substitutability*: when can a certain component be replaced by another one without compromising the correctness of the overall system? This property is clearly extremely important, in particular for incremental design.

The goal of this work is to develop a compositional theory for *synchronous concurrent systems*, systems where a set of components execute in an infinite sequence of *global rounds*. This is a fundamental model of computation with traditionally strong application in the hardware domain (digital circuits). Today the synchronous paradigm is also becoming widespread in software, in particular, in the domains of *embedded* and *cyber-physical systems* [Henzinger and Sifakis 2007; Lee 2008]. Tools such as Simulink from The MathWorks[1], SCADE from Esterel Technologies[2], or Ptolemy from Berkeley[3], and languages such as the synchronous languages [Benveniste et al. 2003] are important players in this field [Miller et al. 2010]. The semantics used in these models are synchronous.

Our work is situated in the context of *interface theories* [de Alfaro and Henzinger 2001a; 2001b]. An interface can be seen as an abstraction of a component: on one hand, it captures information that is essential in order to use the component in a given context; on the other hand, it hides unnecessary information, making reasoning simpler and more efficient. Interface theories typically define a set of composition operators and a refinement relation on interfaces, and provide theorems of preservation of correctness by refinement and preservation of refinement by composition, from which substitutability guarantees can be derived. These concepts are common to most compositional theories. What distinguishes interface theories is a *game-theoretic* interpretation of the basic concepts, namely, composition and refinement. The need for a game-theoretic interpretation has been argued extensively in previous works on interface theories [de Alfaro and Henzinger 2001a; 2001b; de Alfaro 2004]. In order to make this paper more self-contained, we also discuss the motivations behind this choice here, in Section 2.

The type of information about a component that is exposed in an interface varies depending on the application. For instance, in standard programming languages such as C or Java, the signature of a given method can be seen as an interface for that method. This interface provides sufficient information for type checking, but usually does not provide enough information for more detailed analysis, e.g., checking that a method computing division never attempts a division by zero. As this simple example illustrates, we should not expect a single "fits-all" interface theory, but multiple theories that are more or less suitable for different purposes. Suitability metrics could include expressiveness and ease of modeling in particular application domains, as well as tractability of the computational problems involved.

In our theory, an interface consists of a set of input variables X, a set of output variables Y, and a set of *contracts*. Semantically, a contract is simply a set of assignments of values to variables in $X \cup Y$. Syntactically, we use a logical formalism

[1]See `http://www.mathworks.com/products/simulink/`.
[2]See `http://www.esterel-technologies.com/products/scade-suite/`.
[3]See `http://ptolemy.eecs.berkeley.edu/`.

such as first-order logic to represent and manipulate contracts. For example, the predicate $x_2 \neq 0 \wedge y = \frac{x_1}{x_2}$ can be used to represent the contract of a component that performs division, with input variables x_1 and x_2 and output variable y. The contract here is the set of all assignments to variables x_1, x_2 and y that satisfy the predicate. The assignment $(x_1 := 6, x_2 := 2, y := 3)$ satisfies the contract, while any assignment where $x_2 := 0$ violates the contract. A more abstract contract for the same component, which only gives some information about the sign of the output based on the sign of the inputs, is $x_2 \neq 0 \wedge \big(y < 0 \equiv (x_1 < 0 < x_2 \vee x_2 < 0 < x_1)\big)$. An even more abstract contract is $x_2 \neq 0$. The latter guarantees nothing about the output, however, it still enforces that requirement that when performing division the denominator should be non-zero. We should note that these contracts implicitly use the fact that variables are numbers, symbols like $=$ for equality, and arithmetic operations such as division. Our theory does not depend on these, and works with variables of any domain, without assuming any properties on such domains. In practice, however, we often use such properties implicitly for convenience.

Contracts govern the operation of a component, which evolves in a sequence of synchronous rounds. Within a round, values are assigned to the input variables of the component by its environment, and the component assigns values to its output variables. Together the two assignments form a complete assignment over all variables. This assignment must satisfy the contract. A new assignment is found at each round. Interfaces can be *stateless* or *stateful*. In the stateless case, there is a single contract that must hold at every round (the assignments may still differ). In the general, stateful case, a different contract may be specified for each round. The contract in this case depends on the history of assignments observed so far, which we call a *state*. The set of states, as well as the set of contracts, can be infinite. When the set of contracts is finite, we have a *finite-state interface* (note that the domains of variables could still be infinite). Finite-state interfaces are represented as finite automata whose locations are labeled by contracts.

Interfaces can be composed by *connection* or by *feedback* (see Section 6). Connection essentially corresponds to serial (cascade) composition, however, it can also capture parallel composition as a special case (empty connection). Composition by connection is generally not the same as composition of relations. Section 2 discusses this choice extensively. Feedback is allowed only for *Moore interfaces*, where the contract does not depend on the current values of the input variables that are back-fed (although it may depend on past values of such variables). A *hiding* operator (Section 7) can be used to eliminate redundant or intermediate output variables. Hiding is always possible for stateless interfaces and corresponds to existentially quantifying variables in the contract. The situation is more subtle in the stateful case, where we need to ensure that the "hidden" variables do not influence the evolution of the contract from one state to the next. This is necessary to ensure preservation of refinement by hiding.

Our theory includes explicit notions of *environments*, *pluggability* and *substitutability* (Section 8). An environment E for an interface I is simply an interface whose input and output variables "mirror" those of I. I is pluggable to E (and vice versa) iff the closed-loop system formed by connecting the two is *well-formed*, that is, never reaches a state with an unsatisfiable contract. Substitutability means that an interface I' can replace another interface I in any environment. That is, for any environment E, if I is pluggable to E then I' is also pluggable to E.

Our refinement relation is similar in spirit to existing relations, such as function subtyping in type theory [Pierce 2002], behavioral subtyping [Liskov and Wing 1994], conformation in trace theory [Dill 1987], and alternating refinement [Alur et al. 1998]. All these, roughly speaking, state that I' refines I if I' accepts more inputs and produces fewer outputs than I. This requirement is easy to formalize as $\mathsf{in} \rightarrow \mathsf{in}' \wedge \mathsf{out}' \rightarrow \mathsf{out}$ when the input assumptions, in, are separated from the output guarantees, out. When the constraints on the inputs and outputs are mixed in the same contract ϕ, a more careful definition is needed, namely: $\mathsf{in}(\phi) \rightarrow (\mathsf{in}(\phi') \wedge (\phi' \rightarrow \phi))$, where $\mathsf{in}(\phi)$ characterizes the set of *legal* input assignments specified by ϕ. An input assignment is legal if there exists an output assignment such that together the two assignments satisfy the contract. For example, if ϕ is $x_2 \neq 0 \wedge y = \frac{x_1}{x_2}$ then $\mathsf{in}(\phi)$ is $x_2 \neq 0$.

The above definition of refinement applies to the stateless case where an interface has a single contract ϕ. The definition can be extended to the stateful case as shown in Section 9. Refinement is a partial order with the following main properties: (1) it is preserved by composition and hiding; and (2) it is essentially equivalent to substitutability (Theorem 15). It is worth noting that reasonable alternative definitions of refinement result in sufficient but not necessary conditions for substitutability (see discussions in Sections 2.4 and 9.2). Our notion of refinement thus seems to be "the best match" for substitutability.

Our theory supports *shared refinement* (Section 10), which is important for component reuse as argued in [Doyen et al. 2008]. Shared refinement of two interfaces I and I', when defined, is a new interface that refines both I and I', in fact, it is their greatest lower bound with respect to the refinement order, and is therefore denoted $I \sqcap I'$. In this paper we also propose *shared abstraction* $I \sqcup I'$, which is shown to be the least upper bound with respect to refinement.

As a special case, we discuss *input-complete* (sometimes also called *receptive*) interfaces, where contracts are total relations, and *deterministic* interfaces, where contracts are partial functions. These two subclasses of interfaces are interesting, first, because the theory is greatly simplified in those cases (refinement becomes language containment, composition becomes relational, etc.), and second, because there is an interesting duality between the two subclasses, as shown in Sections 11 and 12.

Examples illustrating the concepts of the theory are provided throughout the paper. An application to the hardware domain is described in Section 13.

The main features of the theory are summarized in Table I. This table is given merely for reference and contains only a partial view. The precise definitions and complete set of results are given in the sections that follow.

One of the appealing features of our theory is that it allows a *declarative* way of specifying contracts, and a *symbolic* way of manipulating them, as logical formulas. For this reason, it is relatively straightforward to develop algorithms that implement the theory for finite-state interfaces. Throughout the text we provide such algorithms, for composing interfaces, checking refinement, and so on. These algorithms compute some type of product of the automata that represent the in-

Models	Relational interfaces	Stateless: contracts (predicates) on input/output variables, e.g., $x_2 \neq 0 \wedge y = \frac{x_1}{x_2}$. Legal input assignments in contract ϕ: $\mathsf{in}(\phi) := (\exists y_1, y_2, ..., y_n : \phi)$, where $\{y_1, ..., y_n\}$ is the set of output variables, e.g., $\mathsf{in}(x_2 \neq 0 \wedge y = \frac{x_1}{x_2}) \equiv x_2 \neq 0$.
		Stateful: automata whose states are labeled with contracts (set of states may be infinite).
	Moore w.r.t. input x	Contract does not depend on current value of variable x (can still depend on previous values of x).
	Input-complete	All input values are legal: $\mathsf{in}(\phi) \equiv \mathsf{true}$.
	Deterministic	Given legal inputs, outputs are unique.
	Well-formed	All reachable contracts are satisfiable.
	Well-formable	Can be made well-formed by restricting the inputs.
	Environments	They are interfaces.
Compositions	Connection	Parallel: conjunction of contracts: $\phi := \phi_1 \wedge \phi_2$. Commutative & associative.
		Serial: game, environment vs. source interface: $\phi := \phi_1 \wedge \phi_2 \wedge \forall y_1, ..., y_n : \big(\phi_1 \rightarrow \mathsf{in}(\phi_2)\big)$, where $\{y_1, ..., y_n\}$ is the set of output variables of the source interface. Associative.
	Feedback	Interface must be Moore w.r.t. input variable x that is connected to output y. Commutative & associatve.
	Pluggability: $I \leftrightarrows E$	Closed-loop composition of I and E must be well-formed
	Substitutability: $I \rightarrow_e I'$	For any E, if I is pluggable to E then I' is pluggable to E
Compositionality	Refinement	For stateless: $\phi' \sqsubseteq \phi := \Big(\mathsf{in}(\phi) \rightarrow \big(\mathsf{in}(\phi') \wedge (\phi' \rightarrow \phi)\big)\Big)$. Similar for stateful. $\sqsubseteq$ is partial order. false is top element.
	Preservation	Refinement preserves well-formability.
		Refinement is preserved by both connection and feedback.
		Refinement sufficient for substitutability: if $I' \sqsubseteq I$ then $I \rightarrow_e I'$.
		Refinement necessary for substitutability, in that: if $I' \not\sqsubseteq I$ and I is well-formed, then $I \not\rightarrow_e I'$.
	Special case: input-complete	If ϕ_2 is input-complete then serial composition is conjunction. Refinement becomes: $\phi' \sqsubseteq \phi \equiv \big(\phi' \rightarrow \phi\big)$.
	Special case: deterministic	If ϕ_1 is deterministic then serial composition is conjunction. Refinement becomes: $\phi' \sqsubseteq \phi \equiv \big(\phi \rightarrow \phi'\big)$.

Table I. Summary of the main concepts and results of this paper.

terfaces and syntactically manipulate their contracts. Solving problems such as quantifier elimination and satisfiability checking on the formulas representing the contracts are crucial elements of the algorithms. Decidability of these problems will of course depend on the types of formulas used. Recent advances in *SMT (Satisfiability Modulo Theory)* solvers can be leveraged for this task.

2. MOTIVATION FOR THE DESIGN CHOICES

As mentioned in the introduction, our theory uses a game-theoretic interpretation of composition and refinement. These interpretations are by no means new (see Section 3) and have been extensively motivated in previous works. For the sake of self-containment, in this section we also motivate these choices in our setting.

2.1 A general model for contracts: relational, non-deterministic, non-input-complete

Consider a component performing division. One possible interface for this component is the following:

$$
\begin{aligned}
\mathit{Div} &:= (\{x_1, x_2\}, \{y\}, \phi^1_{\mathit{Div}}) \\
\phi^1_{\mathit{Div}} &:= x_2 \neq 0 \wedge \phi_{sign} \\
\phi_{sign} &:= (y = 0 \equiv x_1 = 0) \wedge \big(y < 0 \equiv (x_1 < 0 < x_2 \vee x_2 < 0 < x_1)\big)
\end{aligned}
$$

Div has two input variables x_1, x_2, one output variable y, and a contract represented by the predicate ϕ^1_{Div}. Interpreting x_1 to be the dividend and x_2 the divisor, and y to be the result of the division, ϕ^1_{Div} states that the divisor must be non-zero and also provides guarantees on the sign of the output depending on the sign of the inputs.

The following points are worth making about contract ϕ^1_{Div}. First, it is relational, in the sense that the value of the output depends on the values of the inputs. A non-relational contract that could be used is, for instance, $x_2 \neq 0$, which represents only an assumption on the input. Another non-relational contract, for a slightly more restrictive component that does not accept negative inputs, would be $x_1 \geq 0 \wedge x_2 > 0 \wedge y \geq 0$. This is non-relational in the sense that the guarantee on the output does not depend on the value of the inputs. The second point about ϕ^1_{Div} is that it is non-deterministic: the output y is not uniquely determined for a given input (unless $x_1 = 0$). The final point about ϕ^1_{Div} is that it is non-input-complete: all inputs where $x_2 = 0$ are *illegal* in the sense that they violate the contract.

As this example illustrates, "rich" contracts, that is, relational, non-deterministic, and non-input-complete, arise even in simple situations. The need to capture relations between inputs and outputs should be clear: if we separate input assumptions from output guarantees (as done in [Doyen et al. 2008], for instance) then we cannot state input-output properties about our system. The need for non-determinism should also be clear: non-determinism is useful when abstracting low-level details that would be too difficult to obtain or too expensive to use. For instance, in our example, we could use a deterministic contract for *Div*:

$$\phi^2_{\mathit{Div}} := x_2 \neq 0 \wedge x_1 = y \cdot x_2$$

But we may opt for ϕ^1_{Div}, since ϕ^1_{Div} can be handled by an SMT solver that can only deal with linear constraints, whereas ϕ^2_{Div} cannot.

The need for non-input-completeness may be less obvious. Why can't we replace the non-input-complete contract ϕ^1_{Div} by the input-complete contract

$$\phi^3_{\mathit{Div}} := x_2 \neq 0 \rightarrow \phi_{sign} \quad ?$$

There are several reasons. First, note that ϕ^3_{Div} allows inputs where $x_2 = 0$ (since an implication $A \rightarrow B$ is trivially satisfied when A is false) and in that case the output y may take any value. However, the implicit assumption is that y *will* take *some* value. In other words, a "real" component (in SW or HW) that implements ϕ^3_{Div} must be "input-complete" in the sense that it always produces some output, even when given illegal inputs. But not all real systems have this property. For example, a program may not terminate on illegal inputs; and a circuit may burn up if an incorrect voltage is applied to it. These systems are not "input-complete", thus cannot be described by input-complete interfaces.

But even when a system is "input-complete", we may still want to capture it with a non-input-complete interface. Indeed, suppose we connect *Div* to another component C, as shown to the left of Figure 1. Our intention is for C to output the constant 2, so that the composition implements a division by 2. But suppose that due to a design error C outputs zero instead. That is, the contract of C is $x_2 = 0$. Combining the latter with ϕ^1_{Div}, by taking the conjunction of the two, gives false, which means that two interfaces are incompatible. Catching this incompatibility early on, i.e., when attempting to compose C with *Div*, is useful, since it permits to localize and correct the error easily.

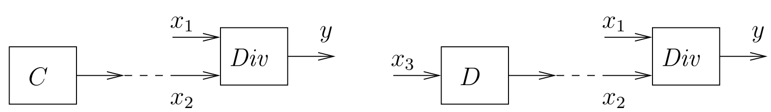

Fig. 1. Connecting C to *Div* (left) and D to *Div* (right).

Suppose we used the input-complete contract ϕ^3_{Div}, instead of ϕ^1_{Div}. Then, the composition of C and *Div* would result in the contract true (after hiding variable x_2). How should we interpret this result? We cannot in general interpret true as indicating incompatibility, since it might simply be the result of lack of information, i.e., trivial contracts. In a large system, there will generally be many components for which we have no information, and others for which we do. We want a systematic (or even automatic) method that distinguishes between "no information" and "incompatible composition".

We could of course perform a "local" verification task, in order to prove that the composition of C and *Div* implements the intended division by 2, namely, the property $\phi_P := (y \cdot 2 = x_1)$. Contract true fails to imply this, which indicates an error. The problem with this approach is that it requires ϕ_P to be specified. This may not always be an easy task. First, formal verification may not be part of the design process. Second, even if it is, it may be the case that only a global, end-to-end specification is available, for the top-level component within which the composition of C and *Div* is embedded. "Decomposing" such a global specification into local specifications such as ϕ_P is not always straightforward.

With non-input-complete interfaces, such local specifications are not required. Instead, a compatibility check is performed to ensure that a composition such as the one between C and *Div* is valid. This is a more light-weight verification process, akin to type checking (but with types that are quite richer than usual). In fact, the goal may not be verification at all, but rather *synthesis* of component interfaces, in a bottom-up fashion: given interfaces for atomic components C and *Div*, compute an interface for their composition. Instead of type-checking, this is akin to type-inference. Once an interface for a complete hierarchical model is computed, and assuming no incompatibilities have been found during the process, that interface can be checked against a global specification, if the latter is available. But the interface provides useful information that can be helpful even in the absence of such a specification.

2.2 On the definition of serial composition

Consider again interface *Div*, and suppose we connect it to another interface D, as shown to the right of Figure 1. Suppose that the contract of D is $\phi_D := \mathsf{true}$. If D abstracts a certain component, this may mean that we have no knowledge about this component (e.g., our automated abstraction tool gave us a trivial answer).

What should the contract of the composition of D and *Div* be? Standard composition of relations corresponds to taking the conjunction $\phi_D \wedge \phi^1_{Div}$, and then eliminating x_2. This yields the formula $\exists x_2 : \phi^1_{Div}$, which is equivalent to the predicate $y = 0 \equiv x_1 = 0$, asserting that y is zero iff x_1 is zero. This assertion is satisfiable (there are inputs and outputs that make it true), therefore, it would appear that the composition of D and *Div* is valid.

Now, suppose that we want to replace D by E, which has the same structure as D (i.e., same input and output variables) but a different contract, namely, $\phi_E := x_2 = 0$. ϕ_E provides stronger output guarantees than ϕ_D, and in any standard framework this means that E refines D (this is also true in our framework). But clearly the composition of E and *Div* is not valid. This violates one of the main properties of any compositional theory, namely, that a component should be replaceable by any component that refines it.

We are therefore forced to revise our assumption that the composition of D and *Div* is valid. The problem is that we interpreted the non-determinism of D as "angelic", or "controllable". We should instead interpret it as "demonic", or "uncontrollable". We should accept the composition as valid only if there exist input values at x_3 for which it can be *guaranteed* that *any* possible output of D satisfies $x_2 \neq 0$. Since D does not guarantee anything, no such input at x_3 can be found. Therefore, the composition of D and *Div* should be considered invalid. Logically, we achieve this by adding the term $\forall x_2 : \phi_D \rightarrow x_2 \neq 0$ to the conjunction $\phi_D \wedge \phi^1_{Div}$, in the definition of the composite contract. The above term reduces to $\forall x_2 : \mathsf{true} \rightarrow x_2 \neq 0$, or $\forall x_2 : x_2 \neq 0$, which is false.

2.3 On the definition of refinement

Once we accept the "demonic" interpretataion of non-determinism in composition, as described above, the choice of refinement appears to be inevitable. Indeed, we seek a refinement relation that is equivalent to substitutability. This means that refinement must be both sufficient for substitutability (i.e., if interface I' refines interface I, then I' can replace I in any context) as well as necessary (i.e., if I' does not refine I, then there is a context where I works but I' does not). As it turns out, the definition that has these properties is the following: contract ϕ' refines contract ϕ, denoted $\phi' \sqsubseteq \phi$, iff the condition $\mathsf{in}(\phi) \rightarrow \big(\mathsf{in}(\phi') \wedge (\phi' \rightarrow \phi)\big)$ is satisfied for any input/output assignment (this is the simplified definition for stateless interfaces, the general definition is given in Section 9). Alternative definitions can be given, which result in sufficient but not necessary conditions for substitutability, as discussed in Sections 2.4 and 9.2.

Refinement is not the same as logical implication. As an example, consider three possible contracts for our division component:

$$\begin{aligned}
\phi^1_{Div} &:= x_2 \neq 0 \wedge (y = 0 \equiv x_1 = 0) \wedge \big(y < 0 \equiv (x_1 < 0 < x_2 \vee x_2 < 0 < x_1)\big) \\
\phi^2_{Div} &:= x_2 \neq 0 \wedge x_1 = y \cdot x_2 \\
\phi^4_{Div} &:= x_2 \neq 0 \rightarrow x_1 = y \cdot x_2
\end{aligned}$$

It can be verified that $\phi^4_{Div} \sqsubseteq \phi^2_{Div} \sqsubseteq \phi^1_{Div}$. Yet observe that $\phi^2_{Div} \rightarrow \phi^1_{Div}$ and $\phi^2_{Div} \rightarrow \phi^4_{Div}$.

2.4 Error-complete interfaces

Illegal inputs can also be captured using input-complete interfaces that have a special boolean output variable e, denoting an "error". In particular, a contract ϕ over input and output variables $X \cup Y$ can be transformed into an input-complete contract $\mathsf{EC}(\phi)$ over $X \cup Y \cup \{e\}$, called the *error-completion* of ϕ. $\mathsf{EC}(\phi)$ sets e to false when the input is legal for ϕ, and to true otherwise:

$$\mathsf{EC}(\phi) \;:=\; (\phi \wedge \neg e) \vee (\neg \mathsf{in}(\phi) \wedge e) \tag{1}$$

For example, the error-completion of the division interface Div^1 yields:

$$\begin{aligned}
Div^1_e &:= (\{x_1, x_2\}, \{y, e\}, \phi_e), \text{ where } \phi_e := \mathsf{EC}(\phi^1_{Div}) \\
\phi_e &\equiv (x_2 \neq 0 \wedge \phi_1 \wedge \neg e) \vee (x_2 = 0 \wedge e)
\end{aligned}$$

We can retrieve ϕ from $\mathsf{EC}(\phi)$ using the inverse transformation:

$$\mathsf{EC}^{-1}(\phi_e) \;:=\; (\exists e : \phi_e) \wedge (\forall Y \cup \{e\} : \phi_e \rightarrow \neg e) \tag{2}$$

It can be shown that, if ϕ_e is of the form described in (1), then $(\exists e : \phi_e) \equiv (\phi \vee \neg \mathsf{in}(\phi))$ and $(\forall Y \cup \{e\} : \phi_e \rightarrow \neg e) \equiv \mathsf{in}(\phi)$. That is, the first term is the *input-completion* of ϕ which adds all its illegal inputs in its domain, whereas the second term isolates the legal inputs. The conjunction of these two terms gives ϕ. Therefore, for any contract ϕ, we have:

$$\phi \;\equiv\; \mathsf{EC}^{-1}(\mathsf{EC}(\phi)) \tag{3}$$

On the other hand, for contracts ϕ_e over $X \cup Y \cup \{e\}$, $\mathsf{EC}(\mathsf{EC}^{-1}(\phi_e))$ is not always equivalent to ϕ_e. For example, if $\phi_e := y \equiv e$ and y is an output, then $\mathsf{EC}^{-1}(\phi_e) \equiv \mathsf{false}$, and $\mathsf{EC}(\mathsf{EC}^{-1}(\phi_e)) \equiv e$. Indeed, EC is injective but not surjective. It is unclear what is the meaning of a contract such as $y \equiv e$. This contract appears to "misuse" the error variable e which is supposed to capture validity of inputs.

It appears that game-theoretic serial composition of two contracts can be performed as a sequence of steps: error-completion, standard relational composition, and inverse error-completion. We illustrate this with an example. Consider the composition of interfaces D and Div discussed above. D is already input-complete, so its error-completion is unnecessary ($\mathsf{EC}(\phi_D)$ would still extend ϕ_D with an additional error output variable, but we omit this for the sake of simplicity). Let $\phi_e := \mathsf{EC}(\phi^1_{Div})$. Let ϕ_{err} be the relational composition of ϕ_D and ϕ_e, that is: $\phi_{err} := \exists x_2 : (\mathsf{true} \wedge \phi_e)$. It can be verified that $\phi_{err} \equiv \mathsf{true}$. Then:

$$\mathsf{EC}^{-1}(\phi_{err}) \equiv (\exists e : \mathsf{true}) \wedge (\forall e : \mathsf{true} \rightarrow \neg e) \equiv \mathsf{true} \wedge \mathsf{false} \equiv \mathsf{false}$$

This is indeed the same as the result obtained in Section 2.2 and indicating that the composition of D and Div is invalid.

Refinement between two contracts is different from logical implication of their error-transformed versions, that is, $\phi_2 \sqsubseteq \phi_1$ is not equivalent to $\mathsf{EC}(\phi_2) \rightarrow \mathsf{EC}(\phi_1)$. In particular, although validity of $\mathsf{EC}(\phi_2) \rightarrow \mathsf{EC}(\phi_1)$ is a sufficient condition for $\phi_2 \sqsubseteq \phi_1$, it is not a necessary condition. To see why, consider two interfaces that only model assumptions on an input variable x, and having contracts $x > 0$ and true, respectively. true accepts more inputs than $x > 0$, therefore we have $\mathsf{true} \sqsubseteq x > 0$. Now consider $\psi_1 := \mathsf{EC}(x > 0)$ and $\psi_2 := \mathsf{EC}(\mathsf{true})$. We have:

$$
\begin{aligned}
\psi_1 &\equiv (x > 0 \wedge \neg e) \vee (x \leq 0 \wedge e) \\
\psi_2 &\equiv \neg e
\end{aligned}
$$

Clearly, $\psi_2 \not\rightarrow \psi_1$. Because of Theorem 15, which states equivalence of refinement and substitutability, this example also shows that $\mathsf{EC}(\phi_2) \rightarrow \mathsf{EC}(\phi_1)$ is a sufficient but not necessary condition for substitutability.

In summary, it appears that: (1) Error-complete interfaces can be used to capture the same information as that contained in relational interfaces. However, the class of error-complete interfaces is larger, and some of these interfaces have no direct meaning as relational interfaces. Thus, relational interfaces appear to be a more "canonical" representation. Moreover, relational interfaces avoid the overhead of designating special error outputs whose semantics differ from other outputs. (2) Composition of relational interfaces can be defined as standard relational composition of their error-complete counterparts. However, in order to check whether such a composition is valid, the inverse transformation needs to be computed. This inverse transformation involves solving a game, therefore, the game-theoretic interpretation of composition is not avoided. (3) Implication of their error-complete counterparts is a strictly stronger condition than refinement/substitutability between two interfaces.

Based on these observations, it appears that our theory could be formulated essentially equivalently in terms of error-complete interfaces. We do not pursue this option, however, as relational interfaces without special error outputs seem more elegant to us. On the other hand, error-complete interfaces are worth studying in greater depth, since error variables can be used for additional purposes than simply indicating illegal inputs. For instance, they may be used to indicate faulty behavior of a component. An in-depth study of these possibilities is beyond the scope of the current paper and part of future work.

3. RELATED WORK

Most of the ideas upon which this work is based, such as stepwise refinement, interfaces, design-by-contract, and game semantics, are by no means new. The main contribution of this paper is the application of these ideas to the development of a working theory for synchronous concurrent systems.

In particular, abstracting components in some mathematical framework that offers stepwise refinement and compositionality guarantees is an idea that goes back to the work of Floyd and Hoare on proving program correctness using pre- and post-conditititions [Floyd 1967; Hoare 1969] and the work of Dijkstra and Wirth on stepwise refinement as a method for gradually developing programs from their specifications [Dijkstra 1972; Wirth 1971]. A pair of pre- and post-conditions can be seen as a contract for a piece of sequential code. These ideas were further developed in a large number of works, including the Z notation [Spivey 1989], the B method [Abrial 1996], CLU [Liskov 1979], Eiffel and the design-by-contract paradigm [Meyer 1992], the refinement calculus [Back and Wright 1998], Larch [Guttag and Horning 1993; Leavens 1994; Cheon and Leavens 1994], and JML [Leavens and Cheon 2006]. Viewing programs as predicates or relations is also not new, for instance, see [Hoare 1985], [Parnas 1983], [Frappier et al. 1998], and [Kahl 2003].

The above works are primarily about sequential programs and therefore are not directly comparable with our framework which is about synchronous concurrent systems. For instance, our model has distinct notions of input and output variables, whereas sequential programs operate on a set of shared variables, i.e., a global, shared state. A program can be modeled as a relation between values of these global variables before and after program execution, i.e., “pre” and “post” variables. However, it seems that our composition operators cannot be directly mapped to those aiming to capture typical constructs in sequential programs, such as “if-then-else” or “while” statements. Consider feedback composition, for example. One might attempt to map this to some form of while statement. But while statements operate on the same set of global variables, which could be seen as a special case of feedback where there is a 1-1 correspondence between inputs and outputs (i.e., pre and post variables). In the general case, an arbitrary output variable can be connected to an arbitrary input, which seems to make denotational approaches such as lifting to powersets inapplicable.

In fact, many of the above works start with a programming language in mind, and then define the pre/post-conditions, abstractions, or interfaces, for this particular language. As is characteristically stated in [Hoare 1985], programs are predicates, but not all predicates are programs. In contrast, our framework is “implementation-agnostic” in the sense that we are not concerned with whether components are implemented in HW or in SW, or in which programming language. For this reason, as well as the fact that non-implementable predicates such as false may arise as a result of composition, we do not attempt to restrict the set of predicates that we consider as contracts.

Another difference regards concerns about program termination, which naturally arise in sequential programs that contain “while” loops. How can non-termination be modeled when programs are captured by relations? This question has received a lot of attention in the literature (an excellent survey can be found in [Nelson 1989]) and has also generated some controversy [Hehner and Parnas 1985]. Our take on this is simple: if a component S may not terminate on a given input value a, then the contract for S should reject a as illegal. That is, the input-output relation for S is partial. In terms of Nelson’s classification, our model can be seen as an instance of the “partial correctness model” [Nelson 1989]. This model does not allow to distinguish a component S that *never* terminates on input a, from another component S' that *may or may not* terminate on a. We do not worry about this loss of expressiveness, however, because in our context, all components must be *guaranteed* to terminate and produce a value at *every* synchronous round. As a result, if an input *may* result in non-termination, it is appropriate to consider this input as illegal. Therefore, S' can be safely abstracted by the same interface as S.

Despite these differences, our approach follows many of the principles advocated in the works mentioned above. In particular, we abide to the design-by-contract paradigm and the well-known principle of refinement by weakening the pre-condition and strengthening the post-condition (although "strengthening the post condition" must be defined carefully in the general, non-input-complete case, as discussed in Section 9.2). Also, we use a "demonic" interpretation of non-determinism during composition, as some of the works above also do [Back and Wright 1998; Frappier et al. 1998]. Computing composition then amounts to finding strategies in a game, or equivalently, solving a controller synthesis problem [de Alfaro 2004].

Interfaces can be viewed as "rich", behavioral types, as suggested in [Lee and Xiong 2001; de Alfaro and Henzinger 2001a]. Behavioral types have been studied in a number of works in the context of sequential and object-oriented programming, such as [Nierstrasz 1993; Liskov and Wing 1994; Dhara and Leavens 1996]. Behavioral subtyping notions defined in the above works follow the same principle of input-contravariance/output-covariance also in our refinement, but there are subtle differences in their definitions. For instance, both the "Post-condition rule" $m_\sigma.post \Rightarrow m_\tau.post$ and the "Constraint rule" $C_\sigma \Rightarrow C_\tau$, defined in Figure 4 of [Liskov and Wing 1994] as requirements for type σ to be a subtype of type τ, appear to follow the rule $\phi' \to \phi$ rather than the rule $(\mathsf{in}(\phi) \wedge \phi') \to \phi$ which is used in our refinement. As explained in Section 9.2, $\phi' \to \phi$ is too strong in the sense that it is not a necessary condition for substitutability. [Dhara and Leavens 1996] weaken the subtyping requirements of [Liskov and Wing 1994], but maintain the $\phi' \to \phi$ rule.

The works mentioned so far focus on sequential programs. In a concurrency setting, a powerful compositional framework is FOCUS [Broy 1997; Broy and Stølen 2001]. FOCUS is a relational framework where specifications are relations on input-output *streams*. The FOCUS framework is in many respects more general than ours, in that it can capture relations that do not preserve the length of input streams. For this reason, FOCUS is applicable also to asynchronous systems. On the other hand, FOCUS targets mainly the input-complete case. I/O automata [Lynch and Tuttle 1989] are also related to our work, but are input-complete by definition. Reactive modules [Alur and Henzinger 1999] are also input-complete.

Dill's *trace theory* is another compositional framework for concurrent systems, focusing on asynchronous concurrency and motivated in particular by the design of asynchronous circuits [Dill 1987]. In trace theory, a component is described using a pair of sets of traces, called *successes* and *failures*, for legal and illegal behaviors, respectively. A *trace* is a sequence of events, and an event is a change in the value of an input or output variable. Because no synchrony is assumed, the number of input and output events in a trace can be arbitrary. Trace theory considers *prefix-closed trace structures*, where the success and failure sets are prefix-closed regular sets, aimed at verification of safety properties, as well as *complete trace structures*, where these are general sets of infinite traces, aimed at liveness properties. Our theory is currently restricted to prefix-closed sets and therefore cannot handle liveness properties. However, it is worth noting that, contrary to prefix-closed trace structures, our theory avoids the problem of trivial implementations that achieve the specification by "doing nothing".

In trace theory, refinement is called *conformation* and is achieved by restricting the set of failures as well as the global set of traces (failures can be turned into successes during refinement). Conformation follows the "accept more inputs, produce less outputs" principle that has later been studied in the context of alternating refinement relations [Alur et al. 1998]. It is worth noting that conformation induces a lattice on prefix-closed trace structures, whereas our refinement relation is only a partial order and in particular has no "bottom" element.

Like trace structures, the framework of interface automata [de Alfaro and Henzinger 2001a] uses an asynchronous model of concurrency. Compared to trace structures, interface automata are more "syntactic" in nature since the interface is the automaton itself (as opposed to, say, a set of traces that can be represented by an automaton). Modal interfaces [Raclet et al. 2010] also focus on asynchronous systems and work directly with an automata representation. It is an interesting question to what extent these automata-based models can be used to capture synchronous systems and input-output relations within a synchronous round. If possible to do so, the result would most likely have an operational flavor, contrary to our framework, which is of a more declarative, denotational and symbolic nature. For instance, to express variables with infinite domains in the above formalisms, one would typically need an infinite set of events; to express a relation such as $y = x + 1$ one would need an infinite set of transitions; and so on.

Our theory can be used as a behavioral type theory for Simulink and related models, in the spirit of [Roy and Shankar 2010]. In the latter work, Simulink blocks are annotated with constraints on input and output variables much like the stateless contracts considered in our work. Our framework provides an extension of such types to the stateful case, as well as the formalization of compositions and refinement which are not considered in [Roy and Shankar 2010].

Within the domain of interface theories, [de Alfaro and Henzinger 2001b] defines *relational nets*, which are networks of processes that non-deterministically relate input values to output values. [de Alfaro and Henzinger 2001b] does not provide an interface theory for the complete class of relational nets. Instead it provides interface theories for subclasses, in particular: *rectangular nets* which have no input-output dependencies; *total nets* which can have input-output dependencies but are input-complete; and *total and rectangular nets* which combine both restrictions above. The interfaces provided in [de Alfaro and Henzinger 2001b] for rectangular nets are called *assume/guarantee (A/G) interfaces*. A/G interfaces form a strict subclass of the relational interfaces that we consider in this paper: A/G interfaces separate the assumptions on the inputs from the guarantees on the outputs, and as such cannot capture input-output relations; on the other hand, every A/G contract can be trivially captured as a relational contract by taking the conjunction of the assume and guarantee parts. [de Alfaro and Henzinger 2001b] studies *stateless* A/G interfaces, while [Doyen et al. 2008] studies also *stateful* A/G interfaces, in a synchronous setting similar to the one considered in this paper. [Doyen et al. 2008] also discusses *extended interfaces* which are essentially the same as the relational interfaces that we study in this paper. However, difficulties with synchronous feedback loops (see discussion below) lead [Doyen et al. 2008] to conclude that extended interfaces are not appropriate.

[Chakrabarti et al. 2002] considers synchronous *Moore interfaces*, defined by two formulas ϕ_i and ϕ_o that specify the legal values of the input and output variables, respectively, at the *next* round, given the current state. This formulation does not allow to describe relations between inputs and outputs within the same round, as our relational theory allows.

Both [de Alfaro and Henzinger 2001b] and [Doyen et al. 2008] can handle very general compositions of interfaces, that can be obtained via parallel composition and arbitrary connection (similar to the denotational composition framework of [Lee and Sangiovanni-Vincentelli 1998]). This allows, in particular, arbitrary feedback loops to be created. In a relational framework, however, synchronous feedback loops can be problematic, as discussed in Example 14 (see also Section 14).

Interface theories are naturally related to work on compositional verification, where the main purpose is to break down the task of checking correctness of a large model into smaller tasks, that are more amenable to automation. A very large body of research exists on this topic. Some of this work is based on an asynchronous, interleaving based concurrency model, e.g., see [Misra and Chandy 1981; Stark 1985; Jonsson 1994], some on a synchronous model, e.g., see [Grumberg and Long 1994; McMillan 1997], while others are done within a temporal logic framework, e.g., see [Barringer et al. 1984; Abadi and Lamport 1995]. Many of these works are based on the assume-guarantee paradigm, and they typically use some type of trace inclusion or simulation as refinement relation [Jones 1983; Stark 1985; Shankar 1998; Henzinger et al. 1998].

4. PRELIMINARIES, NOTATION

We use first-order logic (FOL) notation throughout the paper. For an introduction to FOL, see, for instance, [Tourlakis 2008]. We use true and false for logical constants true and false, $\neg, \wedge, \vee, \rightarrow, \equiv$ for logical negation, conjunction, disjunction, implication, and equivalence, and $\exists$ and $\forall$ for existential and universal quantification, respectively. We use $:=$ when defining concepts or introducing new notation: for instance, $x_0 := \max\{1,2,3\}$ defines x_0 to be the maximum of the set $\{1,2,3\}$.

Let V be a finite set of variables. A *property over* V is a FOL formula ϕ such that any free variable of ϕ is in V. The set of all properties over V is denoted $\mathcal{F}(V)$. Let ϕ be a property over V and V' be a finite subset of V, $V' = \{v_1, v_2, ..., v_n\}$. Then, $\exists V' : \phi$ is shorthand for $\exists v_1 : \exists v_2 : ... : \exists v_n : \phi$. Similarly, $\forall V' : \phi$ is shorthand for $\forall v_1 : \forall v_2 : ... : \forall v_n : \phi$.

We will implicitly assume that variables are *typed*, meaning that every variable is associated with a certain *domain*. An *assignment* over a set of variables V is a (total) 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 $\mathcal{A}(V)$. If a is an assignment over V_1 and b is an assignment over V_2, and V_1, V_2 are disjoint, we use (a, b) to denote the combined assignment over $V_1 \cup V_2$. A formula ϕ is *satisfiable* iff there exists an assignment a over the free variables of ϕ such that a satisfies ϕ, denoted $a \models \phi$. A formula ϕ is *valid* iff it is satisfied by every assignment.

There is a natural mapping from formulas to sets of assignments, that is, from

$\mathcal{F}(V)$ to $2^{\mathcal{A}(V)}$. In particular, a formula $\phi \in \mathcal{F}(V)$ can be interpreted as the set of all assignments over V that satisfy ϕ. Conversely, we can map a subset of $\mathcal{A}(V)$ to a formula over V, provided this subset is representable in FOL. Because of this correspondence, we use set-theoretic or logical notation, as is more convenient. For instance, if ϕ and ϕ' are formulas or sets of assignments, we write $\phi \wedge \phi'$ or $\phi \cap \phi'$ interchangeably.

If S is a set, S^* denotes the set of all finite sequences of elements of S. S^* includes the empty sequence, denoted ε. If $s, s' \in S^*$, then $s \cdot s'$ is the concatenation of s and s'. $|s|$ denotes the *length* of $s \in S^*$, with $|\varepsilon| = 0$ and $|s \cdot a| = |s| + 1$, for $a \in S$. If $s = a_1 a_2 \cdots a_n$, then the i-th element of the sequence, a_i, is denoted s_i, for $i = 1, ..., n$. A *prefix* of $s \in S^*$ is a sequence $s' \in S^*$ such that there exists $s'' \in S^*$ such that $s = s' \cdot s''$. We write $s' \leq s$ if s' is a prefix of s. $s' < s$ means $s' \leq s$ and $s' \neq s$. A subset $L \subseteq S^*$ is *prefix-closed* if for all $s \in L$, for all $s' \leq s$, $s' \in L$.

5. RELATIONAL INTERFACES

DEFINITION 1 RELATIONAL INTERFACE. *A* relational interface *(or simply* interface*) is a tuple $I = (X, Y, f)$ where X and Y are two finite and disjoint sets of* input *and* output variables*, respectively, and f is a non-empty, prefix-closed subset of $\mathcal{A}(X \cup Y)^*$.*

Note that $\mathcal{A}(X \cup Y)$ can be infinite. In the case variables in X and Y have finite domains, $\mathcal{A}(X \cup Y)$ is finite and can be seen as a finite alphabet. In that case, f is a non-empty, prefix-closed language over that alphabet.

We write $\mathsf{InVars}(I)$ for X and $f(I)$ for f. We allow X or Y to be empty: if X is empty then I is a *source* interface; if Y is empty then I is a *sink*. An element of $\mathcal{A}(X \cup Y)^*$ is called a *state*. That is, we identify states with observation histories. The *initial state* is the empty sequence ε. The states in f are also called the *reachable states* of I. f defines a total function that maps a state to a set of input-output assignments. We use the same symbol f to refer to this function. For $s \in \mathcal{A}(X \cup Y)^*$, $f(s)$ is defined as follows:

$$f(s) := \{a \in \mathcal{A}(X \cup Y) \mid s \cdot a \in f\}.$$

We view $f(s)$ as a *contract* between a component and its environment *at that state*. The contract changes dynamically, as the state evolves.

Conversely, if we are given a function $f : \mathcal{A}(X \cup Y)^* \to 2^{\mathcal{A}(X \cup Y)}$, we can define a non-empty, prefix-closed subset of $\mathcal{A}(X \cup Y)^*$ as follows:

$$f := \{a_1 \cdots a_k \mid \forall i = 1, ..., k : a_i \in f(a_1 \cdots a_{i-1})\}$$

Notice that $\varepsilon \in f$ because the condition above trivially holds for $k = 0$. Also note that if $s \notin f$ then $f(s) = \emptyset$. This is because f is prefix-closed.

Because of the above 1-1 correspondence, in the sequel, we treat f either as a subset of $\mathcal{A}(X \cup Y)^*$ or as a function that maps states to contracts, depending on what is more convenient. We will assume that $f(s)$ is representable by a FOL formula. Therefore, $f(s)$ can be seen also as an element of $\mathcal{F}(X \cup Y)$.

DEFINITION 2 INPUT ASSUMPTIONS. *Given a contract $\phi \in \mathcal{F}(X \cup Y)$, the* input assumption *of ϕ is the formula $\mathsf{in}(\phi) := \exists Y : \phi$. Note that $\mathsf{in}(\phi)$ is a property over*

X. Also note that $\phi \rightarrow \mathsf{in}(\phi)$ is a valid formula for any ϕ.

A relational interface $I = (X, Y, f)$ can be seen as specifying a game between a component and its environment. The game proceeds in a sequence of *rounds*. At each round, an assignment $a \in \mathcal{A}(X \cup Y)$ is chosen, and the game moves to the next round. Therefore, the history of the game is the sequence of rounds played so far, that is, a state $s \in \mathcal{A}(X \cup Y)^*$. Suppose that at the beginning of a round the state is s. Typically, the environment plays first, by choosing $a_X \in \mathcal{A}(X)$. If $a_X \notin \mathsf{in}(f(s))$ then this is not a legal input and the environment loses the game. Otherwise, the component plays by choosing $a_Y \in \mathcal{A}(Y)$. If $(a_X, a_Y) \notin f(s)$ 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, with new state $s \cdot (a_X, a_Y)$. There are cases when the interface is *Moore* in the sense that its current outputs do not depend on its current inputs (the formal definition is given in Section 6.2). In this case, the component plays first. More general games can also be considered where the assignments of values to input and output variables are interleaved in an arbitrary order. The study of such a generalization is beyond the scope of the current work.

An *input-complete* interface is one that does not restrict its inputs:

Definition 3 Input-complete interface. *An interface $I = (X, Y, f)$ is* input-complete *if for all $s \in \mathcal{A}(X \cup Y)^*$, $\mathsf{in}(f(s))$ is valid.*

It is important to note that in our framework, input assumptions ("pre-conditions") and output guarantees ("post-conditions") are not separated. It is then crucial to distinguish a non-input-complete interface with a contract of the form $\phi_{pre} \wedge \phi$ and its input-complete version with contract $\phi_{pre} \rightarrow \phi$. These contracts are different (as we will show in Section 11, the latter refines the former). As mentioned in Section 3, our theory is mostly implementation-agnostic, and therefore does not prescribe how illegal inputs should be interpreted in the "real" component that an interface abstracts. As stated in Section 2, an illegal input may correspond to an input that results in non-termination of a SW component, or it may be an input that must be avoided by design, as in a type-checking setting.

A *deterministic* interface is one that maps every input assignment to at most one output assignment:

Definition 4 Determinism. *An interface $I = (X, Y, f)$ is* deterministic *if for all $s \in f$, for all $a_X \in \mathsf{in}(f(s))$, there is a unique $a_Y \in \mathcal{A}(Y)$ such that $(a_X, a_Y) \in f(s)$.*

The specializations of our theory to input-complete and deterministic interfaces are discussed in Sections 11 and 12, respectively.

A *stateless* interface is one where the contract is independent from the state:

Definition 5 Stateless interface. *An interface $I = (X, Y, f)$ is* stateless *if for all $s, s' \in \mathcal{A}(X \cup Y)^*$, $f(s) = f(s')$.*

For a stateless interface, we can treat f as a subset of $\mathcal{A}(X \cup Y)$ instead of a subset of $\mathcal{A}(X \cup Y)^*$. For clarity, if I is stateless, we write $I = (X, Y, \phi)$, where ϕ is a property over $X \cup Y$.

EXAMPLE 1 STATELESS INTERFACES. *Consider a component which is supposed to take as input a positive number n and return n or $n+1$ as output. We can capture such a component in different ways. One way is to use the following stateless interface:*

$$I_1 := (\{x\}, \{y\}, x > 0 \wedge (y = x \vee y = x + 1)\}).$$

Here, x is the input variable and y is the output variable. The contract of I_1 explicitly forbids zero or negative values for x. Indeed, we have $\mathsf{in}(f(I_1)) \equiv x > 0$.

Another possible stateless interface for this component is:

$$I_2 := (\{x\}, \{y\}, x > 0 \rightarrow (y = x \vee y = x + 1)\}).$$

The contract of I_2 is different from that of I_1: it allows $x \leq 0$, but makes no guarantees about the output y in that case. I_2 is input-complete, whereas I_1 is not. Both I_1 and I_2 are non-deterministic.

In general, the state space of an interface is infinite. In some cases, however, only a finite set of states is needed to specify f. In particular, f may be specified by a finite-state automaton:

DEFINITION 6 FINITE-STATE INTERFACE. *A* finite-state interface *is specified by a finite-state automaton $M = (X, Y, L, \ell_0, C, T)$. X and Y are sets of input and output variables, respectively. L is a finite set of* locations *and $\ell_0 \in L$ is the initial location. $C : L \rightarrow 2^{\mathcal{A}(X \cup Y)}$ is a* labeling function *that labels every location with a set of assignments over $X \cup Y$, the contract at that location. $T \subseteq L \times 2^{\mathcal{A}(X \cup Y)} \times L$ is a set of* transitions. *A transition $t \in T$ is a tuple $t = (\ell, g, \ell')$ where ℓ, ℓ' are the source and destination locations, respectively, and $g \subseteq \mathcal{A}(X \cup Y)$ is the* guard *of the transition. We require that, for all $\ell \in L$:*

$$C(\ell) = \bigcup_{(\ell, g, \ell') \in T} g \tag{4}$$

$$\forall (\ell, g_1, \ell_1), (\ell, g_2, \ell_2) \in T : \ell_1 \neq \ell_2 \rightarrow g_1 \cap g_2 = \emptyset \tag{5}$$

These conditions ensure that there is a unique outgoing transition for every assignment that satisfies the contract of the location. Given $a \in C(\ell)$, the a-successor of ℓ is the unique location ℓ' for which there exists transition (ℓ, g, ℓ') such that $a \in g$. A location ℓ is called reachable *if, either $\ell = \ell_0$, or there exists a reachable location ℓ', a transition (ℓ', g, ℓ), and an assignment a such that ℓ is the a-successor of ℓ'.*

M defines interface $I = (X, Y, f)$ where f is the set of all sequences $a_1 \cdots a_k \in \mathcal{A}(X \cup Y)^*$, such that for all $i = 1, ..., k$, $a_i \in C(\ell_{i-1})$, where ℓ_i is the a_i-successor of ℓ_{i-1}.

Note that a finite-state interface can still have variables with infinite domains. If the domains of variables are finite, however, then a finite-state interface can be seen as a prefix-closed regular language. Also notice that we allow $C(\ell)$, the contract at location ℓ, to be empty. This simply means that the interface is not well-formed (see Definition 7 that follows). Finally, although the guard of an outgoing transition from a certain location must be a subset of the contract of that location, we will often abuse notation and violate this constraint in the examples that follow, for the sake of simplicity. Implicitly, all guards should be understood in conjunction with the contracts of their source locations.

It is also worth noting that although the finite-state automaton defining a finite-state interface is deterministic, this does not mean that the interface itself is deterministic. Indeed, in general, it is not, since contracts that label locations are still non-deterministic input-output relations.

EXAMPLE 2 STATEFUL INTERFACE. *Figure 2 shows a finite-state automaton defining a finite-state interface that captures a single-place buffer. The interface has two input variables,* write *and* read*, and two output variables,* empty *and* full*. All variables are boolean. The automaton has two locations, ℓ_0 (the initial location) and ℓ_1. Each location is implicitly annotated by the conjunction of a* global contract*, that must hold at all locations, and a* local contract*, specific to a location. The global contract specifies that the buffer cannot be both empty and full (this is a guarantee on the outputs) and that a user of the buffer cannot read and write at the same round (this is an assumption on the inputs). The global contract also specifies that if the buffer is full then writing is not allowed, and if the buffer is empty then read is not allowed. The local contract at ℓ_0 states that the buffer is empty and at ℓ_1 that it is full.*

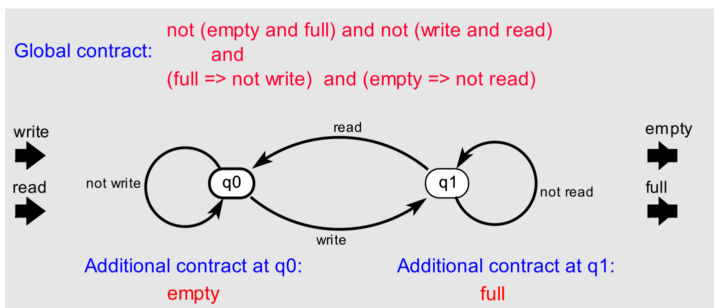

Fig. 2. Stateless and finite-state interfaces for a buffer of size 1.

DEFINITION 7 WELL-FORMEDNESS. *An interface $I = (X, Y, f)$ is* well-formed *iff for all $s \in f$, $f(s)$ is non-empty.*

Well-formed interfaces can be seen as describing components that never "deadlock". If I is well-formed then for all $s \in f$ there exists assignment a such that $s \cdot a \in f$. Moreover, f is non-empty and prefix-closed by definition, therefore, $\varepsilon \in f$. This means that there exists at least one state in f which can be extended to arbitrary length. In a finite-state interface, checking well-formedness amounts to checking that the contract of every reachable location of the corresponding automaton is satisfiable. If contracts are specified in a decidable logic, checking well-formedness of finite-state interfaces is thus decidable.

EXAMPLE 3. *Let I be the finite-state interface represented by the left-most automaton shown in Figure 3. I is assumed to have two boolean variables, an input*

x, and an output y. I is not well-formed, because it has reachable states with contract false *(all states starting with x being* false*). I can be transformed into a well-formed interface by strengthening the contract of the initial state from* true *to x, thus obtaining interface I' shown to the right of the figure.*

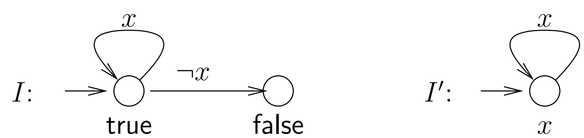

Fig. 3. A well-formable interface I and its well-formed witness I'.

Example 3 shows that some interfaces, even though they are not well-formed, can be turned into well-formed interfaces by appropriately restricting their inputs. This motivates the following definition:

Definition 8 Well-formability. *An interface $I = (X, Y, f)$ is* well-formable *if there exists a well-formed interface $I' = (X, Y, f')$ such that: for all $s \in f'$, $f'(s) \equiv f(s) \wedge \phi_s$, where ϕ_s is some property over X.*

Lemma 1. *Let $I = (X, Y, f)$ be a well-formable interface and let $I' = (X, Y, f')$ be a witness to the well-formability of I. Then $f' \subseteq f$.*

Proofs can be found in the appendix.

Clearly, every well-formed interface is well-formable, but the opposite is not true in general, as Example 3 shows. For stateless or source interfaces, however, the two notions coincide:

Theorem 1. *A stateless or source interface I is well-formed iff it is well-formable.*

For an interface that is finite-state and whose contracts are written in a logic for which satisfiability is decidable, there is an algorithm to check whether the interface is well-formable, and if this is the case, to transform it into a well-formed interface. The algorithm essentially attempts to find a winning strategy in a *game*, and as such is similar in spirit to algorithms proposed in [de Alfaro and Henzinger 2001a]. The algorithm starts by marking all locations with unsatisfiable contracts as *illegal*. Then, a location ℓ is chosen such that ℓ is legal, but has an outgoing transition (ℓ, g, ℓ'), such that ℓ' is illegal. If no such ℓ exists, the algorithm stops. Otherwise, the contract of ℓ is strengthened to

$$C(\ell) \;:=\; C(\ell) \wedge (\forall Y : C(\ell) \to \neg g) \tag{6}$$

$\forall Y : C(\ell) \to \neg g$ is a property on X. An input assignment a_X satisfies this formula iff, for any possible output assignment a_Y that the contract $C(\ell)$ can produce given a_X, the complete assignment (a_X, a_Y) violates g. This means that there is a way of restricting the inputs at ℓ, so that ℓ' becomes unreachable from ℓ. Notice that, in the special case where g is a formula over X, (6) simplifies to $C(\ell) := C(\ell) \wedge \neg g$.

If, during the strengthening process, the contract of a location becomes unsatisfiable, this location is marked as illegal. The process is repeated until no more strengthening is possible, whereupon the algorithm stops. Termination is guaranteed because each location has a finite number of successor locations, therefore, can only be strengthened a finite number of times. If, when the algorithm stops, the initial location ℓ_0 has been marked illegal, then the interface is not well-formed. Otherwise, the modified automaton specifies a well-formed interface, which is a witness for the original interface.

For the above class of interfaces there is also an algorithm to check equality, i.e., given two interfaces I_1, I_2, check whether $I_1 = I_2$. Let $M_i = (X, Y, L_i, \ell_{0,i}, C_i, T_i)$ be finite-state automata representing I_i, for $i = 1, 2$, respectively. We first build a synchronous product $M := (X, Y, L_1 \times L_2 \cup \{\ell_{bad}\}, (\ell_{0,1}, \ell_{0,2}), C, T)$, where $C(\ell_1, \ell_2) := C_1(\ell_1) \vee C_2(\ell_2)$ for all $(\ell_1, \ell_2) \in L_1 \times L_2$, $C(\ell_{bad}) := \mathsf{false}$, and:

$$\begin{aligned} T \;:=\; & \{((\ell_1,\ell_2),(C_1(\ell_1) \equiv C_2(\ell_2)) \wedge g_1 \wedge g_2,(\ell_1',\ell_2')) \mid (\ell_i,g_i,\ell_i') \in T_i,\ \text{for } i=1,2\} \\ & \cup \{((\ell_1,\ell_2),C_1(\ell_1) \not\equiv C_2(\ell_2),\ell_{bad})\} \end{aligned} \tag{7}$$

It can be checked that $I_1 = I_2$ iff location ℓ_{bad} is unreachable.

6. COMPOSITION

We define two types of composition: by *connection* and by *feedback*.

6.1 Composition by connection

First, we can compose two interfaces I_1 and I_2 "in sequence", by connecting some of the output variables of I_1 to some of the input variables of I_2. One output can be connected to many inputs, but an input can be connected to at most one output. Parallel composition is a special case of composition by connection, where the connection is empty. The connections define a new interface. Thus, the composition process can be repeated to yield arbitrary (for the moment, acyclic) interface diagrams. Composition by connection is associative (Theorem 3), so the order in which interfaces are composed does not matter.

Two interfaces $I = (X, Y, f)$ and $I' = (X', Y', f')$ are called *disjoint* if they have disjoint sets of input and output variables: $(X \cup Y) \cap (X' \cup Y') = \emptyset$.

Definition 9 Composition by connection. *Let $I_i = (X_i, Y_i, f_i)$, for $i = 1, 2$, be two disjoint interfaces. A* connection θ between I_1, I_2, *is a finite set of pairs of variables, $\theta = \{(y_i, x_i) \mid i = 1, ..., m\}$, such that: (1) $\forall (y, x) \in \theta : y \in Y_1 \wedge x \in X_2$, and (2) there do not exist $(y, x), (y', x) \in \theta$ such that y and y' are distinct. Define:*

$$\mathsf{InVars}(\theta) \;:=\; \{x \mid \exists y : (y,x) \in \theta\} \tag{8}$$

$$X_{\theta(I_1,I_2)} \;:=\; (X_1 \cup X_2) \setminus \mathsf{InVars}(\theta) \tag{9}$$

$$Y_{\theta(I_1,I_2)} \;:=\; Y_1 \cup Y_2 \cup \mathsf{InVars}(\theta) \tag{10}$$

The connection θ defines the composite interface $\theta(I_1, I_2) := (X_{\theta(I_1,I_2)}, Y_{\theta(I_1,I_2)}, f)$, *where, for every $s \in \mathcal{A}(X_{\theta(I_1,I_2)} \cup Y_{\theta(I_1,I_2)})^*$:*

$$\begin{aligned} f(s) \;&:=\; f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta \wedge \forall Y_{\theta(I_1,I_2)} : \Phi \\ \Phi \;&:=\; (f_1(s_1) \wedge \rho_\theta) \to \mathsf{in}(f_2(s_2)) \\ \rho_\theta \;&:=\; \bigwedge_{(y,x)\in\theta} y = x \end{aligned} \tag{11}$$

and, for $i = 1, 2$, s_i is defined to be the projection of s to variables in $X_i \cup Y_i$.

Note that $X_{\theta(I_1,I_2)} \cup Y_{\theta(I_1,I_2)} = X_1 \cup Y_1 \cup X_2 \cup Y_2$. Also notice that $\mathsf{InVars}(\theta) \subseteq X_2$. This implies that $X_1 \subseteq X_{\theta(I_1,I_2)}$, that is, every input variable of I_1 is also an input variable of $\theta(I_1, I_2)$. Also note that $\forall Y_{\theta(I_1,I_2)} : \Phi$ is equivalent to $\forall Y_1 \cup \mathsf{InVars}(\theta) : \Phi$ because Φ does not contain any Y_2 variables. The term $\forall Y_{\theta(I_1,I_2)} : \Phi$ is a condition on $X_{\theta(I_1,I_2)}$, the free inputs of the composite interface. This term states that, no matter which outputs I_1 chooses to produce for a given input, all such outputs are legal inputs for I_2. This condition is essential for preservation of compatibility by refinement as discussed in Section 2.2, and more generally, for preservation of refinement by composition (Theorem 12).

EXAMPLE 4. *We repeat the example in Section 2.2 while being more pedantic. Let Div be the interface defined in Section 2.1 and let D be the interface $D := (\{x_3\}, \{y_2\}, \mathsf{true})$. Let $\theta := \{(y_2, x_2)\}$. Then the term $\forall Y_{\theta(D,Div)} : \Phi$ instantiates to $\forall y_2, y : (\mathsf{true} \wedge y_2 = x_2) \rightarrow x_2 \neq 0$, or equivalently, $\forall y_2 : y_2 \neq 0$, which is* false*, meaning that D and Div are "incompatible". This notion is formalized next.*

Contrary to other works [de Alfaro and Henzinger 2001a; 2001b; Doyen et al. 2008], we do not impose an a-priori *compatibility* condition on connections. Not doing so allows us to state more general results (Theorem 12). Having said that, compatibility is a useful concept therefore we define it explicitly.

DEFINITION 10 COMPATIBILITY. *Let I_1, I_2 be two disjoint interfaces and θ a connection between them. I_1, I_2 are said to be* compatible *with respect to θ iff $\theta(I_1, I_2)$ is well-formable.*

For finite-state interfaces, connection is computable. Let $M_i = (X_i, Y_i, L_i, \ell_{0,i}, C_i, T_i)$ be finite-state automata representing I_i, for $i = 1, 2$, respectively. The composite interface $\theta(I_1, I_2)$ can be represented as $M := (X_{\theta(I_1,I_2)}, Y_{\theta(I_1,I_2)}, L_1 \times L_2, (\ell_{0,1}, \ell_{0,2}), C, T)$, where $C(\ell_1, \ell_2)$ is defined as $f(s)$ is defined in (11), replacing $f_i(\ell_i)$ by $C_i(\ell_i)$, and T is defined as follows:

$$T := \{((\ell_1, \ell_2), g_1 \wedge g_2, (\ell_1', \ell_2')) \mid (\ell_i, g_i, \ell_i') \in T_i, \text{ for } i = 1, 2\} \tag{12}$$

That is, M is essentially a synchronous product of M_1, M_2.

Checking compatibility of two finite-state interfaces can be effectively done by first computing an automaton representing the composite interface $\theta(I_1, I_2)$ and then checking well-formability of the latter, using the algorithms described earlier.

A connection θ is allowed to be empty. In that case, $\rho_\theta \equiv \mathsf{true}$, and the composition can be viewed as the *parallel composition* of two interfaces. If θ is empty, we write $I_1 \| I_2$ instead of $\theta(I_1, I_2)$. As may be expected, the contract of the parallel composition at a given global state is the conjunction of the original contracts at the corresponding local states, which implies that parallel composition is commutative:

LEMMA 2. *Consider two disjoint interfaces, $I_i = (X_i, Y_i, f_i)$, $i = 1, 2$. Then $I_1 \| I_2 = (X_1 \cup X_2, Y_1 \cup Y_2, f)$, where f is such that for all $s \in \mathcal{A}(X_1 \cup X_2 \cup Y_1 \cup Y_2)^*$, $f(s) \equiv f_1(s_1) \wedge f_2(s_2)$, where, for $i = 1, 2$, s_i is the projection of s to $X_i \cup Y_i$.*

A corollary of Lemma 2 is Theorem 2:

Theorem 2 Commutativity of parallel composition. *Let I_1 and I_2 be two disjoint interfaces. Then:*

$$I_1 \| I_2 = I_2 \| I_1.$$

Theorem 3 Associativity of connection. *Let I_1, I_2, I_3 be pairwise disjoint interfaces. Let θ_{12} be a connection between I_1, I_2, θ_{13} a connection between I_1, I_3, and θ_{23} a connection between I_2, I_3. Then:*

$$(\theta_{12} \cup \theta_{13})\,(I_1, \theta_{23}(I_2, I_3)) = (\theta_{13} \cup \theta_{23})\,(\theta_{12}(I_1, I_2), I_3)\,.$$

Example 5. *Consider the diagram of stateless interfaces shown in Figure 4, where:*

$$\begin{aligned} I_{id} &:= (\{x_1\}, \{y_1\}, y_1 = x_1) \\ I_{+1,2} &:= (\{x_2\}, \{y_2\}, x_2 + 1 \leq y_2 \leq x_2 + 2) \\ I_{\leq} &:= (\{z_1, z_2\}, \{\}, z_1 \leq z_2) \end{aligned}$$

This diagram can be modeled as any of the two following equivalent compositions:

$$\theta_2\big(I_{+1,2}, \theta_1(I_{id}, I_{\leq})\big) = (\theta_1 \cup \theta_2)\big((I_{id} \| I_{+1,2}), I_{\leq}\big)$$

where $\theta_1 := \{(y_1, z_1)\}$ and $\theta_2 := \{(y_2, z_2)\}$.

We proceed to compute the contract of the interface defined by the diagram. It is easier to consider the composition $(\theta_1 \cup \theta_2)((I_{id} \| I_{+1,2}), I_{\leq})$. Define $\theta_3 := \theta_1 \cup \theta_2$. From Lemma 2 we get:

$$I_{id} \| I_{+1,2} = (\{x_1, x_2\}, \{y_1, y_2\}, y_1 = x_1 \wedge x_2 + 1 \leq y_2 \leq x_2 + 2)$$

Then, for $\theta_3((I_{id} \| I_{+1,2}), I_{\leq})$, Formula (11) gives:

$$\Phi := (y_1 = x_1 \wedge x_2 + 1 \leq y_2 \leq x_2 + 2 \wedge y_1 = z_1 \wedge y_2 = z_2) \rightarrow z_1 \leq z_2$$

By quantifier elimination, we get

$$\forall y_1, y_2, z_1, z_2 : \Phi \;\equiv\; x_1 \leq x_2 + 1$$

therefore

$$\begin{aligned} \theta_3((I_{id} \| I_{+1}), I_{\leq}) \;=\; & (\{x_1, x_2\}, \{y_1, y_2, z_1, z_2\}, \\ & y_1 = x_1 \wedge x_2 + 1 \leq y_2 \leq x_2 + 2 \wedge z_1 \leq z_2 \\ & \wedge y_1 = z_1 \wedge y_2 = z_2 \wedge x_1 \leq x_2 + 1). \end{aligned}$$

Notice that $\mathsf{in}(\theta_3((I_{id} \| I_{+1}), I_{\leq})) \equiv x_1 \leq x_2 + 1$. That is, because of the connection θ, new assumptions have been generated for the external inputs x_1, x_2. These assumptions are stronger than those generated by simple composition of relations, which are $x_1 \leq x_2 + 2$ in this case.

A composite interface is not guaranteed to be well-formed, neither well-formable, even if all its components are well-formed:

Example 6. *Consider the composite interface $\theta_3((I_{id} \| I_{+1,2}), I_{\leq})$ from Example 5, and suppose we connect its open inputs x_1, x_2 to outputs v_1, v_2, respectively, of some other interface that guarantees $v_1 > v_2 + 1$. Clearly, the result is* false, *since the constraint $x_1 > x_2 + 1 \wedge x_1 \leq x_2 + 1$ is unsatisfiable.*

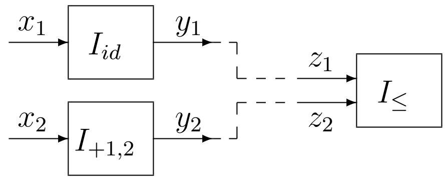

Fig. 4. The interface diagram of Example 5.

6.2 Composition by feedback

Our second type of composition is *feedback composition*, where an output variable of an interface I is connected to one of its input variables x. For feedback, I is required to be *Moore with respect to* x. The term "Moore interfaces" has been introduced in [Chakrabarti et al. 2002]. Our definition is similar in spirit, but less restrictive than the one in [Chakrabarti et al. 2002]. Both definitions are inspired by Moore machines, where the outputs are determined by the current state alone and do not depend directly on the input. In our version, an interface is Moore with respect to a given input variable x, meaning that the contract may depend on the current state as well as on input variables other than x. This allows to connect an output to x to form a feedback loop without creating causality cycles.

DEFINITION 11 MOORE INTERFACES. *An interface* $I = (X, Y, f)$ *is called* Moore with respect to $x \in X$ *iff for all* $s \in f$, $f(s)$ *is a property over* $(X \cup Y) \setminus \{x\}$. *I is called simply* Moore *when it is Moore with respect to every* $x \in X$.

Note that a source interface is by definition Moore, since it has no input variables. Note also that although the contract of a Moore interface should not depend on the *current* value of an input variable, it may very well depend on *past* values of such a variable, which influence the state s. An example where this occurs is the unit delay:

EXAMPLE 7 UNIT DELAY. *A* unit delay *is a basic building block in many modeling languages (including Simulink and SCADE). Its specification is roughly: "output at time* k *the value of the input at time* $k-1$*; at time* $k = 0$ *(initial time), output some initial value* v_0*". We can capture this specification as an interface:*

$$I_{ud} := (\{x\}, \{y\}, f_{ud}),$$

where f_{ud} *is defined as follows:*

$$\begin{aligned} f_{ud}(\varepsilon) &:= (y = v_0) \\ f_{ud}(s \cdot a) &:= (y = a(x)) \end{aligned}$$

That is, initially the contract guarantees $y = v_0$*. Then, when the state is some sequence* $s \cdot a$*, the contract guarantees* $y = a(x)$*, where* $a(x)$ *is the last value assigned to input* x*.* I_{ud} *is Moore (with respect to its unique input variable) since all its contracts are properties over* y *only.*

DEFINITION 12 COMPOSITION BY FEEDBACK. *Let* $I = (X, Y, f)$ *be an interface. A* feedback connection κ on I *is a pair* $(y, x) \in Y \times X$*.* κ *is* valid *if I is Moore* *with respect to x. Define* $\rho_\kappa := (x = y)$. *A valid feedback connection* κ *defines the interface:*

$$\kappa(I) := (X \setminus \{x\}, Y \cup \{x\}, f_\kappa) \tag{13}$$

$$f_\kappa(s) := f(s) \wedge \rho_\kappa, \quad \textit{for all } s \in \mathcal{A}(X \cup Y)^* \tag{14}$$

In the sequel, when we talk about feedback connections we implicitly assume they are valid.

For finite-state interfaces, feedback is computable. Let $M = (X, Y, L, \ell_0, C, T)$ be a finite-state automaton representing I. First, to check whether M represents a Moore interface w.r.t. a given input variable $x \in X$, it suffices to make sure that for every location $\ell \in L$, $C(\ell)$ does not refer to x. Then, if $\kappa = (y, x)$, the interface $\kappa(I)$ can be represented as $M' := (X \setminus \{x\}, Y \cup \{x\}, L, \ell_0, C', T)$, where $C'(\ell) := C(\ell) \wedge x = y$, for all $\ell \in L$.

Theorem 4 Commutativity of feedback. *Let* $I = (X, Y, f)$ *be Moore with respect to both* $x_1, x_2 \in X$, *where* $x_1 \neq x_2$. *Let* $\kappa_1 = (y_1, x_1)$ *and* $\kappa_2 = (y_2, x_2)$ *be feedback connections. Then*

$$\kappa_1(\kappa_2(I)) = \kappa_2(\kappa_1(I)).$$

Let K be a set of feedback connections, $K = \{\kappa_1, ..., \kappa_n\}$, such that $\kappa_i = (y_i, x_i)$, and all x_i are pairwise distinct, for $i = 1, ..., n$. Let I be an interface that is Moore with respect to all $x_1, ..., x_n$. We denote by $K(I)$ the interface $\kappa_1(\kappa_2(\cdots \kappa_n(I) \cdots))$. By commutativity of feedback composition, the resulting interface is independent from the order of application of feedback connections. We will use the notation $\mathsf{InVars}(K) := \{x_i \mid (y_i, x_i) \in K\}$, for the set of input variables connected in K.

Theorem 5 Commutativity between connection and feedback. *Let* I_1, I_2 *be disjoint interfaces and let* θ *be a connection between* I_1, I_2. *Let* κ_1, κ_2 *be valid feedback connections on* I_1, I_2, *respectively. Suppose that* $\mathsf{InVars}(\kappa_2) \cap \mathsf{InVars}(\theta) = \emptyset$. *Then:*

$$\kappa_1(\theta(I_1, I_2) = \theta(\kappa_1(I_1), I_2) \qquad \textit{and} \qquad \kappa_2(\theta(I_1, I_2) = \theta(I_1, \kappa_2(I_2)).$$

Theorem 6 Preservation of Mooreness by connection. *Let* I_1, I_2 *be disjoint interfaces such that* $I_i = (X_i, Y_i, f_i)$, *for* $i = 1, 2$. *Let* θ *be a connection between* I_1, I_2.

(1) *If* I_1 *is Moore w.r.t.* $x_1 \in X_1$ *then* $\theta(I_1, I_2)$ *is Moore w.r.t.* x_1.
(2) *If* I_1 *is Moore and* $\mathsf{InVars}(\theta) = X_2$ *then* $\theta(I_1, I_2)$ *is Moore.*
(3) *If* I_2 *is Moore w.r.t.* $x_2 \in X_2$ *and* $x_2 \notin \mathsf{InVars}(\theta)$, *then* $\theta(I_1, I_2)$ *is Moore w.r.t.* x_2.

An interesting question is to what extent and how to transform a given *diagram* of interfaces, such as the one shown in Figure 5, to a valid expression of interface compositions. This cannot be done for arbitrary diagrams, due to restrictions on feedback, but it can be done for diagrams that satisfy the following condition: every dependency cycle in the diagram, formed by block connections, must visit at least one input variable x of some interface I, such that I is Moore w.r.t. x. If this condition holds, then we say that the diagram is *causal*. For example, the diagram in Figure 5 is causal iff I_1 is Moore w.r.t. x_2 or I_2 is Moore w.r.t. x_4.

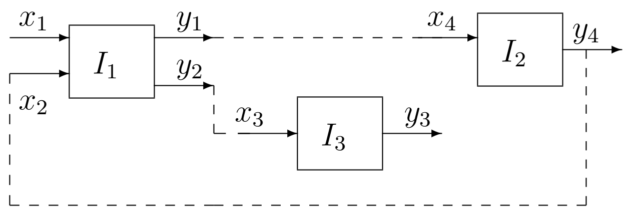

Fig. 5. An interface diagram with feedback.

We can systematically transform causal interface diagrams into expressions of interface compositions as follows. First, we remove from the diagram any *Moore connections*. A connection from output variable y to input variable x is a Moore connection if the interface I where x belongs to is Moore w.r.t. x. Because the original diagram is by hypothesis causal, the diagram obtained after removing Moore connections is guaranteed to have no dependency cycles. This acyclic diagram can be easily transformed into an expression involving only interface compositions by connection. By associativity of connection (Theorem 3), the order in which these connections are applied does not matter. Call the resulting interface I_c. Then, the removed Moore connections can be turned into feedback compositions, and applied to I_c. Because Mooreness is preserved by connection (Theorem 6), I_c is guaranteed to be Moore w.r.t. any input variable x that is the destination of a Moore connection. Therefore, the above feedback compositions are valid for I_c. Moreover, because of commutativity of feedback (Theorem 4), the resulting interface is again uniquely defined.

Example 8. *Consider the diagram of interfaces shown in Figure 5. Suppose that I_1 is Moore with respect to x_2. Then, the diagram can be expressed as any of the two compositions*

$$\kappa\Big(\theta_1\big(I_1, (I_2 \| I_3)\big)\Big) = \theta_3\Big(\kappa\big(\theta_2(I_1, I_2)\big), I_3\Big)$$

where $\theta_1 := \{(y_1, x_4), (y_2, x_3)\}$, $\theta_2 := \{(y_1, x_4)\}$, $\theta_3 := \{(y_2, x_3)\}$, and $\kappa := (y_4, x_2)$. The two expressions are equivalent, since, by Theorem 5, $\theta_3\Big(\kappa\big(\theta_2(I_1, I_2)\big), I_3\Big) = \kappa\Big(\theta_3\big(\theta_2(I_1, I_2), I_3\big)\Big)$, and by Theorem 3, $\theta_3\big(\theta_2(I_1, I_2), I_3\big) = \theta_1\big(I_1, (I_2 \| I_3)\big)$.

Lemma 3. *Let $I = (X, Y, f)$ be a Moore interface with respect to $x \in X$. Let $\kappa = (y, x)$ be a feedback connection on I. Let $\kappa(I) = (X \setminus \{x\}, Y \cup \{y\}, f_\kappa)$. Then:*

(1) $f_\kappa \subseteq f$.
(2) For any $s \in f_\kappa$, $\mathsf{in}(f_\kappa(s)) \equiv \mathsf{in}(f(s))$.

Theorem 7 Feedback preserves well-formedness. *Let I be a Moore interface with respect to some of its input variables, and let κ be a valid feedback connection on I. If I is well-formed then $\kappa(I)$ is well-formed.*

Feedback does not preserve well-formability:

Example 9. *Consider a finite-state interface I_f with two states, s_0 (the initial state) and s_1, one input variable x and one output variable y. I_f remains at state s_0*

when $x \neq 0$ *and moves from* s_0 *to* s_1 *when* $x = 0$*. Let* $\phi_0 := y = 0$ *be the contract at state* s_0 *and let* $\phi_1 :=$ false *be the contract at state* s_1*.* I_f *is not well-formed because* ϕ_1 *is unsatisfiable while state* s_1 *is reachable.* I_f *is well-formable, however: it suffices to restrict* ϕ_0 *to* $\phi'_0 := y = 0 \wedge x \neq 0$*. Denote the resulting (well-formed) interface by* I'_f*. Note that* I_f *is Moore with respect to* x*, whereas* I'_f *is not. Let* κ *be the feedback connection* (y, x)*. Because* I_f *is Moore,* $\kappa(I_f)$ *is defined, and is such that its contract at state* s_0 *is* $y = 0 \wedge x = y$*, and its contract at state* s_1 *is* false $\wedge\, x = y \equiv$ false*.* $\kappa(I_f)$ *is not well-formable: indeed,* $y = 0 \wedge x = y$ *implies* $x = 0$*, therefore, state* s_1 *cannot be avoided.*

7. HIDING

As can be seen in Example 5, composition often creates redundant output variables, in the sense that some of these variables are equal to each other. This happens because input variables that get connected become output variables. To remove redundant output variables, we propose a *hiding* operator. Hiding may also be used to remove other output variables that may not be redundant, provided they do not influence the evolution of contracts, as we shall see below.

For a stateless interface $I = (X, Y, \phi)$, the (stateless) interface resulting from hiding an output variable $y \in Y$ can simply be defined as:

$$\mathsf{hide}(y, I) := (X, Y \setminus \{y\}, \exists y : \phi)$$

This definition does not directly extend to the general case of stateful interfaces, however. The reason is that the contract of a stateful interface I may depend on the history of y. Then, hiding y is problematic because it results in the environment not being able to uniquely determine the contract based on the history of observations. This results in particular in refinement not being preserved by hiding, as we show later in Example 15. To avoid these problems, we allow hiding only for those outputs which do not influence the evolution of the contract.

Given $s, s' \in \mathcal{A}(X \cup Y)^*$ such that $|s| = |s'|$ (i.e., s, s' have same length), and given $Z \subseteq X \cup Y$, we say that s and s' *agree on* Z, denoted $s =_Z s'$, when for all $i \in \{1, ..., |s|\}$, and all $z \in Z$, $s_i(z) = s'_i(z)$. Given interface $I = (X, Y, f)$, we say that f *is independent from* z if for every $s, s' \in f$, $s =_{(X \cup Y) \setminus \{z\}} s'$ implies $f(s) = f(s')$. That is, the evolution of z does not affect the evolution of f.

Notice that f being independent from z does *not* imply that f cannot refer to variable z. Indeed, all stateless interfaces trivially satisfy the independence condition: their contracts are invariant in time, i.e., they do not depend on the evolution of variables. Clearly, the contract of a stateless interface can refer to any of its variables. Conversely, even if the contracts specified by f do not refer to z, f may still depend on z, because the evolution of contracts may depend on z. For example, suppose that $f(\varepsilon) \equiv$ true, and that $f(z = 0)$ is different from $f(z = 1)$, although no contract refers to z. Here, $f(z = k)$ denotes the contract at a state where $z = k$. In this case, f depends on z since the value z assumes at the first round determines the contract to be used in the second round.

The above notion of independence is weaker than redundancy in variables, as we show next. First, we formalize redundancy in variables. Given $z \in X \cup Y$, we say that z *is redundant in* f if there exists $z' \in X \cup Y$ such that $z' \neq z$, and for all $s \in f$, for all $i \in \{1, ..., |s|\}$, $s_i(z) = s_i(z')$. It should be clear that all outputs in

$\mathsf{InVars}(\theta)$ in an interface obtained by connection θ are redundant (see Definition 9). Similarly, in an interface obtained by feedback $\kappa = (y, x)$, newly introduced output variable x is redundant (see Definition 12).

LEMMA 4. *If z is redundant in f then f is independent from z.*

When f is independent from z, f can be viewed as a function from $\mathcal{A}((X \cup Y) \setminus \{z\})^*$ to $\mathcal{F}(X \cup Y)$ instead of a function from $\mathcal{A}(X \cup Y)^*$ to $\mathcal{F}(X \cup Y)$. We use this when we write $f(s)$ for $s \in \mathcal{A}((X \cup Y) \setminus \{z\})^*$ in the following definition:

DEFINITION 13 HIDING. *Let $I = (X, Y, f)$ be an interface and let $y \in Y$, such that f is independent from y. Then $\mathsf{hide}(y, I)$ is defined to be the interface*

$$\mathsf{hide}(y, I) := (X, Y \setminus \{y\}, f') \tag{15}$$

such that for any $s \in \mathcal{A}(X \cup Y \setminus \{y\})^*$, $f'(s) := \exists y : f(s)$.

For finite-state interfaces, hiding is computable. Let $M = (X, Y, L, \ell_0, C, T)$ be a finite-state automaton representing I. We first need to ensure that the contract of I is independent from y. A simple way to do this is to check that no guard of M refers to y. This condition is sufficient, but not necessary. Consider, for example, two complementary guards $y < 1$ and $y \geq 1$ whose transitions lead to locations with identical contracts. Then the two locations may be merged to a single one, and the two transitions to a single transition with guard true. Another situation where the above condition may be too strict is when a guard refers to y but y is redundant. In that case, all occurrences of y in guards of M can be replaced by its equal variable y'. Once independence from y is ensured, $\mathsf{hide}(y, I)$ can be represented as $M' := (X, Y \setminus \{y\}, L, \ell_0, C', T)$, where $C'(\ell) := \exists y : C(\ell)$, for all $\ell \in L$.

8. ENVIRONMENTS, PLUGGABILITY AND SUBSTITUTABILITY

We wish to formalize the notion of interface contexts and substitutability, and we introduce *environments* for that purpose. Environments are interfaces. An interface I can be connected to an environment E to form a closed-loop system, as illustrated in Figure 6. E acts both as a *controller* and an *observer* for I. It is a controller in the sense that it "steers" I by providing inputs to it, depending on the outputs it receives. At the same time, E acts as an observer, that monitors the inputs consumed and outputs produced by I, and checks whether a given property is satisfied. These notions are formalized in Definition 14 that follows.

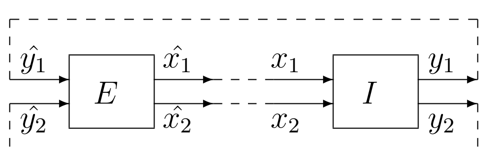

Fig. 6. Illustration of pluggability.

Before giving the definition, however, a remark is in order. Interfaces and environments are to be connected in a closed-loop, as illustrated in Figure 6. In order to do this in our setting, every dependency cycle must be "broken" by a Moore connection, as prescribed by the transformation of interface diagrams to composition expressions, given in Section 6.2. It can be seen that, in the case of two interfaces connected in closed-loop, the above requirement implies that one of the two interfaces is Moore. For instance, consider Figure 6. If I is not Moore w.r.t. x_2, then E must be Moore w.r.t. to both $\hat{y_1}$ and $\hat{y_2}$, so that both feedback connections can be formed. Similarly, if E is not Moore w.r.t. $\hat{y_2}$, say, then I must be Moore w.r.t. both x_1, x_2. This remark justifies the definition below:

Definition 14 Environments and pluggability. *Consider interfaces $I = (X, Y, f)$ and $E = (\hat{Y}, \hat{X}, f_e)$. E is said to be an* environment *for I if there exist bijections between X and $\hat{X}$, and between Y and $\hat{Y}$. $\hat{X}$ are called the* mirror variables *of X, and similarly for $\hat{Y}$ and Y. For $x \in X$, we denote by $\hat{x}$ the corresponding (by the bijection) variable in $\hat{X}$, and similarly with y and $\hat{y}$. I is said to be* pluggable *to E, denoted $I \leftrightarrows E$, iff the following conditions hold:*

—*I is Moore or E is Moore.*
—*If E is Moore then the interface $K(\theta(E, I))$ is well-formed, where $\theta := \{(\hat{x}, x) \mid x \in X\}$ and $K := \{(y, \hat{y}) \mid y \in Y\}$. Notice that, because E is Moore and $\mathsf{InVars}(\theta) = X$, part 2 of Theorem 6 applies, and guarantees that $\theta(E, I)$ is Moore. Therefore, $K(\theta(E, I))$ is well-defined.*
—*If I is Moore then the interface $K(\theta(I, E))$ is well-formed, where $\theta := \{(y, \hat{y}) \mid y \in Y\}$ and $K := \{(\hat{x}, x) \mid x \in X\}$.*

Note that, by definition, I is pluggable to E iff E is pluggable to I.

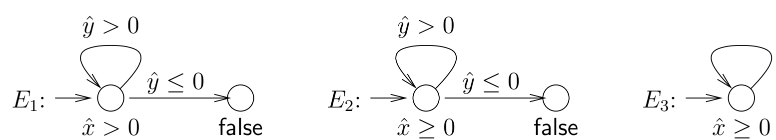

Fig. 7. Three environments.

Example 10. *Consider interfaces I_1 and I_2 from Example 1 and environments E_1, E_2, E_3 of Figure 7 (implicitly, transitions without guards are assumed to have guard* true*). It can be checked that both I_1 and I_2 are pluggable to E_1. I_1 is not pluggable to neither E_2 nor E_3: indeed, the output guarantee $\hat{x} \geq 0$ of these two environments is not strong enough to meet the input assumption $x > 0$ of I_1. I_2 is not pluggable to E_2: although the input assumption of I_2 is* true*, I_2 guarantees $y > 0$ only when $x > 0$. Therefore the guard $\hat{y} \leq 0$ of E_2 is enabled in some cases, leading to location with contract* false*, which means that the closed-loop interface is not well-formed. On the other hand, I_2 is pluggable to E_3.*

Theorem 8 Pluggability and well-formability.

—*If an interface I is well-formable then there exists an environment E for I such that $I \leftrightarrows E$.*

—*If there exists an environment E for interface I such that $I \leftrightarrows E$ and I is not Moore then I is well-formable.*

EXAMPLE 11. *Consider interfaces I and E shown in Figure 8. Observe that I is Moore and $I \leftrightarrows E$. However, I is not well-formable.*

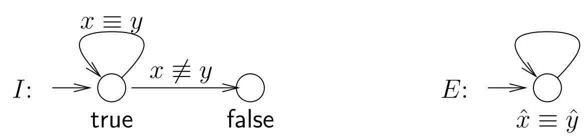

Fig. 8. A Moore interface I and a non-Moore environment E.

Example 11 shows that the non-Mooreness assumption on I is indeed necessary in part 2 of Theorem 8. This example also illustrates an aspect of our definition of well-formability, which may appear inappropriate for Moore interfaces: indeed, interface I of Figure 8 is non-well-formable, yet there is clearly an environment that can be plugged to I so that false location is avoided. An alternative definition of well-formability for an interface I would have been existence of an environment that can be plugged to I. This would make Theorem 8 a tautology. Nevertheless, we opt for Definition 8, which allows to transform interfaces into a "canonical form" where all contracts are satisfiable.

DEFINITION 15 SUBSTITUTABILITY. *We say that* interface I' may replace interface I *(or I'* may be substituted for *I), denoted $I \rightarrow_e I'$, iff for any environment E, if I is pluggable to E then I' is pluggable to E. We say that I and I' are* mutually substitutable*, denoted $I \equiv_e I'$, iff both $I \rightarrow_e I'$ and $I' \rightarrow_e I$ hold.*

As we shall show in Theorem 16, for well-formed interfaces, mutual substitutability coincides with interface equality.

9. REFINEMENT

DEFINITION 16 REFINEMENT. *Consider two interfaces $I = (X, Y, f)$ and $I' = (X', Y', f')$. We say that I'* refines *I, written $I' \sqsubseteq I$, iff $X' = X$, $Y' = Y$, and for any $s \in f \cap f'$, the following formula is valid:*

$$\mathsf{in}(f(s)) \rightarrow \Big(\mathsf{in}(f'(s)) \wedge \big(f'(s) \rightarrow f(s)\big)\Big) \tag{16}$$

Condition 16 can be rewritten equivalently as the conjunction of the following two conditions:

$$\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f'(s)) \tag{17}$$

$$\big(\mathsf{in}(f(s)) \wedge f'(s)\big) \rightarrow f(s) \tag{18}$$

Condition (17) states that every input assignment that is legal in I is also legal in I'. This guarantees that, for any possible input assignment that can be provided to I by a context C, if this assignment is accepted by I then it is also accepted by I'. Condition (18) states that, for every input assignment that is legal in I, all output assignments that can be possibly produced by I' from that input, can also be produced by I. This guarantees that if C accepts the assignments produced by I then it also accepts those produced by I'.

It should be noted that the refinement conditions are required only for states that belong in both f and f'. The intuition for this choice is as follows. The initial state is ε and by definition $\varepsilon \in f \cap f'$. At this state, we only wish to consider legal inputs for I, that is, $a_X \in \mathsf{in}(f(\varepsilon))$. Otherwise, I' is free to behave as it wishes since the behavior is not possible in I. Condition (17) then implies that $a_X \in \mathsf{in}(f'(\varepsilon))$. Next, we wish to consider only outputs that I' may produce given a_X, that is, a_Y such that $(a_X, a_Y) \in f'(\varepsilon)$. Otherwise, I is free to behave as it wishes, since the behavior is not possible in I'. Condition (18) then implies that $(a_X, a_Y) \in f(\varepsilon)$. Therefore, $(a_X, a_Y) \in f \cap f'$, that is, the requirements should be applied only to states of length one that belong in both f and f'. Reasoning inductively, the same can be derived for states of arbitrary length.

A remark is in order regarding the constraint $X' = X$ and $Y' = Y$ imposed during refinement. This constraint may appear as too strict, but we argue that it is not. To begin, recall that $I' \sqsubseteq I$ should imply that I' can replace I in any context. In our setting, contexts are formalized as environments. Consider such an environment with controller C. C provides values to the input variables of I, and requires values from the output variables of I. Suppose I' has an input variable x that I does not have, that is, there exists $x \in X' \setminus X$. In general, C may not provide x. In that case, I' cannot replace I, because by doing so, input x would remain free. Therefore, $X' \subseteq X$ must hold. Similarly, suppose that there exists $y \in Y \setminus Y'$. In general, C may require y, that is, y may be a free input for C. In that case, I' cannot replace I, because by doing so, y would remain free. Therefore, $Y \subseteq Y'$ must hold.

Now, suppose that X' is a strict subset of X or Y' is a strict superset of Y (or both). Then, we can easily modify I and I' as follows: we add to X' all the input variables missing from I', so that $X' = X$, and we add to Y all the output variables missing from I, so that $Y = Y'$. While doing so, we do not change the contracts of either I or I': the contracts simply ignore the additional variables, that is, do not impose any constraints on their values. It can be seen that this transformation preserves the validity of refinement Condition 16. Indeed, $\mathsf{in}(\phi) \rightarrow (\mathsf{in}(\phi') \wedge (\phi' \rightarrow \phi))$ holds when ϕ is over $X \cup Y$ and ϕ' is over $X' \cup Y'$ iff it holds when both ϕ and ϕ' are taken to be over $X \cup Y'$, provided $X' \subseteq X$ and $Y' \supseteq Y$. Therefore, without loss of generality, we require $X = X'$ and $Y = Y'$.

EXAMPLE 12 BUFFER INTERFACE REFINEMENTS. *This example builds on Example 2. Consider Figure 9. It depicts a variant of the single-place buffer interface, where the buffer may fail to complete a read or write operation. This interface has one more boolean output variable, namely,* ack*, in addition to those of Example 2, and two more locations,* after_read *and* after_write*. Its global contract is identical to that of Example 2. So are local contracts at locations ℓ_0 and ℓ_1. After a write operation, the interface moves to location* after_write*, where it non-deterministically chooses to set* ack *to* true *or* false*: setting it to* true *means the write was successful,* false *means the write failed. The meaning is symmetric for read. This particular interface does not allow read or write operations in the two intermediate locations.*

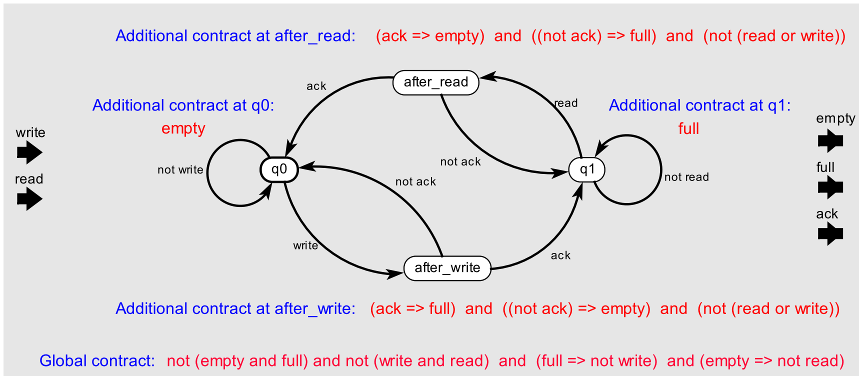

Fig. 9. Interface for a buffer of size 1 that may fail to do a read or write.

It is natural to expect that a buffer that never fails can replace a buffer that may fail. We would like to have a formal guarantee of this, in terms of refinement of their corresponding interfaces. That is, we would like the interface of Figure 2 to refine the one of Figure 9. This does not immediately hold, since ack *is not a variable of the former. We can easily add it however, obtaining the interface shown in Figure 10. This buffer never fails, therefore,* ack *is always* true. *With this modification, the interface of Figure 10 refines the one of Figure 9. On the other hand, the converse is not true: the interface of Figure 9 does not refine the one of Figure 10, because in the latter output* ack *is always* true, *whereas in the former in can also be* false.

With respect to the above discussion on the $X = X'$ and $Y = Y'$ requirements, note that in this example the condition $X' \subseteq X$ and $Y \subseteq Y'$ does not hold: indeed, Y (the outputs of Figure 9) includes ack *whereas Y' (the outputs of Figure 2) does not. For this reason,* ack *is not simply a "dummy" variable in this case, and we need to specify a contract for it, as done in the revised interface of Figure 10.*

This example also illustrates the fact that our notion of refinement is different from language inclusion. For instance, the following sequence:

$$(\text{empty}, \neg\text{full}, \text{ack}, \text{write}, \neg\text{read}) \cdot (\neg\text{empty}, \text{full}, \text{ack}, \neg\text{write}, \text{read})$$

belongs in the language (i.e., contract) of the interface of Figure 10, but not in the language of the interface of Figure 9. This is because the latter does not allow a "read" to happen at state "after_write".

For finite-state interfaces, refinement can be checked as follows. Let $M_i = (X, Y, L_i, \ell_{0,i}, C_i, T_i)$ be finite-state automata representing I_i, for $i = 1, 2$, respectively. We first build a synchronous product

$$M := (X, Y, L_1 \times L_2 \cup \{\ell_{good}, \ell_{bad}\}, (\ell_{0,1}, \ell_{0,2}), C, T),$$

where $C(\ell_1, \ell_2) := \mathsf{in}(C_1(\ell_1))$ for all $(\ell_1, \ell_2) \in L_1 \times L_2$, $C(\ell_{good}) := \mathsf{true}$, $C(\ell_{bad}) :=$

false, and:

$$
\begin{aligned}
T &:= \{((\ell_1,\ell_2), g_{both} \wedge g_1 \wedge g_2, (\ell_1',\ell_2')) \mid (\ell_i, g_i, \ell_i') \in T_i, \text{ for } i=1,2\} \\
&\quad \cup \{((\ell_1,\ell_2), g_{bad}, \ell_{bad}), ((\ell_1,\ell_2), g_{good}, \ell_{good}), (\ell_{good}, \mathsf{true}, \ell_{good})\} \quad (19)\\
g_{both} &:= C_1(\ell_1) \wedge C_2(\ell_2) \quad (20)\\
g_{good} &:= \mathsf{in}(C_1(\ell_1)) \wedge \mathsf{in}(C_2(\ell_2)) \wedge \neg C_2(\ell_2) \quad (21)\\
g_{bad} &:= \mathsf{in}(C_1(\ell_1)) \wedge \Big(\neg \mathsf{in}(C_2(\ell_2)) \vee \big(C_2(\ell_2) \wedge \neg C_1(\ell_1)\big)\Big) \quad (22)
\end{aligned}
$$

Notice that guard g_{bad} encodes the negation of the refinement Condition (16). Also note that $g_{both}, g_{good}, g_{bad}$ are pairwise disjoint, and such that $g_{both} \vee g_{good} \vee g_{bad} \equiv \mathsf{in}(C_1(\ell_1))$, for all $(\ell_1,\ell_2) \in L_1 \times L_2$. This ensures determinism of M. It can be checked that $I_2 \sqsubseteq I_1$ iff location ℓ_{bad} is unreachable.

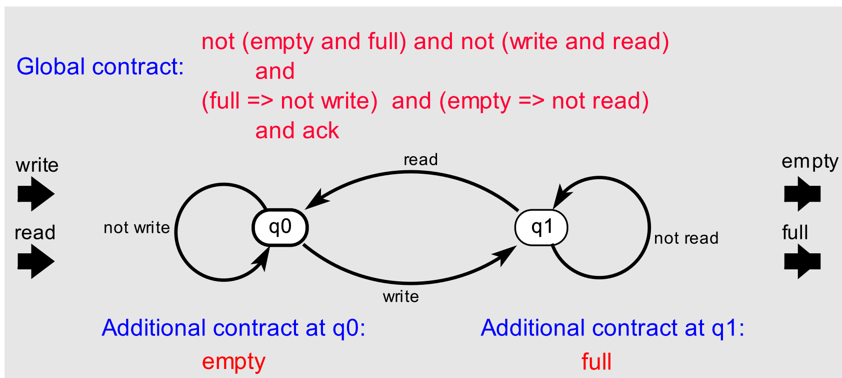

Fig. 10. Buffer interface of Figure 2 with additional output variable *ack*.

9.1 Properties of the refinement relation

We proceed to state the main properties of refinement. First, observe that, perhaps surprisingly, interfaces with false contracts (i.e., $f = \{\varepsilon\}$) are "top" elements with respect to the $\sqsubseteq$ order, that is, they are refined by any interface that has the same input and output variables. This is in accordance with the spirit of refinement as a condition for substitutability. The false interface is not pluggable to any environment, therefore, it can be replaced by any interface.

We next provide a result used in the proof of the theorems that follow.

LEMMA 5. *Let $I = (X, Y, f)$, $I' = (X, Y, f')$, $I'' = (X, Y, f'')$ be interfaces such that $I'' \sqsubseteq I'$ and $I' \sqsubseteq I$. Then $f \cap f'' \subseteq f'$.*

An illustration of the above lemma can be found in the division example in Section 2.3, where $\phi_1 \wedge \phi_4 \equiv \phi_2$.

THEOREM 9 PARTIAL ORDER. *$\sqsubseteq$ is a partial order, that is, a reflexive, antisymmetric and transitive relation.*

Theorem 10. *Let I, I' be stateless interfaces such that $I' \sqsubseteq I$. If I is well-formed then I' is well-formed.*

Theorem 10 does not generally hold for stateful interfaces: the reason is that, because I' may accept more inputs than I, there may be states that are reachable in I' but not in I, and the contract of I' in these states may be unsatisfiable. When this situation does not occur, refinement preserves well-formedness also in the stateful case. Moreover, refinement always preserves well-formability:

Theorem 11 Refinement and well-formedness/-formability. *Let I, I' be interfaces such that $I' \sqsubseteq I$.*

(1) *If I is well-formed and $f(I') \subseteq f(I)$ then I' is well-formed.*

(2) *If I, I' are sources and I is well-formed, then I' is also well-formed.*

(3) *If I is well-formable then I' is well-formable.*

The following lemma is used in the proof of Theorem 12 that follows.

Lemma 6. *Consider two disjoint interfaces I_1 and I_2, and a connection θ between I_1, I_2. Let f_1 and f_2 be the projections of $f(\theta(I_1, I_2))$ to states over the variables of I_1 and I_2, respectively. Then $f_1 \subseteq f(I_1)$ and $f_2 \subseteq f(I_2)$.*

Theorems 12 and 13 that follow state a major property of our theory, namely, that refinement is preserved by composition.

Theorem 12 Connection preserves refinement. *Consider two disjoint interfaces I_1 and I_2, and a connection θ between I_1, I_2. Let I_1', I_2' be interfaces such that $I_1' \sqsubseteq I_1$ and $I_2' \sqsubseteq I_2$. Then, $\theta(I_1', I_2') \sqsubseteq \theta(I_1, I_2)$.*

Notice that Theorem 12 holds independently of whether the connection yields a well-formed interface or not, that is, independently of whether the composed interfaces are compatible. This is a reason why we do not impose compatibility as a condition for composition, as we mentioned earlier. Together with Theorems 10 and 11, Theorem 12 guarantees that if the refined composite interface is well-formed/formable, then so is the refining one. In particular, if I_1 and I_2 are compatible with respect to θ, then so are I_1' and I_2'.

Theorem 13 Feedback preserves refinement. *Let I, I' be interfaces such that $I' \sqsubseteq I$. Suppose both I and I' are Moore interfaces with respect to one of their input variables, x. Let $\kappa = (y, x)$ be a feedback connection. Then $\kappa(I') \sqsubseteq \kappa(I)$.*

Note that the assumption that I' be Moore w.r.t. x in Theorem 13 is essential. Indeed, Mooreness is not generally preserved by refinement:

Example 13. *Consider the stateless interfaces $I_{even} := (\{x\}, \{y\}, y \div 2 = 0)$, where $\div$ denotes the modulo operator, and $I_{\times 2} := (\{x\}, \{y\}, y = 2x)$. I_{even} is Moore. $I_{\times 2}$ is not Moore. Yet $I_{\times 2} \sqsubseteq I_{even}$.*

It is instructive at this point to justify our restrictions regarding feedback composition, by illustrating some of the problems that would arise if we allowed arbitrary feedback:

Example 14. *This example is borrowed from [Doyen et al. 2008]. Suppose* I_{true} *is an interface on input* x *and output* y*, with trivial contract* true*, making no assumptions on the inputs and no guarantees on the outputs. Suppose* $I_{y\neq x}$ *is another interface on* x *and* y*, with contract* $y \neq x$*, meaning that it guarantees that the value of the output will be different from the value of the input. As expected,* $I_{y\neq x}$ *refines* I_{true}*: because* $I_{y\neq x}$ *is "more deterministic" than* I_{true}*, that is, the output guarantees of* $I_{y\neq x}$ *are stronger. Now, consider the feedback connection* $x = y$*. This could be considered an allowed connection for* I_{true}*, since it does not contradict its contract: the resulting interface would be* $I_{x=y}$ *with contract* $x = y$*. But the same feedback connection contradicts the contract of* $I_{y\neq x}$*: the resulting interface would be* I_{false} *with contract* false*. Although* $I_{y\neq x}$ *refines* I_{true}*,* I_{false} *does not refine* $I_{x=y}$*, therefore, allowing arbitrary feedback would violate preservation of refinement by feedback. Notice that both* I_{true} *and* $I_{y\neq x}$ *are input-complete, which means that this problem is present also in that special case.*

Theorem 14 Hiding preserves refinement. *Let* $I_1 = (X, Y, f_1)$ *and* $I_2 = (X, Y, f_2)$ *be two interfaces such that* $I_2 \sqsubseteq I_1$*. Let* $y \in Y$ *be such that both* f_1 *and* f_2 *are independent from* y*. Then* $\mathsf{hide}(y, I_2) \sqsubseteq \mathsf{hide}(y, I_1)$*.*

It is worth noting that the above theorem would not hold if we were to define hiding without requiring independence of contracts from hidden variables. The example that follows illustrates this:

Example 15. *Consider the interfaces shown in Figure 11.* I_1 *and* I_2 *have a single input variable* x *and a single output* y*. It can be verified that* $I_2 \sqsubseteq I_1$*.* I_2 *is independent from* y*, whereas* I_1 *is not. Therefore,* $\mathsf{hide}(y, I_2)$ *is defined (and shown in the figure), whereas* $\mathsf{hide}(y, I_1)$ *is not defined. Suppose we were to define the latter as interface* I_1' *shown in the figure, which corresponds to existentially quantifying away* y *from all contracts, as is usually done. Then hiding would not preserve refinement. Indeed,* $\mathsf{hide}(y, I_2) \not\sqsubseteq I_1'$*, because* $x \cdot \neg x$ *is a legal input sequence in* I_1' *but not in* $\mathsf{hide}(y, I_2)$*.*

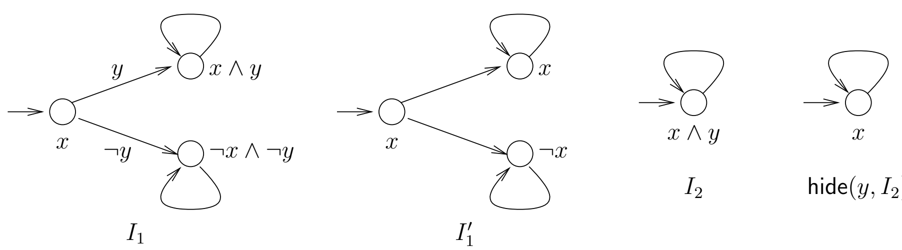

Fig. 11. Example illustrating the need for independence from hidden variables.

Theorem 15 Refinement and substitutability. *Let* I, I' *be two interfaces.*

(1) *If* $I' \sqsubseteq I$ *then* I' *can replace* I*.*

(2) *If* $I' \not\sqsubseteq I$ *and* I *is well-formed, then* I' *cannot replace* I*.*

The requirement that I be well-formed in part 2 of Theorem 15 is necessary, as the following example shows.

EXAMPLE 16. *Consider the finite-state interfaces I and I' defined by the automata shown in Figure 3. Both have a single boolean input variable x. I' is well-formed but I is not (I is well-formable, however, and I' is a witness). $I' \not\sqsubseteq I$, because at the initial state the input $x =$ false is legal for I but not for I'. But there is no environment E such that $I \models E$ but $I' \not\models E$.*

We next state a result that is not about refinement, but follows from properties of refinement:

THEOREM 16. *Let I, I' be well-formed interfaces. Then $I \equiv_e I'$ iff $I = I'$.*

PROOF. By Theorem 15, $I \equiv_e I'$ implies $I' \sqsubseteq I$ and $I \sqsubseteq I'$. The result follows by antisymmetry of refinement (Theorem 9). □

9.2 Discussion: alternative definition of refinement

The reader may wonder why Condition (18) could not be replaced with the simpler condition:

$$f'(s') \to f(s) \tag{23}$$

Indeed, for input-complete interfaces, Condition (16) reduces to Condition (23) – see Theorem 26 in Section 11. In general, however, Condition (16) is too strong in the sense that it results in a refinement condition that is sufficient but not necessary for substitutability, as the following example demonstrates:

EXAMPLE 17. *Consider interface $I_{id} := (\{x\}, \{y\}, x = y)$, and interface I_1 from Example 1. It can be checked that $I_{id} \sqsubseteq I_1$. If we used Condition (23) instead of Condition (18) in the definition of refinement, then I_{id} would not refine I_1: this is because $x = y \not\to x > 0$. Yet, by Theorem 15, I_{id} can replace I_1, that is, there is no environment E such that $I_1 \models E$ but $I_{id} \not\models E$.*

10. SHARED REFINEMENT AND SHARED ABSTRACTION

A *shared refinement* operator $\sqcap$ is introduced in [Doyen et al. 2008] for A/G interfaces, as a mechanism to combine two such interfaces I and I' into a single interface $I \sqcap I'$ that refines both I and I': $I \sqcap I'$ is able to accept inputs that are legal in either I or I', and provide outputs that are legal in both I and I'. Because of this, $I \sqcap I'$ can replace both I and I', which, as argued in [Doyen et al. 2008], is important for component reuse. A similar mechanism called *fusion* has also been proposed in [Benveniste et al. 2008].

[Doyen et al. 2008] also discusses shared refinement for extended (i.e., relational) interfaces and conjectures that it represents the greatest lower bound with respect to refinement. We show that this holds only if a certain condition is imposed. We call this condition *shared refinability*. It states that for every inputs that is legal in both I and I', the corresponding sets of outputs of I and I' must have a non-empty intersection. Otherwise, it is impossible to provide an output that is legal in both I and I'.

Definition 17 Shared refinement. *Two interfaces* $I = (X, Y, f)$ *and* $I' = (X', Y', f')$ *are* shared-refinable *if* $X = X'$, $Y = Y'$ *and the following formula is true for all* $s \in f \cap f'$:

$$\forall X : \big(\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s))\big) \rightarrow \exists Y : (f(s) \wedge f'(s)) \tag{24}$$

In that case, the shared refinement *of* I *and* I', *denoted* $I \sqcap I'$, *is the interface defined as follows:*

$$\begin{aligned} I \sqcap I' &:= (X, Y, f_{\sqcap}) \\ f_{\sqcap}(s) &:= \big(\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))\big) \wedge \big(\mathsf{in}(f(s)) \rightarrow f(s)\big) \wedge \big(\mathsf{in}(f'(s)) \rightarrow f'(s)\big) \end{aligned} \tag{25}$$

Example 18. *Consider interfaces* $I_{00} := (\{x\}, \{y\}, x = 0 \rightarrow y = 0)$ *and* $I_{01} := (\{x\}, \{y\}, x = 0 \rightarrow y = 1)$. I_{00} *and* I_{01} *are not shared-refinable because there is no way to satisfy* $y = 0 \wedge y = 1$ *when* $x = 0$.

For finite-state interfaces, shared refinement is computable. Let $M_i = (X, Y, L_i, \ell_{0,i}, C_i, T_i)$ be finite-state automata representing I_i, for $i = 1, 2$, respectively. Suppose I_1, I_2 are shared-refinable. Then, $I_1 \sqcap I_2$ can be represented as the automaton $M := (X, Y, L_1 \times L_2 \cup L_1 \cup L_2, (\ell_{0,1}, \ell_{0,2}), C, T)$, where C and T are defined as follows (guard g_{both} is defined as in (20)):

$$C(\ell) := \begin{cases} \big(\mathsf{in}(C_1(\ell_1)) \vee \mathsf{in}(C_2(\ell_2))\big) \wedge \big(\mathsf{in}(C_1(\ell_1)) \rightarrow C_1(\ell_1)\big) \wedge \big(\mathsf{in}(C_2(\ell_2)) \rightarrow C_2(\ell_2)\big), \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } \ell = (\ell_1, \ell_2) \in L_1 \times L_2 \\ C_1(\ell), \qquad\qquad\qquad\qquad\qquad \text{if } \ell \in L_1 \\ C_2(\ell), \qquad\qquad\qquad\qquad\qquad \text{if } \ell \in L_2 \end{cases} \tag{26}$$

$$\begin{aligned} T := \; & \{((\ell_1, \ell_2), g_{both} \wedge g_1 \wedge g_2, (\ell_1', \ell_2')) \mid (\ell_i, g_i, \ell_i') \in T_i, \text{ for } i = 1, 2\} \\ & \cup \{((\ell_1, \ell_2), \neg C_2(\ell_2) \wedge g_1, \ell_1') \mid (\ell_1, g_1, \ell_1') \in T_1\} \cup T_1 \\ & \cup \{((\ell_1, \ell_2), \neg C_1(\ell_1) \wedge g_2, \ell_2') \mid (\ell_2, g_2, \ell_2') \in T_2\} \cup T_2 \end{aligned}$$

As long as the contracts of both M_1 and M_2 are satisfied, M behaves as a synchronous product. If the contract of one automaton is violated, then M continues with the other.

Lemma 7. *If* I *and* I' *are shared-refinable interfaces then*

$$f(I) \cap f(I') \subseteq f(I \sqcap I') \subseteq f(I) \cup f(I')$$

Lemma 8. *Let* I *and* I' *be shared-refinable interfaces such that* $I = (X, Y, f)$, $I' = (X, Y, f')$ *and* $I \sqcap I' = (X, Y, f_{\sqcap})$. *Then for all* $s \in f \cap f'$:

$$\mathsf{in}(f_{\sqcap}(s)) \equiv \mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))$$

Theorem 17 Greatest lower bound. *If* I *and* I' *are shared-refinable interfaces then* $(I \sqcap I') \sqsubseteq I$, $(I \sqcap I') \sqsubseteq I'$, *and for any interface* I'' *such that* $I'' \sqsubseteq I$ *and* $I'' \sqsubseteq I'$, *we have* $I'' \sqsubseteq (I \sqcap I')$.

Shared-refinability is a sufficient, but not necessary condition to existence of an interface I'' that refines both I and I'. The following example illustrates this fact.

Example 19. *Consider interfaces* I *and* I' *shown in Figure 12. They have a single output variable* y, *and no inputs.* I *and* I' *are not shared-refinable. Indeed,*

$y = 1$ is initially possible in both interfaces, but after that, I requires $y = 0$ whereas I' requires $y = 1$, and there is no way of satisfying both. Nevertheless, an interface I'' exists that refines both I and I': I'' is the stateless interface with contract $y = 0$.

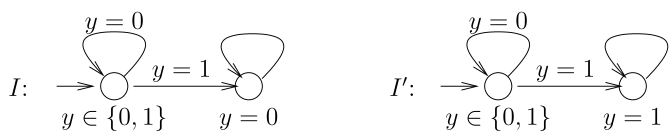

Fig. 12. Two interfaces that are not shared-refinable.

THEOREM 18. *If I and I' are shared-refinable interfaces and both are well-formed, then $I \sqcap I'$ is well-formed.*

It is useful to consider the dual operator to $\sqcap$, that we call *shared abstraction* and denote $\sqcup$. Contrary to $\sqcap$, $\sqcup$ is always defined, provided the interfaces have the same input and output variables:

DEFINITION 18 SHARED ABSTRACTION. *Two interfaces $I = (X, Y, f)$ and $I' = (X', Y', f')$ are* shared-abstractable *if $X = X'$ and $Y = Y'$. In that case, the* shared abstraction *of I and I', denoted $I \sqcup I'$, is the interface:*

$$I \sqcup I' := (X, Y, f_\sqcup)$$

$$f_\sqcup(s) := \begin{cases} \mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge \big(f(s) \vee f'(s)\big) & \textit{if } s \in f \cap f' \\ f(s) & \textit{if } s \in f \setminus f' \\ f'(s) & \textit{if } s \in f' \setminus f \end{cases} \tag{27}$$

Notice that it suffices to define $f_\sqcup(s)$ for $s \in f \cup f'$. Indeed, the above definition inductively implies $f_\sqcup \subseteq f \cup f'$:

LEMMA 9. *If I and I' are shared-abstractable interfaces then*

$$f(I) \cap f(I') \subseteq f(I \sqcup I') \subseteq f(I) \cup f(I')$$

For finite-state interfaces, shared abstraction is computable. Let $M_i = (X, Y, L_i, \ell_{0,i}, C_i, T_i)$ be finite-state automata representing I_i, for $i = 1, 2$, respectively. Suppose I_1, I_2 are shared-abstractable. Then, $I_1 \sqcup I_2$ can be represented as the automaton $M := (X, Y, L_1 \times L_2 \cup L_1 \cup L_2, (\ell_{0,1}, \ell_{0,2}), C, T)$, where C and T are defined as follows (guard g_{both} is defined as in (20)):

$$C(\ell) := \begin{cases} \mathsf{in}(C_1(\ell_1)) \wedge \mathsf{in}(C_2(\ell_2)) \wedge \big(L_1(\ell_1) \vee C_2(\ell_2)\big), & \text{if } \ell = (\ell_1, \ell_2) \in L_1 \times L_2 \\ C_1(\ell), & \text{if } \ell \in L_1 \\ C_2(\ell), & \text{if } \ell \in L_2 \end{cases} \tag{28}$$

$$\begin{aligned} T := {} & \{((\ell_1, \ell_2), g_{both} \wedge g_1 \wedge g_2, (\ell_1', \ell_2')) \mid (\ell_i, g_i, \ell_i') \in T_i, \text{ for } i = 1, 2\} \\ & \cup \{((\ell_1, \ell_2), \mathsf{in}(C_1(\ell_1)) \wedge \mathsf{in}(C_2(\ell_2)) \wedge \neg C_2(\ell_2) \wedge g_1, \ell_1') \mid (\ell_1, g_1, \ell_1') \in T_1\} \cup T_1 \\ & \cup \{((\ell_1, \ell_2), \mathsf{in}(C_1(\ell_1)) \wedge \mathsf{in}(C_2(\ell_2)) \wedge \neg C_1(\ell_1) \wedge g_2, \ell_2') \mid (\ell_2, g_2, \ell_2') \in T_2\} \cup T_2 \end{aligned}$$

Like the automaton for $I \sqcap I'$, M behaves as the synchronous product of M_1 and M_2, as long as the contracts of both are satisfied. When the contract of one is violated, then M continues with the other.

THEOREM 19 LEAST UPPER BOUND. *If I and I' are shared-abstractable interfaces then $I \sqsubseteq (I \sqcup I')$, $I' \sqsubseteq (I \sqcup I')$, and for any interface I'' such that $I \sqsubseteq I''$ and $I' \sqsubseteq I''$, we have $(I \sqcup I') \sqsubseteq I''$.*

Notice that, even when I, I' are both well-formed, $I \sqcup I'$ may be non-well-formed, or even non-well-formable. This occurs, for instance, when I and I' are stateless with contracts ϕ and ϕ' such that $\mathsf{in}(\phi) \wedge \mathsf{in}(\phi')$ is false. This does not contradict Theorem 19 since false is refined by any contract, as observed earlier.

11. THE INPUT-COMPLETE CASE

Input-complete interfaces do not restrict the set of input values, although they may provide no guarantees when the input values are illegal. Although input-complete interfaces are a special case of general interfaces, it is instructive to study them separately for two reasons: first, input-completeness makes things much simpler, thus easier to understand and implement; second, some interesting properties hold for input-complete interfaces but not in general.

THEOREM 20. *Every well-formed Moore interface is input-complete.*

Note that source interfaces are Moore by definition, therefore every well-formed source interface is also input-complete.

THEOREM 21. *Every input-complete interface is well-formed.*

Every interface I can be transformed into an input-complete interface $\mathsf{IC}(I)$. The illegal inputs of I become legal in $\mathsf{IC}(I)$, but $\mathsf{IC}(I)$ guarantees nothing about the value of the outputs when given such inputs. This transformation idea is well-known, for instance, it is called *chaotic closure* in [Broy and Stølen 2001].

DEFINITION 19 INPUT-COMPLETION. *Consider an interface $I = (X, Y, f)$. The* input-completion *of I, denoted $\mathsf{IC}(I)$, is the interface $\mathsf{IC}(I) := (X, Y, f_{ic})$, where $f_{ic}(s) := f(s) \vee \neg\mathsf{in}(f(s))$, for all $s \in \mathcal{A}(X \cup Y)^*$.*

THEOREM 22 INPUT-COMPLETION REFINES ORIGINAL. *If I is an interface then:*

(1) $\mathsf{IC}(I)$ is an input-complete interface.
(2) $\mathsf{IC}(I) \sqsubseteq I$.

Theorems 22 and 15 imply that for any environment E, if $I \models E$ then $\mathsf{IC}(I) \models E$. The converse does not hold in general (see Examples 1 and 10, and observe that I_2 is the input-complete version of I_1).

Composition by connection reduces to conjunction of contracts for input-complete interfaces, and preserves input-completeness:

THEOREM 23. *Let $I_i = (X_i, Y_i, f_i)$, $i = 1, 2$, be disjoint input-complete interfaces, and let θ be a connection between I_1, I_2. Then the contract f of the composite interface $\theta(I_1, I_2)$ is such that for all $s \in \mathcal{A}(X_{\theta(I_1,I_2)} \cup Y_{\theta(I_1,I_2)})^*$*

$$f(s) \equiv f_1(s) \wedge f_2(s) \wedge \rho_\theta$$

Moreover, $\theta(I_1, I_2)$ is input-complete.

Input-complete interfaces alone do not help in avoiding problems with arbitrary feedback compositions: indeed, in the example given in the introduction both interfaces I_{true} and $I_{y\neq x}$ are input-complete.[4] This means that in order to add a feedback connection (y, x) in an input-complete interface, we must still ensure that this interface is Moore w.r.t. input x. In that case, feedback preserves input-completeness.

Theorem 24. *Let $I = (X, Y, f)$ be an input-complete interface which is also Moore with respect to some $x \in X$. Let $\kappa = (y, x)$ be a feedback connection on I. Then $\kappa(I)$ is input-complete.*

Theorem 25. *Let $I = (X, Y, f)$ be an input-complete interface and let $y \in Y$, such that f is independent from y. Then, $\mathsf{hide}(y, I)$ is input-complete.*

Theorem 26 follows directly from Definitions 16 and 3:

Theorem 26. *Let I and I' be input-complete interfaces. Then $I' \sqsubseteq I$ iff $f(I') \subseteq f(I)$.*

For input-complete interfaces, the shared-refinability condition, i.e., Condition (24), simplifies to

$$\forall X : \exists Y : f(s) \wedge f'(s)$$

Clearly, this condition does *not* always hold. Indeed, the interfaces of Example 18 are not shared-refinable, even though they are input-complete. For shared-refinable input-complete interfaces, shared refinement reduces to intersection. Dually, for shared-abstractable input-complete interfaces, shared abstraction reduces to union.

Theorem 27 follows directly from Definitions 17, 18 and 3:

Theorem 27. *Let I and I' be input-complete interfaces.*

(1) If I and I' are shared-refinable then $f(I \sqcap I') = f(I) \cap f(I')$.

(2) If I and I' are shared-abstractable then $f(I \sqcup I') = f(I) \cup f(I')$.

12. THE DETERMINISTIC CASE

Deterministic interfaces produce a unique output for each legal input. As in the case of input-complete interfaces, it is instructive to study this sub-class of deterministic interfaces because the theory becomes simpler. Moreover, there is an interesting duality between the deterministic and input-complete case.

To begin, note that sink interfaces are by definition deterministic:

Theorem 28. *All sink interfaces are deterministic.*

Composition by connection reduces to composition of relations when the source interface is deterministic:

[4] It is not surprising that input-complete interfaces alone cannot solve the problems with arbitrary feedback compositions, since these are general problems of causality, not particular to interfaces.

THEOREM 29. *Consider two disjoint interfaces, $I_i = (X_i, Y_i, f_i)$, $i = 1, 2$, and a connection θ between I_1, I_2. Let $\theta(I_1, I_2) = (X, Y, f)$. If I_1 is deterministic, then $f(s) \equiv f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta$ for all states s.*

THEOREM 30 HIDING PRESERVES DETERMINISM. *Let $I = (X, Y, f)$ be a deterministic interface and let $y \in Y$, such that f is independent from y. Then, $\mathsf{hide}(y, I)$ is deterministic.*

THEOREM 31 REFINEMENT FOR DETERMINISTIC INTERFACES. *Let I and I' be deterministic interfaces. Then $I' \sqsubseteq I$ iff $f(I') \supseteq f(I)$.*

A corollary of Theorems 26 and 31 is that refinement for input-complete and deterministic interfaces is equality.

For deterministic interfaces, the shared-refinability condition, i.e., Condition (24), simplifies to

$$\forall X, Y : \big(\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s))\big) \to \big(f(s) \wedge f'(s)\big)$$

Again, this condition does not always hold. For shared-refinable deterministic interfaces, shared refinement reduces to union. Dually, for shared-abstractable deterministic interfaces, shared abstraction reduces to intersection.

THEOREM 32. *Let I and I' be deterministic interfaces.*

(1) *If I and I' are shared-refinable then $f(I \sqcap I') = f(I) \cup f(I')$.*
(2) *If I and I' are shared-abstractable then $f(I \sqcup I') = f(I) \cap f(I')$.*

Notice that Theorems 31 and 32 are duals of Theorems 26 and 27.

13. APPLICATION: NON-DEFENSIVE HARDWARE DESIGN

The theory developed in the previous sections is directly applicable to the domain of synchronous systems, which covers a broad class of applications, both in software and hardware. In particular, it applies to the class of applications captured in synchronous embedded software environments, as mentioned in the introduction. For instance, it can be used as a behavioral type theory for Simulink and other related models, in the spirit of [Roy and Shankar 2010].

Synchronous hardware is another important application domain for our work. To illustrate this, we consider *non-defensive hardware design*, which is an application of Meyer's ideas of *non-defensive programming* in the HW setting. To paraphrase Meyer, defensive programming consists in making SW modules input-complete, to guard against all possible inputs, including undesirable inputs that should not arise in principle [Meyer 1992]. Meyer argues that this is bad SW design practice, and we agree. Meyer proposes design-by-contract as an alternative. In a HW setting, the same defensive design practice is often encountered. Important benefits are to be obtained by abandoning this practice and by following the design-by-contract paradigm instead, to which our theory subscribes. We illustrate these points through an example.

Consider two HW components, Prod and Cons, having the input and output variables shown in Figure 13. Prod models a producer and Cons a consumer. Suppose that Cons requires that, once data starts being delivered at its input (i.e., once $valid_{in}$ becomes true), data continues to be delivered for 8 consecutive clock cycles. This is a typical requirement in HW "IP" ("intellectual property") blocks that perform signal processing [Ravindran and Yang 2010].

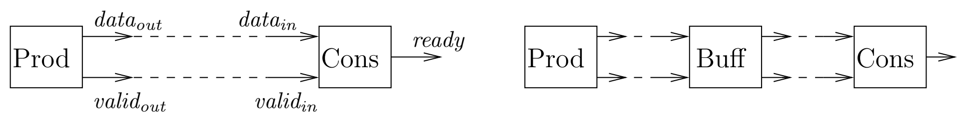

Fig. 13. Connecting a producer and a consumer.

We would like to connect Prod and Cons directly, as shown to the left of Figure 13. If we have no knowledge about Prod, however, we cannot do that, because Prod may produce data only intermittently, in which case the requirement of Cons is violated. Instead, we can insert a third component, Buff, to act as a mediator, as shown to the right of Figure 13. Buff acts as a temporary buffer that stores 8 values produced by Prod, and once 8 values become available, it signals and delivers them to Cons. The implementation details of Buff are not needed in this discussion. What is important is that Buff is an extra component that results in additional cost, both in terms of circuit size and performance (Cons must wait for Buff to accumulate 8 values before it starts processing them). We would like to avoid this cost. We can do this if we know that Prod conforms to the requirements of Cons: namely, that once Prod starts outputting data (i.e., once it sets $valid_{out}$ to true) it will continue to do so for 8 consecutive cycles. In that case, the direct connection of Prod to Cons is valid, and Buff becomes redundant.

The above situations can all be formally captured in our framework. Stateful interfaces can be used to model Prod and Cons, as shown in Figure 14. Interface Prod1 models a producer for which we have no knowledge, as in the first scenario described above. Interface Prod2 captures a different scenario where the producer is guaranteed to produce 16 consecutive outputs once it starts producing data. Prod2 is captured as an automaton extended with an integer counter i ranging between 1 and 15. Since the domain of i is finite, Prod2 is a finite-state interface.[5]

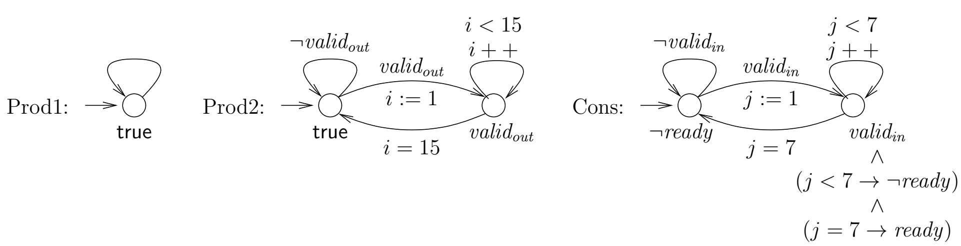

Fig. 14. Stateful interfaces for a producer and a consumer.

[5]Note that i is initialized to 1 and not to 0 when Prod2 switches from the initial location (with contract true) to the location with contract $valid_{out}$. This is because at this point one round where $valid_{out}$ was true already elapsed, namely, the round that triggered this transition when the automaton was at the initial location.

The interface Cons for the consumer is also shown in Figure 14. The structure of Cons is similar to that of Prod2. Cons requires that $valid_{in}$ remains true for 8 consecutive rounds once it has been set to true. At the end of this period, the output *ready* of Cons is set to true in order to signal that a batch of 8 consecutive inputs have been processed. Typically, Cons would also produce a value, but data values are completely abstracted in these interfaces. This results in simpler interfaces (with only a few states each), that can still be quite useful as this example illustrates.

Having these interfaces, we can formally state the fact that the unknown producer cannot be directly connected to our consumer. This is formalized by the fact that Prod1 and Cons are incompatible, that is, their serial composition is not well-formed (it is not well-formable either, since Prod1 has no inputs). On the other hand, we can formally state that Prod2 and Cons are compatible, therefore, an intermediate buffer is redundant in this case.

Note that the standard synchronous parallel composition of automata Prod1 and Cons does not reveal their incompatibility, since the conjunction of contracts true of Prod1 and $valid_{in}$ of Cons at its rightmost state, results in a satisfiable contract for the product state. On the other hand, a "demonic" interpretation of the non-determinism of contract true of Prod1 reveals the error. In this simple example, where Prod1 has no inputs, this demonic interpretation can be easily captured by transforming Cons to an automaton Cons' with an additional error location. This is similar to the error-completion transformation discussed in Section 2.4. Cons' moves to the error location when an illegal input is received, i.e., when $valid_{in}$ becomes false before 8 consecutive rounds have elapsed. Then, compatibility of Prod1 and Cons can be stated as a simple safety property on the standard parallel composition of Prod1 and Cons', namely, that the error location of Cons' is unreachable. This can be checked using a standard finite-state model-checker. In the general case, where Prod1 has inputs, compatibility cannot be stated as reachability and controller-synthesis algorithms must be used instead.

14. CONCLUSION AND PERSPECTIVES

We have proposed an interface theory that allows to reason formally about components and offers guarantees of substitutability. The framework we propose is general, and can be applied to a wide spectrum of cases, in particular within the synchronous model of computation. We are currently implementing our theory on the open-source Ptolemy software, and experimenting with different kinds of applications.

One major avenue for future work is to examine the current limitations on feedback compositions. Requiring feedback loops to contain Moore interfaces that "break" potential causality cycles is arguably a reasonable restriction in practice. After all, arbitrary feedback loops in synchronous models generally result in ambiguous semantics [Malik 1994; Berry 1999]. In many languages and tools these problems are avoided by making restrictions similar to (and often stricter than) ours. For example, Simulink and SCADE generally require a unit-delay to be present in every feedback loop. Similar restrictions are used in the synchronous language Lustre [Caspi et al. 1987].

Still, it would be interesting to study to what extent the current restrictions can be weakened. One possibility could be to refine the definition of Moore interfaces to include dependencies between specific pairs of input and output variables. For example, this would allow one to express the fact that in the parallel composition of $(\{x_1\}, \{y_1\}, x_1 = y_1)$ and $(\{x_2\}, \{y_2\}, x_2 = y_2)$, y_1 does not depend on x_2 and y_2 does not depend on x_1 (and therefore one of the feedbacks (y_1, x_2) or (y_2, x_1) can be allowed). Such an extension could perhaps be achieved by combining our relational interfaces with the *causality interfaces* of [Zhou and Lee 2008], input-output dependency information such as that used in reactive modules [Alur and Henzinger 1999], or the coarser *profiles* of [Lublinerman and Tripakis 2008]. A more general solution could involve studying fixpoints in a relational context, as is done, for instance, in [Desharnais and Möller 2005].

In the current version of our theory contracts are prefix-closed sets, and therefore cannot express liveness properties. For instance, in the example of the buffer that may fail (Figure 9), we cannot express the requirement that if writes are attempted infinitely often then they must eventually succeed. In the future we plan to study extensions of the theory to handle liveness properties. It is worth noting, however, that the current theory already avoids the problem of trivial implementations that achieve the specification by "doing nothing". An interface that "does nothing" is false, but false refines no other interface but itself. More generally, if an interface I is well-formed, then any refinement of I is well-formable, which means it can be executed forever without deadlocks.

Other directions of future work include examining canonical/minimal finite-state interfaces, as well as how non-deterministic automata can be used as representations of interfaces.

Acknowledgments

We are grateful to Jan Reineke, for helpful discussions on the definition of environments and on error-complete interfaces, to Marc Geilen, who observed that the deterministic case is a dual of the input-complete case, and to Kaushik Ravindran and Guang Yang, for motivating the buffer examples. We would also like to thank Albert Benveniste, Manfred Broy, Rupak Majumdar and Slobodan Matic for their valuable feedback. We finally would like to thank the anonymous Reviewers for their careful reading and comments that have greatly helped to improve this paper.

REFERENCES

Abadi, M. and Lamport, L. 1995. Conjoining specifications. *ACM Trans. Program. Lang. Syst. 17,* 3, 507–535.

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

Alur, R. and Henzinger, T. 1999. Reactive modules. *Formal Methods in System Design 15,* 7–48.

Alur, R., Henzinger, T., Kupferman, O., and Vardi, M. 1998. Alternating refinement relations. In *CONCUR'98.* LNCS, vol. 1466. Springer.

Back, R.-J. and Wright, J. 1998. *Refinement Calculus.* Springer.

Barringer, H., Kuiper, R., and Pnueli, A. 1984. Now you may compose temporal logic specifications. In *STOC '84: 16th ACM Symposium on Theory of Computing.* ACM, New York, NY, USA, 51–63.

Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., and Sofronis,

C. 2008. Multiple viewpoint contract-based specification and design. In *Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007*. Springer, 200–225.

Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. 2003. The synchronous languages 12 years later. *Proceedings of the IEEE 91,* 1 (Jan.), 64–83.

Berry, G. 1999. The Constructive Semantics of Pure Esterel.

Broy, M. 1997. Compositional refinement of interactive systems. *J. ACM 44,* 6, 850–891.

Broy, M. and Stølen, K. 2001. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer.

Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. 1987. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symp. POPL*. ACM.

Chakrabarti, A., de Alfaro, L., Henzinger, T., and Mang, F. 2002. Synchronous and bidirectional component interfaces. In *CAV*. LNCS 2404. Springer, 414–427.

Cheon, Y. and Leavens, G. 1994. The Larch/Smalltalk interface specification language. *ACM Trans. Softw. Eng. Methodol. 3,* 3, 221–153.

de Alfaro, L. 2004. Game models for open systems. In *Verification: Theory and Practice*, N. Dershowitz, Ed. Lecture Notes in Computer Science, vol. 2772. Springer, 192–213.

de Alfaro, L. and Henzinger, T. 2001a. Interface automata. In *Foundations of Software Engineering (FSE)*. ACM Press.

de Alfaro, L. and Henzinger, T. 2001b. Interface theories for component-based design. In *EMSOFT'01*. Springer, LNCS 2211.

Desharnais, J. and Möller, B. 2005. Least reflexive points of relations. *Higher Order Symbol. Comput. 18,* 1-2, 51–77.

Dhara, K. and Leavens, G. 1996. Forcing behavioral subtyping through specification inheritance. In *ICSE'96: 18th Intl. Conf. on Software Engineering.* IEEE Computer Society, 258–267.

Dijkstra, E. 1972. Notes on structured programming. In *Structured programming*, O. Dahl, E. Dijkstra, and C. Hoare, Eds. Academic Press, London, UK, 1–82.

Dill, D. 1987. *Trace theory for automatic hierarchical verification of speed-independent circuits*. MIT Press, Cambridge, MA, USA.

Doyen, L., Henzinger, T., Jobstmann, B., and Petrov, T. 2008. Interface theories with component reuse. In *8th ACM & IEEE International conference on Embedded software, EMSOFT*. 79–88.

Floyd, R. 1967. Assigning meanings to programs. In *In. Proc. Symp. on Appl. Math. 19.* American Mathematical Society, 19–32.

Frappier, M., Mili, A., and Desharnais, J. 1998. Unifying program construction and modification. *Logic Journal of the IGPL 6*, 317–340.

Grumberg, O. and Long, D. 1994. Model checking and modular verification. *ACM Trans. Program. Lang. Syst. 16,* 3, 843–871.

Guttag, J. and Horning, J. 1993. *Larch: languages and tools for formal specification.* Springer.

Hehner, E. and Parnas, D. 1985. Technical correspondence. *Commun. ACM 28,* 5, 534–538.

Henzinger, T., Qadeer, S., and Rajamani, S. 1998. You assume, we guarantee: Methodology and case studies. In *CAV'98*. LNCS, vol. 1427. Springer-Verlag.

Henzinger, T. and Sifakis, J. 2007. The discipline of embedded systems design. *IEEE Computer 40,* 10, 32–40.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Comm. ACM 12,* 10, 576–580.

Hoare, C. A. R. 1985. Programs are predicates. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 141–155.

Jones, C. B. 1983. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems 5,* 4.

Jonsson, B. 1994. Compositional specification and verification of distributed systems. *ACM Trans. Program. Lang. Syst. 16,* 2, 259–303.

Kahl, W. 2003. Refinement and development of programs from relational specifications. *Electronic Notes in Theoretical Computer Science 44,* 3, 51 – 93. RelMiS 2001, Relational Methods in Software (a Satellite Event of ETAPS 2001).

Leavens, G. 1994. Inheritance of interface specifications. *SIGPLAN Not. 29,* 8, 129–138.

Leavens, G. and Cheon, Y. 2006. Design by Contract with JML. Available at `http://www.jmlspecs.org/jmldbc.pdf`.

Lee, E. 2008. Cyber physical systems: Design challenges. Tech. Rep. UCB/EECS-2008-8, EECS Department, University of California, Berkeley. Jan.

Lee, E. and Sangiovanni-Vincentelli, A. 1998. A unified framework for comparing models of computation. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems 17,* 12 (Dec.), 1217–1229.

Lee, E. and Xiong, Y. 2001. System-level types for component-based design. In *EMSOFT'01: 1st Intl. Workshop on Embedded Software*. Springer, 237–253.

Liskov, B. 1979. Modular program construction using abstractions. In *Abstract Software Specifications*. Lecture Notes in Computer Science, vol. 86. Springer, 354–389.

Liskov, B. and Wing, J. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16,* 6, 1811–1841.

Lublinerman, R. and Tripakis, S. 2008. Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams. In *Design, Automation, and Test in Europe (DATE'08)*. ACM.

Lynch, N. and Tuttle, M. 1989. An introduction to input/output automata. *CWI Quarterly 2*, 219–246.

Malik, S. 1994. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design 13,* 7, 950–956.

McMillan, K. 1997. A compositional rule for hardware design refinement. In *Computer Aided Verification (CAV'97)*. LNCS, vol. 1254. Springer-Verlag.

Meyer, B. 1992. Applying "design by contract". *Computer 25,* 10, 40–51.

Miller, S., Whalen, M., and Cofer, D. 2010. Software model checking takes off. *Comm. ACM 53,* 2, 58–64.

Misra, J. and Chandy, K. 1981. Proofs of networks of processes. *IEEE Transactions on Software Engineering 7,* 4 (July), 417–426.

Nelson, G. 1989. A generalization of dijkstra's calculus. *ACM Trans. Program. Lang. Syst. 11,* 4, 517–561.

Nierstrasz, O. 1993. Regular types for active objects. *SIGPLAN Not. 28,* 10, 1–15.

Parnas, D. 1983. A generalized control structure and its formal definition. *Commun. ACM 26,* 8, 572–581.

Pierce, B. 2002. *Types and Programming Languages*. MIT Press.

Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., and Passerone, R. 2010. A modal interface theory for component based design. Available at `http://www.irisa.fr/distribcom/benveniste/pub/Fundamenta2010.html`.

Ravindran, K. and Yang, G. 2010. Personal communication.

Roy, P. and Shankar, N. 2010. SimCheck: An expressive type system for Simulink. In *2nd NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, C. Muñoz, Ed. NASA, Langley Research Center, Hampton VA 23681-2199, USA, 149–160.

Shankar, N. 1998. Lazy compositional verification. In *Compositionality: The Significant Difference (COMPOS'97)*. Springer, 541–564.

Spivey, J. M. 1989. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Stark, E. 1985. A proof technique for rely/guarantee properties. In *Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*. LNCS. Springer-Verlag.

Tourlakis, G. 2008. *Mathematical Logic*. Wiley.

Tripakis, S., Lickly, B., Henzinger, T., and Lee, E. 2009a. On relational interfaces. Tech. Rep. UCB/EECS-2009-60, EECS Department, University of California, Berkeley. May.

Tripakis, S., Lickly, B., Henzinger, T., and Lee, E. 2009b. On Relational Interfaces. In *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT'09)*. ACM, 67–76.

Wirth, N. 1971. Program development by stepwise refinement. *Comm. ACM 14,* 4, 221–227.

Zhou, Y. and Lee, E. 2008. Causality interfaces for actor networks. *ACM Trans. Embed. Comput. Syst. 7,* 3, 1–35.

A. PROOFS

Proof of Lemma 1. By induction. ε belongs in both f and f'. Suppose $s \cdot a \in f'$. Thus $s \in f'$. By the induction hypothesis, $s \in f$. From $s \cdot a \in f'$ we get $a \in f'(s)$. Since $f'(s) = f(s) \cap \phi_s$, we have $a \in f(s)$, therefore $s \cdot a \in f$. $\square$

Proof of Theorem 1. Well-formedness implies well-formability for all interfaces. For the converse, let $I = (X, Y, f)$ be a well-formable interface. Then there exists a witness $I' = (X, Y, f')$ such that I' is well-formed.

First, suppose that I is stateless. Then $f(s) = f(\varepsilon)$ for any s. Since I' is a witness, $f'(\varepsilon) = f(\varepsilon) \wedge \phi_\varepsilon$, for some property ϕ_ε over X. Since I' is well-formed, $f'(\varepsilon)$ is non-empty, thus, $f(\varepsilon)$ is also non-empty, thus, so is $f(s)$ for any s.

Second, suppose that I is a source, that is, $X = \emptyset$. Since I' is a witness, for any state s, $f'(s) = f(s) \wedge \phi_s$, where ϕ_s is a property over X. Since X is empty, ϕ_s can be either true or false. Since $f'(s)$ is non-empty, ϕ_s must be true for any s. Therefore, $f(s) = f'(s)$ for any s, thus, $f(s)$ is non-empty for all s. $\square$

Proof of Lemma 2. Following Definition 9, we have:

$$I_1 \| I_2 \;=\; (X_1 \cup X_2, Y_1 \cup Y_2, f)$$

where for all $s \in \mathcal{A}(X_1 \cup X_2 \cup Y_1 \cup Y_2)^*$

$$f(s) \;=\; f_1(s_1) \wedge f_2(s_2) \wedge \big(\forall Y_1 \cup Y_2 : f_1(s_1) \rightarrow \mathsf{in}(f_2(s_2))\big)$$

Observe that $\mathsf{in}(f_2(s_2))$ is a formula over X_2, that is, does not depend on $Y_1 \cup Y_2$. Therefore,

$$\big(\forall Y_1 \cup Y_2 : f_1(s_1) \rightarrow \mathsf{in}(f_2(s_2))\big) \equiv \neg(\exists Y_1 \cup Y_2 : f_1(s_1) \wedge \neg \mathsf{in}(f_2(s_2))) \equiv$$
$$\neg(\neg \mathsf{in}(f_2(s_2)) \wedge \exists Y_1 \cup Y_2 : f_1(s_1)) \equiv \big(\mathsf{in}(f_2(s_2)) \vee \neg\exists Y_1 \cup Y_2 : f_1(s_1)\big)$$

Now, observe that $\phi \rightarrow \mathsf{in}(\phi)$ and $\phi \rightarrow (\phi \vee \phi')$ are valid formulas for any ϕ, ϕ'. Therefore, $f_2(s_2) \rightarrow \mathsf{in}(f_2(s_2))$ is valid and so is $\mathsf{in}(f_2(s_2)) \rightarrow \mathsf{in}(f_2(s_2)) \vee \neg\exists Y_1 \cup Y_2 : f_1(s_1)$, which gives

$$\Big(f_1(s_1) \wedge f_2(s_2) \wedge \forall Y_1 \cup Y_2 : f_1(s_1) \rightarrow \mathsf{in}(f_2(s_2))\Big) \equiv (f_1(s_1) \wedge f_2(s_2))$$

$\square$

Proof of Theorem 3. For simplicity of notation, we conduct the proof assuming the interfaces are stateless. The proof is almost identical for general interfaces, except that $f(s)$ replaces ϕ, $f'(s)$ replaces ϕ', and so on.

Suppose the setting is as illustrated in Figure 15. That is, $I_1 = (X_1, Y_1 \cup Y_{12} \cup Y_{13}, \phi_1)$; $I_2 = (X_2 \cup X_{12}, Y_2 \cup Y_{23}, \phi_2)$; $I_3 = (X_3 \cup X_{13} \cup X_{23}, Y_3, \phi_3)$; and θ_{12} connects X_{12} and Y_{12}; θ_{13} connects X_{13} and Y_{13}; θ_{23} connects X_{23} and Y_{23}.

Our first step is to clearly express what the definitions tell us about $I := (\theta_{12} \cup \theta_{13})\,(I_1, \theta_{23}(I_2, I_3))$ and $I' := (\theta_{13} \cup \theta_{23})\,(\theta_{12}(I_1, I_2), I_3)$.

For simplicity, we will use the notation ρ_θ to refer to $\bigwedge_{(y,x)\in\theta} y = x$. We also refer to the outputs of $\theta_{12}(I_1, I_2)$ as $P = Y_1 \cup Y_{12} \cup Y_{13} \cup X_{12} \cup Y_2 \cup Y_{23}$ and the outputs of $\theta_{23}(I_2, I_3)$ as $Q = Y_2 \cup Y_{23} \cup X_{23} \cup Y_3$ and the overall outputs as $O = Y_1 \cup Y_2 \cup Y_3 \cup Y_{12} \cup Y_{13} \cup Y_{23} \cup X_{12} \cup X_{13} \cup X_{23}$.

The definitions are as follows:

$$\begin{aligned}
\theta_{12}(I_1, I_2) &= \quad (X_1 \cup X_2, P, \phi_1 \wedge \phi_2 \wedge \rho_{\theta_{12}} \wedge \forall P : \phi_1 \wedge \rho_{\theta_{12}} \rightarrow \mathsf{in}(\phi_2)) \\
\theta_{23}(I_2, I_3) &= \quad (X_2 \cup X_{12} \cup X_3 \cup X_{13}, Q, \phi_2 \wedge \phi_3 \wedge \rho_{\theta_{23}} \wedge \forall Q : \phi_2 \wedge \rho_{\theta_{23}} \rightarrow \mathsf{in}(\phi_3))
\end{aligned}$$

Let ϕ_{12} and ϕ_{23} be the contracts of $\theta_{12}(I_1, I_2)$ and $\theta_{23}(I_2, I_3)$, respectively. Then:

$$\begin{aligned}
I &= (X_1 \cup X_2 \cup X_3, O, \phi_{12} \wedge \phi_3 \wedge \rho_{\theta_{13}} \wedge \rho_{\theta_{23}} \wedge \forall O : \phi_{12} \wedge \rho_{\theta_{13}} \wedge \rho_{\theta_{23}} \rightarrow \mathsf{in}(\phi_3)) \\
I' &= (X_1 \cup X_2 \cup X_3, O, \phi_1 \wedge \phi_{23} \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{13}} \wedge \forall O : \phi_1 \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{13}} \rightarrow \mathsf{in}(\phi_{23}))
\end{aligned}$$

Let ϕ and ϕ' be the contracts of I and I', respectively. Letting $\theta = \theta_{12} \cup \theta_{13} \cup \theta_{23}$ and substituting, we get:

$$\begin{aligned}
\phi &\equiv \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \rho_\theta \wedge (\forall P : \phi_1 \wedge \rho_{\theta_{12}} \rightarrow \mathsf{in}(\phi_2)) \wedge (\forall O : \phi_{12} \wedge \rho_{\theta_{13}} \wedge \rho_{\theta_{23}} \rightarrow \mathsf{in}(\phi_3)) \\
\phi' &\equiv \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \rho_\theta \wedge (\forall Q : \phi_2 \wedge \rho_{\theta_{23}} \rightarrow \mathsf{in}(\phi_3)) \wedge (\forall O : \phi_1 \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{13}} \rightarrow \mathsf{in}(\phi_{23}))
\end{aligned}$$

In order to simplify discussion, we will name the subformulae as follows:

$$\begin{aligned}
C &:= \quad \forall P : \phi_1 \wedge \rho_{\theta_{12}} \rightarrow \mathsf{in}(\phi_2) \\
D &:= \quad \forall O : \phi_{12} \wedge \rho_{\theta_{13}} \wedge \rho_{\theta_{23}} \rightarrow \mathsf{in}(\phi_3) \\
E &:= \quad \forall Q : \phi_2 \wedge \rho_{\theta_{23}} \rightarrow \mathsf{in}(\phi_3) \\
F &:= \quad \forall O : \phi_1 \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{13}} \rightarrow \mathsf{in}(\phi_{23})
\end{aligned}$$

Then, we have

$$\begin{aligned}
\phi &\equiv \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \rho_\theta \wedge C \wedge D \\
\phi' &\equiv \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \rho_\theta \wedge E \wedge F
\end{aligned}$$

Also note that

$$\rho_\theta \equiv \rho_{\theta_{12}} \wedge \rho_{\theta_{13}} \wedge \rho_{\theta_{23}}$$

In order to prove equivalence of I and I', we need to prove that the following four formulae are valid:

$$\phi \rightarrow E, \quad \phi \rightarrow F, \quad \phi' \rightarrow C, \text{ and } \phi' \rightarrow D$$

Proof of $\phi \rightarrow E$: Let (x, q, o) be an arbitrary assignment such that $(x, q, o) \models \phi$, where $x \in X_1 \cup X_2 \cup X_3$, $q \in Q$, and $o \in O \setminus Q$. We want to show that $(x, q, o) \models E$, i.e., that $(x, o) \models E$, since E has no free variables in Q. Expanding E, we must show $(x, o) \models \forall Q : \phi_2 \wedge \rho_{\theta_{23}} \rightarrow \mathsf{in}(\phi_3)$. Let q' be an arbitrary assignment over Q such that $(x, q', o) \models \phi_2 \wedge \rho_{\theta_{23}}$. We want to show that $(x, q', o) \models \mathsf{in}(\phi_3)$. We will use the fact D which is implied by ϕ. We will first show that $(x, q', o) \models \phi_{12}$. Expanding ϕ_{12}, we must show:

$$(x, q', o) \models \phi_1 \wedge \phi_2 \wedge \rho_\theta \wedge (\forall P : \phi_1 \wedge \rho_{\theta_{12}} \rightarrow \mathsf{in}(\phi_2))$$

or equivalently

$$(x, q', o) \models \phi_1 \wedge \phi_2 \wedge \rho_\theta \wedge C.$$

Clearly, we have $(x, q', o) \models \phi_2 \wedge \rho_{\theta_{23}}$ by construction of q'. Also (x, o) satisfies $\phi_1 \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{13}} \wedge C$ since this formula contains no free variables in Q and $(x, q, o) \models \phi$. Thus we have shown $(x, q', o) \models \phi_{12}$, and, by D, we have $(x, q', o) \models \mathsf{in}(\phi_3)$. Thus we have $(x, o) \models E$. End of proof of $\phi \to E$.

Proof of $\phi \to F$: Suppose we are given an assignment $(x, q, o) \models \phi$ where x is over $X_1 \cup X_2 \cup X_3$, q is over Q, and o is over $O \setminus Q$. We want to show that $(x, q, o) \models F$ (i.e. $x \models F$).

Let (q', o') be an arbitrary assignment over O such that $(x, q', o') \models \phi_1 \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{23}}$. We want to now show that $(x, q', o') \models \mathsf{in}(\phi_{23})$. To do so, we first expand $\mathsf{in}(\phi_{23})$:

$$\mathsf{in}(\phi_{23}) \equiv (\exists Q : \phi_2 \wedge \phi_3 \wedge \rho_{\theta_{23}}) \wedge (\forall Q : \phi_2 \wedge \rho_{\theta_{23}} \to \mathsf{in}(\phi_3))$$

Thus we can reduce the proof to two parts:

(a) $(x, o') \models (\exists Q : \phi_2 \wedge \phi_3 \wedge \rho_{\theta_{23}})$, and
(b) $(x, o') \models (\forall Q : \phi_2 \wedge \rho_{\theta_{23}} \to \mathsf{in}(\phi_3))$

For part (a), we want to show that for any assignment q_a over Q: $(x, q_a, o') \models \phi_2 \wedge \rho_{\theta_{23}} \Rightarrow (x, q_a, o') \models \mathsf{in}(\phi_3)$. We start with such an assignment q_a. Combining this with the fact that $(x, o') \models \phi_1 \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{23}}$, we get $(x, q_a, o') \models \phi_1 \wedge \phi_2 \wedge \rho_\theta$. Combined with the fact that $x \models C$, we get $(x, q_a, o') \models \phi_1 \wedge \phi_2 \wedge \rho_\theta \wedge C$. This is exactly the premise of D. Since $x \models D$, this gives us $(x, q_a, o') \models \mathsf{in}(\phi_3)$, which is exactly what we wanted to prove.

For part (b), we want to show that there exists an assignment over Q that models $\phi_2 \wedge \phi_3 \wedge \rho_{\theta_{23}}$. For our purposes, we will divide this assignment into q_{Y2} over $Y_2 \cup Y_{23}$, q_{X3} over X_{23}, and q_{Y3} over Y_3. First, since $x \models C$ and $(x, o') \models \phi_1 \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{23}}$ we have that $(x, o') \models \mathsf{in}(\phi_2)$. Expanding the definition of in, this means that $\exists Y_2 : \phi_2$. Using this as our assignment of q_{Y2}, we have that $(x, q_{Y2}, o') \models \phi_2$. We can set the values of X_{23} to those of Y_{23} in order to get an assignment of q_{X3} that satisfies $\rho_{\theta_{23}}$. Combining the definition of o' with the assignments to q_{Y2}, q_{X3} with the fact that $x \models C$, gives us:

$$(x, q_{Y2}, q_{X3}, o') \models (\phi_1 \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{23}}) \wedge (\phi_2 \wedge \rho_{\theta_{23}}) \wedge C$$

Since this is exactly the premise of D, we get $(x, q_{Y2}, q_{X3}, o') \models \mathsf{in}(\phi_3)$. But this means that $\exists Y_3 : \phi_3$. Using this as our assignment to q_{Y3}, we get $(x, q_{Y2}, q_{X3}, q_{Y3}, o') \models \phi_3$. Combining the terms that we have satisfied over the course of our assignment, we get $(x, q_{Y2}, q_{X3}, q_{Y3}, o') \models \phi_2 \wedge \phi_3 \wedge \rho_{\theta_{23}}$, which is what we wanted to prove.

Combining our results from part (a) and part (b) we get $(x, o') \models \mathsf{in}(\phi_{23})$. Thus $(x, q, o) \models F$. End of proof of $\phi \to F$.

Proof of $\phi' \to C$: Suppose $(x, p, o) \models B$ where $x \in X_1 \cup X_2 \cup X_3$, $p \in P$, and $o \in O \setminus P$. We want to show that $(x, p, o) \models C$ (i.e. $(x, o) \models C$).

Let p' be an assignment over P such that $(x, p', o) \models \phi_1 \wedge \rho_{\theta_{12}}$. Now take o' over $O \setminus P$ such that $(x, p', o') \models \phi_1 \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{13}}$. This can be done by setting the variables of Y_{13} to those of X_{13}. By F, we have that $(x, p', o') \models \mathsf{in}(\phi_{23})$, so in particular, $(x, p', o') \models \mathsf{in}(\phi_2)$. Since $\mathsf{in}(\phi_2)$ does not contain free variables in $O \setminus P$, this means $(x, p', o) \models \mathsf{in}(\phi_2)$. Thus we have $(x, o) \models C$. End of proof of $\phi' \to C$.

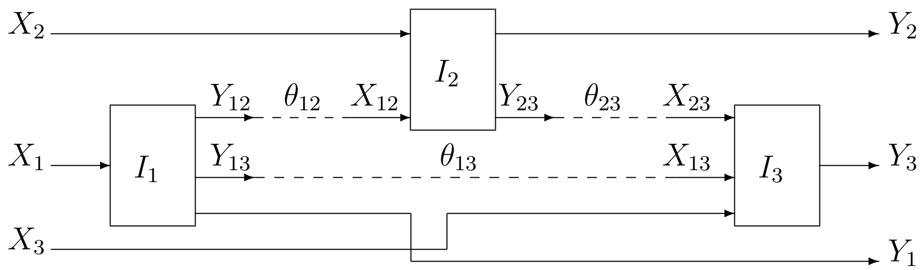

Fig. 15. Setting used in the proof of Theorem 3.

Proof of $\phi' \to D$: Suppose $(x, o) \models \phi'$, where x is over $X_1 \cup X_2 \cup X_3$, and o is over O.

Let o' be an arbitrary assignment over O with $(x, o') \models \phi_{12} \wedge \rho_{\theta_{13}} \wedge \rho_{\theta_{23}}$. Clearly $(x, o') \models \phi_1 \wedge \rho_{\theta_{12}} \wedge \rho_{\theta_{13}}$. By F, we have $(x, o') \models \mathsf{in}(\phi_{23})$. But this also means that $(x, o') \models \mathsf{in}(\phi_3)$ Thus we have $(x, o) \models D$. End of proof of $\phi' \to D$. □

Proof of Theorem 4. Following Definition 12, we derive

$$\begin{aligned} \kappa_1(\kappa_2(I)) &= (X \setminus \{x_1, x_2\}, Y \cup \{x_1, x_2\}, f_1) \\ \kappa_2(\kappa_1(I)) &= (X \setminus \{x_1, x_2\}, Y \cup \{x_1, x_2\}, f_2) \end{aligned}$$

where for all $s \in \mathcal{A}(X \cup Y)^*$

$$f_1(s) \equiv (f(s) \wedge y_1 = x_1 \wedge y_2 = x_2) \equiv f_2(s)$$

□

Proof of Theorem 5. Let $I_i = (X_i, Y_i, f_i)$, for $i = 1, 2$. Let $\kappa_i = (y_i, x_i)$, for $i = 1, 2$. Then, since κ_i are valid feedback connections, I_i must be Moore w.r.t. x_i, for $i = 1, 2$.

Claim 1: $\kappa_1(\theta(I_1, I_2)) = \theta(\kappa_1(I_1), I_2)$. We have:

$$\begin{aligned} \theta(I_1, I_2) &= \big(X_{\theta(I_1,I_2)}, Y_{\theta(I_1,I_2)}, f_t(s)\big) \\ \kappa_1(\theta(I_1, I_2)) &= \big(X_{\theta(I_1,I_2)} \setminus \{x_1\}, Y_{\theta(I_1,I_2)} \cup \{x_1\}, f^1_{kt}(s)\big) \\ \kappa_1(I_1) &= \big(X_1 \setminus \{x_1\}, Y_1 \cup \{x_1\}, f^1_k(s)\big) \\ \theta(\kappa_1(I_1), I_2) &= \big(X_{\theta(I_1,I_2)} \setminus \{x_1\}, Y_{\theta(I_1,I_2)} \cup \{x_1\}, f^1_{tk}(s)\big) \end{aligned}$$

where the contracts are as follows:

$$\begin{aligned} f_t(s) &= f_1(s) \wedge f_2(s) \wedge \rho_\theta \wedge \forall Y_{\theta(I_1,I_2)} : \big((f_1(s) \wedge \rho_\theta) \to \mathsf{in}(f_2(s))\big) \\ f^1_{kt}(s) &= f_t(s) \wedge x_1 = y_1 \\ &= f_1(s) \wedge f_2(s) \wedge \rho_\theta \wedge \forall Y_{\theta(I_1,I_2)} : \big((f_1(s) \wedge \rho_\theta) \to \mathsf{in}(f_2(s))\big) \wedge x_1 = y_1 \\ f^1_k(s) &= f_1(s) \wedge x_1 = y_1 \\ f^1_{tk}(s) &= f^1_k(s) \wedge f_2(s) \wedge \rho_\theta \wedge \forall Y_{\theta(I_1,I_2)} \cup \{x_1\} : \big((f^1_k(s) \wedge \rho_\theta) \to \mathsf{in}(f_2(s))\big) \\ &= (f_1(s) \wedge x_1 = y_1) \wedge f_2(s) \wedge \rho_\theta \wedge \\ &\qquad \forall Y_{\theta(I_1,I_2)} \cup \{x_1\} : ((f_1(s) \wedge x_1 = y_1 \wedge \rho_\theta) \to \mathsf{in}(f_2(s))) \end{aligned}$$

Since θ only changes input variables of I_2 to outputs, and κ_1 only changes an input port of I_1 to an output, the composition of these two connections in either order is well formed, and will result in an interface with the same input and output variables. Thus, it remains to prove that the resulting contract is also the same, i.e. that $f^1_{tk}(s) = f^1_{kt}(s)$. Let $Y = Y_{\theta(I_1,I_2)}$. We have:

$$
\begin{aligned}
f^1_{tk}(s) &= (f_1(s) \wedge x_1 = y_1) \wedge f_2(s) \wedge \rho_\theta \wedge \\
&\qquad\qquad \forall Y \cup \{x_1\} : ((f_1(s) \wedge x_1 = y_1 \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s))) \\
&= (f_1(s) \wedge x_1 = y_1) \wedge f_2(s) \wedge \rho_\theta \wedge \\
&\qquad\qquad \forall Y \cup \{x_1\} : (\neg f_1(s) \vee x_1 \neq y_1 \vee \neg\rho_\theta \vee \mathsf{in}(f_2(s))) \\
&= (f_1(s) \wedge x_1 = y_1) \wedge f_2(s) \wedge \rho_\theta \wedge \\
&\qquad\qquad \forall Y : (\neg f_1(s) \vee \neg\rho_\theta \vee \mathsf{in}(f_2(s)) \vee \forall x_1 : x_1 \neq y_1) \\
&= (f_1(s) \wedge x_1 = y_1) \wedge f_2(s) \wedge \rho_\theta \wedge \\
&\qquad\qquad \forall Y : (\neg f_1(s) \vee \neg\rho_\theta \vee \mathsf{in}(f_2(s)) \vee false) \\
&= f_1(s) \wedge f_2(s) \wedge \rho_\theta \wedge \forall Y : ((f_1(s) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s))) \wedge x_1 = y_1 \\
&= f^1_{kt}(s)
\end{aligned}
$$

Claim 2: $\kappa_2(\theta(I_1, I_2)) = \theta(I_1, \kappa_2(I_2))$. We have:

$$
\begin{aligned}
\theta(I_1, I_2) &= \big(X_{\theta(I_1,I_2)}, Y_{\theta(I_1,I_2)}, f_t(s)\big) \\
\kappa_2(\theta(I_1, I_2)) &= \big(X_{\theta(I_1,I_2)} \setminus \{x_2\}, Y_{\theta(I_1,I_2)} \cup \{x_2\}, f^2_{kt}(s)\big) \\
\kappa_2(I_2) &= \big(X_2 \setminus \{x_2\}, Y_2 \cup \{x_2\}, f^2_k(s)\big) \\
\theta(I_1, \kappa_1(I_2)) &= \big(X_{\theta(I_1,I_2)} \setminus \{x_2\}, Y_{\theta(I_1,I_2)} \cup \{x_2\}, f^2_{tk}(s)\big)
\end{aligned}
$$

where the new contracts are as follows:

$$
\begin{aligned}
f^2_{kt}(s) &= f_1(s) \wedge f_2(s) \wedge \rho_\theta \wedge \forall Y_{\theta(I_1,I_2)} : ((f_1(s) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s))) \wedge x_2 = y_2 \\
f^2_k(s) &= f_2(s) \wedge x_2 = y_2 \\
f^2_{tk}(s) &= f_1(s) \wedge (f_2(s) \wedge x_2 = y_2) \wedge \rho_\theta \wedge \\
&\qquad\qquad \forall Y_{\theta(I_1,I_2)} \cup \{x_2\} : ((f_1(s) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s) \wedge x_2 = y_2))
\end{aligned}
$$

Here we need to rely on the assumption $\mathsf{InVars}(\kappa_2) \cap \mathsf{InVars}(\theta) = \emptyset$ to prove that the composition by κ_2 and θ in either order is well formed, and that the input and output variables of the resulting interface are the same. Thus, it remains to prove that the resulting contract is also the same, i.e. that $f^2_{tk}(s) = f^2_{kt}(s)$. As before, let

$Y = Y_{\theta(I_1,I_2)}$. We have:

$$
\begin{aligned}
f^2_{tk}(s) &= f_1(s) \wedge (f_2(s) \wedge x_2 = y_2) \wedge \rho_\theta \wedge \\
&\qquad\qquad \forall Y \cup \{x_2\} : ((f_1(s) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s) \wedge x_2 = y_2)) \\
&= (f_1(s) \wedge x_2 = y_2) \wedge f_2(s) \wedge \rho_\theta \wedge \\
&\qquad\qquad \forall Y : ((f_1(s) \wedge \rho_\theta) \rightarrow \exists Y_2 \cup \{x_2\} : (f_2(s) \wedge x_2 = y_2)) \\
&= (f_1(s) \wedge x_2 = y_2) \wedge f_2(s) \wedge \rho_\theta \wedge \\
&\qquad\qquad \forall Y : ((f_1(s) \wedge \rho_\theta) \rightarrow \exists Y_2 : (f_2(s) \wedge \exists x_2 : x_2 = y_2)) \\
&= (f_1(s) \wedge x_2 = y_2) \wedge f_2(s) \wedge \rho_\theta \wedge \\
&\qquad\qquad \forall Y : ((f_1(s) \wedge \rho_\theta) \rightarrow \exists Y_2 : (f_2(s) \wedge true)) \\
&= f_1(s) \wedge f_2(s) \wedge \rho_\theta \wedge \forall Y : ((f_1(s) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s))) \wedge x_2 = y_2 \\
&= f^2_{kt}(s)
\end{aligned}
$$

□

PROOF OF THEOREM 6.

(1) The contract f of $\theta(I_1, I_2)$ is defined as $f(s) := f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta \wedge \forall Y_{\theta(I_1,I_2)} : \Phi$, where $\Phi := (f_1(s_1) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s_2))$. Because I_1 is Moore w.r.t. x_1, $f_1(s_1)$ does not refer to x_1. Because I_2 is disjoint from I_1, $f_2(s_2)$ does not refer to x_1 either. ρ_θ refers to outputs of I_1 and inputs of I_2, thus does not refer to x_1. Because none of $f_1(s_1)$, $f_2(s_2)$ or ρ_θ refer to x_1, Φ does not refer to x_1 either. Therefore, $f(s)$ does not refer to x_1, thus $\theta(I_1, I_2)$ is Moore w.r.t. x_1.

(2) By definition, the set of input variables of the composite interface $\theta(I_1, I_2)$ is $X_{\theta(I_1,I_2)} = (X_1 \cup X_2) \setminus \mathsf{InVars}(\theta) = X_1$. By hypothesis, I_1 is Moore w.r.t. all $x_1 \in X_1$. By part 1, $\theta(I_1, I_2)$ is also Moore w.r.t. all $x_1 \in X_1$, thus $\theta(I_1, I_2)$ is Moore.

(3) Since $x_2 \notin \mathsf{InVars}(\theta)$, x_2 is an input variable of $\theta(I_1, I_2)$ and ρ_θ does not refer to x_2. The result follows by a reasoning similar to that of part 1.

□

PROOF OF LEMMA 3.

(1) Proof is by induction on the length of states. Basis: the result holds for the empty state ε, because $\varepsilon \in f$ for any contract f. Induction step: let $s \cdot a \in f_\kappa$. Then $a \models f(s) \wedge x = y$, thus $a \models f(s)$. $s \cdot a \in f_\kappa$ implies $s \in f_\kappa$, thus, by the induction hypothesis, $s \in f$. This and $a \models f(s)$ imply $s \cdot a \in f$.

(2) Let $s \in f_\kappa$. Note that $\mathsf{in}(f_\kappa(s)) \equiv \mathsf{in}(f(s))$ is a formula over X: $\mathsf{in}(f_\kappa(s))$ is a formula over $X \setminus \{x\}$ and $\mathsf{in}(f(s))$ is a formula over X.
To show that $\mathsf{in}(f_\kappa(s)) \rightarrow \mathsf{in}(f(s))$ is valid, we need to show that every assignment over X that satisfies $\mathsf{in}(f_\kappa(s))$ also satisfies $\mathsf{in}(f(s))$. Consider such an assignment (a, p), where a is an assignment over $X \setminus \{x\}$ and p is an assignment over $\{x\}$. $(a, p) \models \mathsf{in}(f_\kappa(s))$ means $(a, p) \models \exists Y \cup \{x\} : f(s) \wedge x = y$. Therefore, there exists assignment b over $Y \cup \{x\}$ such that $(a, b) \models f(s) \wedge x = y$.

Let b' be the restriction of b to Y. We claim that $(a, p, b') \models f(s)$. Indeed, since I is Moore w.r.t. x, $f(s)$ does not depend on x, therefore, we can assign any value to x, in particular, the value assigned by p. $(a, p, b') \models f(s)$ implies $(a, p) \models \exists Y : f(s) \equiv \mathsf{in}(f(s))$.
To show that $\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f_\kappa(s))$ is valid, we need to show that every assignment over X that satisfies $\mathsf{in}(f(s))$ also satisfies $\mathsf{in}(f_\kappa(s))$. Consider such an assignment (a, p), where a is an assignment over $X \setminus \{x\}$ and p is an assignment over $\{x\}$. $(a, p) \models \mathsf{in}(f(s))$ means $(a, p) \models \exists Y : f(s)$. Therefore, there exists assignment b over Y such that $(a, p, b) \models f(s)$. Let p' be the assignment over $\{x\}$ such that $p'(x) := b(y)$. Since I is Moore w.r.t. x, $f(s)$ does not depend on x, therefore, $(a, p', b) \models f(s)$. Moreover, $(a, p', b) \models x = y$, therefore $(a, p', b) \models f(s) \wedge x = y \equiv f_\kappa(s)$. This implies $a \models \exists X \setminus \{x\} : f_\kappa(s) \equiv \mathsf{in}(f_\kappa(s))$. Therefore $(a, p) \models \mathsf{in}(f_\kappa(s))$.

□

Proof of Theorem 7. Let $I = (X, Y, f)$ and $\kappa = (y, x)$. Let $s \in f(\kappa(I))$. We must show that $f(s) \wedge x = y$ is satisfiable. By part 1 of Lemma 3, $s \in f$. Since I is well-formed, $f(s)$ is satisfiable. Let a be an assignment such that $a \models f(s)$. Consider the assignment a' which is identical to a, except that $a'(x) := a(y)$. Since I is Moore w.r.t. x, the satisfaction of $f(s)$ does not depend on the value x. Therefore, $a' \models f(s)$. Moreover, by definition, $a' \models x = y$, and the proof is complete. □

Proof of Lemma 4. Since z is redundant in f there exists $z' \neq z$ such that $\forall s \in f : \forall i \in \{1, ..., |s|\} : s_i(z) = s_i(z')$. Let $s, s' \in f$ such that $s =_{(X \cup Y) \setminus \{z\}} s'$. This means that for any $v \in X \cup Y$ if $v \neq z$ then $\forall i \in \{1, ..., |s|\} : s_i(v) = s'_i(v)$. But z' is such a v, therefore, $\forall i \in \{1, ..., |s|\} : s_i(z') = s'_i(z')$. Since $s_i(z') = s_i(z)$ and $s'_i(z') = s'_i(z)$ for all i, we get that $\forall i \in \{1, ..., |s|\} : s_i(z) = s'_i(z)$. Therefore, $s = s'$, which trivially implies $f(s) = f(s')$. □

Proof of Theorem 8.

—Let $I = (X, Y, f)$ be a well-formable interface. Then there exists $I' = (X, Y, f')$ such that I' is well-formed, and for all $s \in f'$, $f'(s) \equiv f(s) \wedge \phi_s$, where ϕ_s is some property over X. Slightly abusing notation, we define environment E with contract function $f_e(s) := \mathsf{in}(f'(s)) \equiv \mathsf{in}(f(s)) \wedge \phi_s$, for any state s. In this definition we implicitly use the mapping between variables of I and mirror variables of E. We claim that $I \leftrightarrows E$. Indeed, E is Moore, therefore $f_e(s)$ refers to no input variables, therefore $\mathsf{in}(f_e(s))$ has no free variables, thus it is equivalent to either true or false. E is well-formed, so $\mathsf{in}(f_e(s))$ must be true for all s. That is, E is input-complete. Also, $f_e(s) \rightarrow \mathsf{in}(f(s))$, therefore, any output of E is a legal input for I. Finally, the behavior of the closed-loop system of E and I is equivalent to I', therefore, it is well-formed.

—Conversely, suppose there exists environment E such that $I \leftrightarrows E$. We prove that I is well-formable. Let f_e be the contract function of E. Since I is not Moore, E must be Moore. Therefore, $f_e(s)$ is essentially a property over X for any s. We define $I' = (X, Y, f')$ such that $f'(s) := f(s) \wedge f_e(s)$. I' must be well-formed, because the closed-loop composition of I and E is well-formed. Thus, I' is a witness for I, which is well-formable.

□

Proof of Lemma 5. By induction on the length of states. Basis: $\varepsilon \in f'$. Induction step: suppose $s \cdot a \in f \cap f''$. Then $s \in f \cap f''$. From the induction hypothesis, $s \in f'$. $s \cdot a \in f \cap f''$ implies $a \models f(s) \wedge f''(s)$. $a \models f(s)$ implies $a \models \mathsf{in}(f(s))$. The latter and $I' \sqsubseteq I$ imply $a \models \mathsf{in}(f'(s))$. The latter, together with $I'' \sqsubseteq I'$ and $a \models f''(s)$, imply $a \models f'(s)$. This and $s \in f'$ imply $s \cdot a \in f'$. □

Proof of Theorem 9. $\sqsubseteq$ is reflexive because Condition 16 clearly holds when $f = f'$. To show that $\sqsubseteq$ is transitive, let $I = (X, Y, f)$, $I' = (X', Y', f')$, $I'' = (X'', Y'', f'')$, and suppose $I'' \sqsubseteq I'$ and $I' \sqsubseteq I$. We must prove $I'' \sqsubseteq I$. Suppose $s \in f \cap f''$. By Lemma 5, $s \in f \cap f'$ and $s \in f' \cap f''$. These facts together with $I'' \sqsubseteq I'$ and $I' \sqsubseteq I$ imply $\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f'(s))$, $\mathsf{in}(f(s)) \wedge f'(s) \rightarrow f(s)$, $\mathsf{in}(f'(s)) \rightarrow \mathsf{in}(f''(s))$, and $\mathsf{in}(f'(s)) \wedge f''(s) \rightarrow f'(s)$. These imply $\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f''(s))$ and $\mathsf{in}(f(s)) \wedge f''(s) \rightarrow f(s)$. To show that $\sqsubseteq$ is antisymmetric suppose $I' \sqsubseteq I$ and $I \sqsubseteq I'$. We must prove $I = I'$. By Lemma 5 and setting $I'' := I$ we get $f \subseteq f'$. By the same lemma and reversing the roles of I and I' we get $f' \subseteq f$. □

Proof of Theorem 10. Let $I = (X, Y, \phi)$ and $I' = (X', Y', \phi')$. I is well-formed, thus ϕ is satisfiable. Let a be an assignment satisfying ϕ and let a_X and a_Y be the restrictions of a to X and Y, respectively. By definition of $\mathsf{in}(\phi)$, $a_X \models \mathsf{in}(\phi)$. By Condition (17), $a_X \models \mathsf{in}(\phi') \equiv \exists Y' : \phi'$. Therefore, there exists $a'_{Y'}$ such that $(a_X, a'_{Y'}) \models \phi'$. Thus, ϕ' is satisfiable. Thus, I' is well-formed. □

Proof of Theorem 11. Let $I = (X, Y, f)$ and $I' = (X', Y', f')$.

(1) Suppose I is well-formed and $f' \subseteq f$. We need to show that for any $s \in f'$, $f'(s)$ is non-empty. By hypothesis, $s \in f$ and I is well-formed, therefore, $f(s)$ is non-empty. Reasoning as in the proof of Theorem 10, we can show that $f'(s)$ is also non-empty.

(2) This is a special case of part 1 of the theorem: I is source, therefore $f(s)$ refers to no input variables, therefore $\mathsf{in}(f(s))$ has no free variables, thus it is equivalent to either true or false. I is well-formed, so $\mathsf{in}(f(s))$ must be true for all s. That is, I is input-complete. This and the hypothesis $I' \sqsubseteq I$ imply that I' is also input-complete. For input-complete interfaces, $I' \sqsubseteq I$ is equivalent to $f' \subseteq f$. Therefore, part 1 applies.

(3) Suppose I is well-formable. Then there exists $I_1 = (X, Y, f_1)$ such that I_1 is well-formed, and for all $s \in f_1$, $f_1(s) \equiv f(s) \wedge \phi_s$, for some property ϕ_s over X. Since f_1 strengthens f, $f_1 \subseteq f$. Since $f(s) \wedge \phi_s \equiv f(s) \wedge \mathsf{in}(f(s)) \wedge \phi_s$, we can assume without loss of generality that $\phi_s \rightarrow \mathsf{in}(f(s))$. We define $I_2 := (X, Y, f_2)$ such that $f_2(s) := f'(s) \wedge \phi_s$, if $s \in f_1$, and $f_2(s) := f'(s)$, if $s \notin f_1$.
Claim 1: $f_2 \subseteq f_1$. By induction on the length of a state s. The result holds for $s = \varepsilon$. Suppose $s \cdot a \in f_2$. Then $s \in f_2$ and from the induction hypothesis, $s \in f_1$. Also, $a \models f_2(s) \equiv f'(s) \wedge \phi_s$ (because $s \in f_1$). Since $\phi_s \rightarrow \mathsf{in}(f(s))$, $a \models \mathsf{in}(f(s)) \wedge f'(s)$. This and $I' \sqsubseteq I$ imply $a \models f(s)$, thus, $a \models f(s) \wedge \phi_s \equiv f_1(s)$. Thus, $s \cdot a \in f_1$.

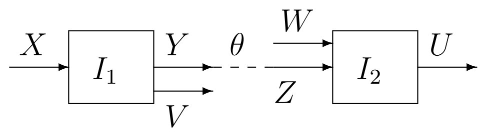

Fig. 16. Setting used in the proof of Theorem 12.

Claim 2: $f_2 \subseteq f'$. Because f_2 is a strengthening of f'.
Claim 3: $I_2 \sqsubseteq I_1$. Suppose $s \in f_1 \cap f_2$. By Claim 2 and the fact $f_1 \subseteq f$, we have $s \in f \cap f'$. Then: $\mathsf{in}(f_1(s)) \equiv \mathsf{in}(f(s)) \wedge \phi_s$. Since $I' \sqsubseteq I$ and $s \in f \cap f'$, $\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f'(s))$. Therefore $\mathsf{in}(f(s)) \wedge \phi_s \rightarrow \mathsf{in}(f'(s)) \wedge \phi_s$. The latter formula is equivalent to $\mathsf{in}(f_2(s))$ because $s \in f_1$. Also, $\mathsf{in}(f_1(s)) \wedge f_2(s) \equiv \mathsf{in}(f(s)) \wedge f'(s) \wedge \phi_s \rightarrow f(s) \wedge \phi_s \equiv f_1(s)$. This completes Claim 3.
Claim 4: for all $s \in f_2$, $f_2(s) \equiv f'(s) \wedge \phi_s$. Follows by definition of f_2 and Claim 1.
Claim 1 and Claim 3, together with the fact that I_1 is well-formed, and by the part 1 of this theorem, imply that I_2 is well-formed. Claim 4 implies that I_2 is a witness for I', thus, I' is well-formable.

□

Proof of Lemma 6. Let $f := f(\theta(I_1, I_2))$. Proof is by induction on the length of states. Basis: the result holds for ε. Induction step: Let $s_1 \cdot a_1 \in f_1$. This means that there exists state $s \cdot a \in f$ such that $s_1 \cdot a_1$ is the projection of $s \cdot a$ to the variables of I_1. From $s \cdot a \in f$, we get $a \models f(s)$ i.e. $a \models f_1(s_1) \wedge f_2(s_2) \wedge \cdots$. Therefore, $a \models f_1(s_1)$, which means $a_1 \models f_1(s_1)$. By the induction hypothesis, $s_1 \in f(I_1)$. These two facts imply $s_1 \cdot a \in f(I_1)$. This proves $f_1 \subseteq f(I_1)$. The proof of $f_2 \subseteq f(I_2)$ is similar. □

Proof of Theorem 12. Let $I_1 = (X, Y \cup V, f_1)$ and $I_2 = (Z \cup W, U, f_2)$, so that $Y \cap V = Z \cap W = \emptyset$, $Z = \mathsf{InVars}(\theta)$ and $Y = \{y \mid \exists (y, x) \in \theta\}$. In other words, Y represents the set of output variables of I_1 that are connected to input variables of I_2. V is the set of the rest of the output variables of I_1. Z represents those input variables of I_2 that are connected to outputs of I_1 and W those that are not connected. Any of the sets X, Y, V, Z, W, U may be empty. Let $I_1' = (X, Y \cup V, f_1')$ and $I_2' = (Z \cup W, U, f_2')$. The composition setting is illustrated in Figure 16.

Given the above, and Definition 9, we have, for $s \in \mathcal{A}(X \cup W \cup Y \cup V \cup Z \cup U)^*$, s_1 the projection of s to $X \cup Y \cup V$, and s_2 the projection of s to $W \cup Z \cup U$:

$$\theta(I_1, I_2) := (X \cup W, Y \cup V \cup Z \cup U, f) \tag{29}$$

$$f(s) := f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta \wedge \Psi \tag{30}$$

$$\Psi := \forall Y \cup V \cup Z \cup U : (f_1(s_1) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s_2)) \tag{31}$$

$$\theta(I_1', I_2') := (X \cup W, Y \cup V \cup Z \cup U, f') \tag{32}$$

$$f'(s) := f_1'(s_1) \wedge f_2'(s_2) \wedge \rho_\theta \wedge \Psi' \tag{33}$$

$$\Psi' := \forall Y \cup V \cup Z \cup U : (f_1'(s_1) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2'(s_2)) \tag{34}$$

Let $s \in f \cap f'$. To prove $\theta(I_1', I_2') \sqsubseteq \theta(I_1, I_2)$ we need to prove that: (A) $\mathsf{in}(f(s)) \rightarrow$

$\mathsf{in}(f'(s))$ is valid; and (B) $(\mathsf{in}(f(s)) \wedge f'(s)) \rightarrow f(s)$ is valid. Note that, by Lemma 6, $s_1 \in f_1 \cap f_1'$ and $s_2 \in f_2 \cap f_2'$. We use these two facts without mention in the rest of the proof. We proceed in proving claims (A) and (B).

(A): $\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f'(s))$ is valid: Suppose the result does not hold. This means that $\mathsf{in}(f(s)) \wedge \neg\mathsf{in}(f'(s))$ is satisfiable, i.e.,

$$\begin{aligned}\psi_1 \ := \ & (\exists Y \cup V \cup Z \cup U : f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta \wedge \Psi) \\ & \wedge(\forall Y \cup V \cup Z \cup U : \neg f_1'(s_1) \vee \neg f_2'(s_2) \vee \neg\rho_\theta \vee \neg\Psi')\end{aligned}$$

is satisfiable. Note that ψ_1, Ψ and Ψ' are all formulae over $X \cup W$, therefore, ψ_1 is equivalent to:

$$\begin{aligned}\psi_2 \ := \ & \Psi \wedge (\exists Y \cup V \cup Z \cup U : f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta) \\ & \wedge\Big(\neg\Psi' \vee (\forall Y \cup V \cup Z \cup U : \neg f_1'(s_1) \vee \neg f_2'(s_2) \vee \neg\rho_\theta)\Big)\end{aligned}$$

Let a be an assignment over $X \cup W$ satisfying ψ_2. We claim that $a \models \neg\Psi'$. Suppose not, i.e., $a \models \Psi'$. Then, from $a \models \psi_2$, we derive $a \models \forall Y \cup V \cup Z \cup U : \neg f_1'(s_1) \vee \neg f_2'(s_2) \vee \neg\rho_\theta$. Also, $a \models \mathsf{in}(f_1(s_1))$. Since $I_1' \sqsubseteq I_1$, $a \models \mathsf{in}(f_1'(s_1))$. This means that there exists an assignment c over $Y \cup V$ such that $(a, c) \models f_1'(s_1)$. Let d be an assignment over Z such that $(c, d) \models \rho_\theta$: that is, we set an input variable z of I_2 to the value $c(y)$ of the output variable y of I_1 that z is connected to. Combining, we have $(a, c, d) \models f_1'(s_1) \wedge \rho_\theta$. This and $a \models \Psi'$ imply that $(a, c, d) \models \mathsf{in}(f_2'(s_2))$. Therefore, there exists an assignment e over U such that $(a, c, d, e) \models f_2'(s_2)$. Combining, we have $(a, c, d, e) \models f_1'(s_1) \wedge f_2'(s_2) \wedge \rho_\theta$, which contradicts $a \models \forall Y \cup V \cup Z \cup U : \neg f_1'(s_1) \vee \neg f_2'(s_2) \vee \neg\rho_\theta$. Thus, the claim $a \models \neg\Psi'$ is proven and we have that a satisfies:

$$\psi_3 := \Psi \wedge \neg\Psi' \wedge (\exists Y \cup V \cup Z \cup U : f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta)$$

Since a does not satisfy Ψ', there exists an assignment b over $Y \cup V \cup Z \cup U$, such that $(a, b) \models f_1'(s_1) \wedge \rho_\theta \wedge \neg\mathsf{in}(f_2'(s_2))$. Since $I_2' \sqsubseteq I_2$, $\mathsf{in}(f_2(s_2)) \rightarrow \mathsf{in}(f_2'(s_2))$, or $\neg\mathsf{in}(f_2'(s_2)) \rightarrow \neg\mathsf{in}(f_2(s_2))$. Therefore, $(a, b) \models \neg\mathsf{in}(f_2(s_2))$. Now, from $a \models \psi_3$, we get $(a, b) \models \mathsf{in}(f_1(s_1))$. From $I_1' \sqsubseteq I_1$ we have $\mathsf{in}(f_1(s_1)) \wedge f_1'(s_1) \rightarrow f_1(s_1)$. Therefore, $(a, b) \models f_1(s_1)$. This, together with $a \models \Psi$ and $(a, b) \models \rho_\theta$, imply $(a, b) \models \mathsf{in}(f_2(s_2))$. Contradiction. This completes the proof of Part (A).

(B): $(\mathsf{in}(f(s)) \wedge f'(s)) \rightarrow f(s)$ is valid: Suppose the result does not hold. This means that $\mathsf{in}(f(s)) \wedge f'(s) \wedge \neg f(s)$ is satisfiable, i.e.,

$$\begin{aligned}\psi_4 \ := \ & (\exists Y \cup V \cup Z \cup U : f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta \wedge \Psi) \\ & \wedge(f_1'(s_1) \wedge f_2'(s_2) \wedge \rho_\theta \wedge \Psi') \wedge (\neg f_1(s_1) \vee \neg f_2(s_2) \vee \neg\rho_\theta \vee \neg\Psi)\end{aligned}$$

is satisfiable. Because Ψ and Ψ' are formulae over $X \cup W$, ψ_4 simplifies to:

$$\begin{aligned}\psi_5 \ := \ & \Psi \wedge \Psi' \wedge (\exists Y \cup V \cup Z \cup U : f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta) \\ & \wedge(f_1'(s_1) \wedge f_2'(s_2) \wedge \rho_\theta) \wedge (\neg f_1(s_1) \vee \neg f_2(s_2))\end{aligned}$$

Let a be an assignment over $X \cup W$ such that $a \models \psi_5$. Then $a \models \mathsf{in}(f_1(s_1)) \wedge \mathsf{in}(f_2(s_2)) \wedge f_1'(s_1) \wedge f_2'(s_2)$. From the hypotheses $I_1' \sqsubseteq I_1$ and $I_2' \sqsubseteq I_2$, we get $\mathsf{in}(f_1(s_1)) \wedge f_1'(s_1) \rightarrow f_1(s_1)$ and $\mathsf{in}(f_2(s_1)) \wedge f_2'(s_2) \rightarrow f_2(s_2)$. Therefore $a \models f_1(s_1) \wedge f_2(s_2)$, which contradicts $a \models \psi_5$. This completes the proof of Part (B) and of the theorem. □

Proof of Theorem 13. Let $I = (X, Y, f)$. Because $I' \sqsubseteq I$, $I' = (X, Y, f')$ for some f'. Then: $\kappa(I) = (X \setminus \{x\}, Y \cup \{x\}, f_\kappa)$ and $\kappa(I') = (X \setminus \{x\}, Y \cup \{x\}, f'_\kappa)$, where $f_\kappa(s) := f(s) \wedge x = y$ and $f'_\kappa(s) := f'(s) \wedge x = y$, for all $s \in \mathcal{A}(X \cup Y)^*$. To show that $\kappa(I') \sqsubseteq \kappa(I)$, we need to prove that for any $s \in f_\kappa \cap f'_\kappa$, the following formulae are valid:

$$\mathsf{in}(f_\kappa(s)) \to \mathsf{in}(f'_\kappa(s))$$
$$\big(\mathsf{in}(f_\kappa(s)) \wedge f'_\kappa(s)\big) \to f_\kappa(s)$$

By part 1 of Lemma 3, $s \in f_\kappa \cap f'_\kappa$ implies $s \in f \cap f'$. By part 2 of Lemma 3, $\mathsf{in}(f_\kappa(s)) \equiv \mathsf{in}(f(s))$ and $\mathsf{in}(f'(s)) \equiv \mathsf{in}(f'_\kappa(s))$. This and $\mathsf{in}(f(s)) \to \mathsf{in}(f'(s))$ imply $\mathsf{in}(f_\kappa(s)) \to \mathsf{in}(f'_\kappa(s))$. Moreover:

$$\big(\mathsf{in}(f_\kappa(s)) \wedge f'_\kappa(s)\big) \equiv \big(\mathsf{in}(f(s)) \wedge f'(s) \wedge x = y\big) \to (f(s) \wedge x = y) \equiv f_\kappa(s)$$

□

Proof of Theorem 14. Recall that $\mathsf{hide}(y, I_i) = (X, Y \setminus \{y\}, f'_i)$, such that for any $s \in \mathcal{A}(X \cup Y \setminus \{y\})^*$, $f'_i(s) \equiv \exists y : f_i(s)$. To show $\mathsf{hide}(y, I_2) \sqsubseteq \mathsf{hide}(y, I_1)$ we need to show that for any $s \in f'_1 \cap f'_2$, we have $\mathsf{in}(f'_1(s)) \to \mathsf{in}(f'_2(s))$ and $\mathsf{in}(f'_1(s)) \wedge f'_2(s) \to f'_1(s)$. The first proof obligation becomes $(\exists Y \setminus \{y\} : \exists y : f_1(s)) \to (\exists Y \setminus \{y\} : \exists y : f_2(s))$, or equivalently, $\mathsf{in}(f_1(s)) \to \mathsf{in}(f_2(s))$, which holds by hypothesis $I_2 \sqsubseteq I_1$. Note that although s is a state in $\mathcal{A}(X \cup Y \setminus \{y\})^*$, we can write $f_1(s)$ and $f_2(s)$, because both f_1 and f_2 are independent from y.

The second proof obligation becomes $(\exists Y \setminus \{y\} : \exists y : f_1(s)) \wedge (\exists y : f_2(s)) \to (\exists y : f_1(s))$, or equivalently, $\mathsf{in}(f_1(s)) \wedge (\exists y : f_2(s)) \to (\exists y : f_1(s))$. Let $(a_X, a_{Y\setminus\{y\}}) \in \mathcal{A}(X \cup Y \setminus \{y\})$ be such that $a_X \in \mathsf{in}(f_1(s))$ and $(a_X, a_{Y\setminus\{y\}}) \in (\exists y : f_2(s))$. Then there exists $a_Y \in \mathcal{A}(Y)$ such that $(a_X, a_Y) \in f_2(s)$ and $a_{Y\setminus\{y\}}$ is the projection of a_Y to $Y \setminus \{y\}$. From hypothesis $I_2 \sqsubseteq I_1$, it must be that $(a_X, a_Y) \in f_1(s)$. Therefore $(a_X, a_{Y\setminus\{y\}}) \in (\exists y : f_1(s))$. □

Proof of Theorem 15.

(1) Suppose $I' \sqsubseteq I$ and let E be an environment such that $I \leftrightarrows E$. We prove that $I' \leftrightarrows E$. Clearly, E is an environment for I', since the input and output variables of I' are the same as those of I. We distinguish cases:
 —E is Moore. Then we must prove that $K(\theta(E, I'))$ is well-formed, assuming that $K(\theta(E, I))$ is well-formed. By Theorems 12 and 13, $K(\theta(E, I')) \sqsubseteq K(\theta(E, I))$. Both $K(\theta(E, I))$ and $K(\theta(E, I'))$ are source interfaces, therefore, by part 2 of Theorem 11, $K(\theta(E, I'))$ is well-formed.
 —E is not Moore, therefore I is Moore. Then we must prove that $K(\theta(I', E))$ is well-formed, assuming that $K(\theta(I, E))$ is well-formed. The argument is similar to the previous case.

(2) Let $I = (X, Y, f)$ and $I' = (X', Y', f')$ and suppose $I' \not\sqsubseteq I$. If $X \neq X'$ or $Y \neq Y'$ then we can find, by Theorem 8, environment E for I such that $I \leftrightarrows E$, and E is not an environment for I', thus $I' \not\leftrightarrows E$. We concentrate on the case $X = X'$ and $Y = Y'$. Then $I' \not\sqsubseteq I$ means there exists $s \in f \cap f'$ such that Condition (16) does not hold. Define environment E for I with contract function f_e where $f_e(r) := \mathsf{in}(f(r))$ for all states r. (Again we are slightly abusing notation: $\mathsf{in}(f(r))$ is a property over X, but $f_e(r)$ is a property over

$\hat{X}$, the output variables of E.) By definition, E is Moore. Because I is well-formed, $I \leftrightarrows E$. We claim that $I' \not\leftrightarrows E$. We distinguish cases:

—$\mathsf{in}(f(s)) \not\to \mathsf{in}(f'(s))$: Observe that, in the contract of the connection of E and I', the term Φ of (11) evaluates to false at state s: this is because $f_e(s) \not\to \mathsf{in}(f'(s))$. Therefore, the entire contract of the connection is also false at s, which means that the connection of I' and E is not well-formed.

—$\mathsf{in}(f(s)) \to \mathsf{in}(f'(s))$ but $\mathsf{in}(f(s)) \wedge f'(s) \not\to f(s)$: At state s, there exists input $a_X \in \mathsf{in}(f(s)) = f_e(s)$, for which I' can produce output a_Y such that $a := (a_X, a_Y) \in f'(s) \setminus f(s)$. Since $a \notin f(s)$, $f(s \cdot a)$ is empty, thus $f_e(s \cdot a) \equiv \mathsf{false}$, thus, again, the composition of I' with E is not well-formed.

□

PROOF OF LEMMA 7. Let $I = (X, Y, f)$ and $I' = (X', Y', f')$.

$f \cap f' \subseteq f(I \sqcap I')$: By induction on the length of states. It holds for the state of length zero, i.e., the empty state ε, because ε is reachable in any interface. Suppose $s \cdot a \in f \cap f'$. Then $s \in f \cap f'$, and from the induction hypothesis, $s \in f(I \sqcap I')$. Since $s \cdot a \in f$, $a \models f(s)$. Since $s \cdot a \in f'$, $a \models f'(s)$. Thus $a \models f(s) \wedge f'(s)$. Thus $a \models (\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))) \wedge (\mathsf{in}(f(s)) \to f(s)) \wedge (\mathsf{in}(f'(s)) \to f'(s)) \equiv f_\sqcap(s)$.

$f(I \sqcap I') \subseteq f \cup f'$: By induction on the length of states. Basis: It holds for the empty state ε. Induction step: Suppose $s \cdot a \in f(I \sqcap I')$. Then $a \models f_\sqcap(s)$. Also, $s \in f(I \sqcap I')$, and from the induction hypothesis, $s \in f \cup f'$. Suppose $s \in f$ (the other case is symmetric). There are two sub-cases:

Case 1: $s \in f'$: Then $f_\sqcap(s) \equiv (\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))) \wedge (\mathsf{in}(f(s)) \to f(s)) \wedge (\mathsf{in}(f'(s)) \to f'(s))$. Since $a \models f_\sqcap(s)$, $a \models (\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s)))$. Suppose $a \models \mathsf{in}(f(s))$ (the other case is symmetric). Then, since $a \models \mathsf{in}(f(s)) \to f(s)$, we have $a \models f(s)$, thus, $s \cdot a \in f$.

Case 2: $s \notin f'$: Then $f_\sqcap(s) \equiv f(s)$, therefore, $a \models f(s)$, thus, $s \cdot a \in f$. □

PROOF OF LEMMA 8. By Lemma 7, $s \in f_\sqcap$. Using the fact that $\mathsf{in}(f(s))$ and $\mathsf{in}(f'(s))$ are properties over X, and the fact that the existential quantifier distributes over disjunctions, we can show the following equivalences:

$$\mathsf{in}(f_\sqcap(s)) \equiv$$
$$\exists Y : \Big(\big(\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))\big) \wedge \big(\mathsf{in}(f(s)) \to f(s)\big) \wedge \big(\mathsf{in}(f'(s)) \to f'(s)\big) \Big) \equiv$$
$$\big(\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))\big) \wedge \exists Y : \Big(\big(\neg\mathsf{in}(f(s)) \vee f(s)\big) \wedge \big(\neg\mathsf{in}(f'(s)) \vee f'(s)\big) \Big)$$

Clearly, the last formula implies $\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))$. For the converse, we need to show that $\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))$ implies

$$\exists Y : \big(\neg\mathsf{in}(f(s)) \vee f(s)\big) \wedge \big(\neg\mathsf{in}(f'(s)) \vee f'(s)\big) \tag{35}$$

Now, $\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))$ is equivalent to

$$\Big(\neg\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s))\Big) \vee \Big(\mathsf{in}(f(s)) \wedge \neg\mathsf{in}(f'(s))\Big) \vee \Big(\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s))\Big)$$

and clearly both the first two disjuncts imply Condition (35). It remains to show that $\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s))$ implies Condition (35). This holds thanks to shared-refinability Condition (24). □

PROOF OF THEOREM 17. Since I and I' are shared-refinable, they have the same sets of input and output variables. Let $I = (X, Y, f)$ and $I' = (X, Y, f')$. Let $I \sqcap I' = (X, Y, f_\sqcap)$. To prove $(I \sqcap I') \sqsubseteq I$, we need to show that for all $s \in f_\sqcap \cap f$:

$$\mathsf{in}(f(s)) \to \mathsf{in}(f_\sqcap(s))$$
$$\big(\mathsf{in}(f(s)) \wedge f_\sqcap(s)\big) \to f(s)$$

The first condition follows from Lemma 8 and the second by definition of $f_\sqcap$. The proof for $(I \sqcap I') \sqsubseteq I'$ is symmetric. Thus, $I \sqcap I'$ is a lower bound of I and I'.

To show that $I \sqcap I'$ is the *greatest* lower bound, let $I'' = (X, Y, f'')$. To prove $I'' \sqsubseteq (I \sqcap I')$ we must prove $\mathsf{in}(f_\sqcap(s)) \to \mathsf{in}(f''(s))$ and $\mathsf{in}(f_\sqcap(s)) \wedge f''(s) \to f_\sqcap(s)$. By Lemma 8 and the definition of $f_\sqcap$, these conditions become:

$$\big(\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))\big) \ \to\ \mathsf{in}(f''(s))$$
$$\Big(\big(\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))\big) \wedge f''(s)\Big) \ \to$$
$$\Big(\big(\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))\big) \wedge \big(\mathsf{in}(f(s)) \to f(s)\big) \wedge \big(\mathsf{in}(f'(s)) \to f'(s)\big)\Big)$$

From hypotheses $I'' \sqsubseteq I$ and $I'' \sqsubseteq I'$ we get $\mathsf{in}(f(s)) \to \mathsf{in}(f''(s))$ and $\mathsf{in}(f'(s)) \to \mathsf{in}(f''(s))$, from which the first condition follows. We also get $\mathsf{in}(f(s)) \wedge f''(s) \to f(s)$ and $\mathsf{in}(f'(s)) \wedge f''(s) \to f'(s)$, therefore,

$$\big(\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))\big) \wedge f''(s) \to \big(f(s) \wedge f'(s)\big),$$

from which the second condition follows. □

PROOF OF THEOREM 18. Let $I = (X, Y, f)$, $I' = (X, Y, f')$ and $I \sqcap I' = (X, Y, f_\sqcap)$. Let $s \in f_\sqcap$. By Lemma 7, $s \in f \cup f'$. Suppose $s \in f$. By hypothesis, $f(s) \neq \emptyset$. Let $a \in f(s)$ and $a = (a_X, a_Y)$ where $a_X \in \mathcal{A}(X)$ and $a_Y \in \mathcal{A}(Y)$. Clearly, $a_X \in \mathsf{in}(f(s))$. If $a_X \notin \mathsf{in}(f'(s))$ then a clearly satisfies Formula (25), thus $a \in f_\sqcap(s)$. If $a_X \in \mathsf{in}(f'(s))$ then $a_X \in \mathsf{in}(f(s)) \cap \mathsf{in}(f'(s))$, therefore, by shared-refinability Condition (24), there must exist $a'_Y \in \mathcal{A}(Y)$ such that $(a_X, a'_Y) \in f(s) \cap f'(s)$. Then (a_X, a'_Y) clearly satisfies Formula (25), thus $(a_X, a'_Y) \in f_\sqcap(s)$. The case $s \in f'$ is symmetric. □

PROOF OF LEMMA 9. Let $I = (X, Y, f)$, $I' = (X, Y, f')$ and $I \sqcup I' = (X, Y, f_\sqcup)$. We prove $f_\sqcup \subseteq f \cup f'$ by induction on the length of states. Basis: it holds for ε. Step: let $s \cdot a \in f_\sqcup$. Then $a \in f_\sqcup(s)$. Thus $s \in f_\sqcup$ and from the induction hypothesis, $s \in f \cup f'$. There are three cases:

—$s \in f \cap f'$: Then $a \models \mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge \big(f(s) \vee f'(s)\big)$, thus, $a \in f(s) \cup f'(s)$. Thus $s \cdot a \in f \cup f'$.

—$s \in f \setminus f'$: Then $a \models f(s)$, thus $s \cdot a \in f$.

—$s \in f' \setminus f$: Then $a \models f'(s)$, thus $s \cdot a \in f'$.

The proof $f \cap f' \subseteq f_\sqcup$ is also by induction. Let $s \cdot a \in f \cap f'$. Then $a \in f(s) \cap f'(s)$, so $s \in f \cap f'$. Clearly then, $a \models f_\sqcup(s)$, thus $s \cdot a \in f_\sqcup$. □

PROOF OF THEOREM 19. Let $I = (X, Y, f)$, $I' = (X, Y, f')$ and $I \sqcup I' = (X, Y, f_\sqcup)$. Consider $s \in f \cap f_\sqcup$. There are two cases:

—$s \in f'$: Then

$$\mathsf{in}(f_\sqcup(s)) \equiv \mathsf{in}\Big(\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge (f(s) \vee f'(s))\Big) \equiv$$
$$\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge \mathsf{in}(f(s) \vee f'(s)) \equiv$$
$$\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge \Big(\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))\Big) \equiv \mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s))$$

and the refinement conditions for $I \sqsubseteq (I \sqcup I')$ become

$$\big(\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s))\big) \;\rightarrow\; \mathsf{in}(f(s))$$
$$\big(\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge f(s)\big) \;\rightarrow\; \Big(\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge \big(f(s) \vee f'(s)\big)\Big)$$

which clearly hold.

—$s \notin f'$: Then $\mathsf{in}(f_\sqcup(s)) \equiv \mathsf{in}(f(s))$, and the refinement conditions for $I \sqsubseteq (I \sqcup I')$ become $\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f(s))$ and $\mathsf{in}(f(s)) \wedge f(s) \rightarrow f(s)$, which clearly hold.

This proves $I \sqsubseteq (I \sqcup I')$. Similarly we show $I' \sqsubseteq (I \sqcup I')$.

Now, let $I'' = (X, Y, f'')$ and consider $s \in f_\sqcup \cap f''$. By Lemma 9, $s \in (f \cup f') \cap f''$. To show $(I \sqcup I') \sqsubseteq I''$, we need to show $\mathsf{in}(f''(s)) \rightarrow \mathsf{in}(f_\sqcup(s))$ and $\mathsf{in}(f''(s)) \wedge f_\sqcup(s) \rightarrow f''(s)$. We reason by cases:

—$s \in f \cap f' \cap f''$: then the proof obligations above become: $\mathsf{in}(f''(s)) \rightarrow \mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s))$ and $\mathsf{in}(f''(s)) \wedge \mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge \big(f(s) \vee f'(s)\big) \rightarrow f''(s)$. From hypotheses $s \in f \cap f'$, $I \sqsubseteq I''$ and $I' \sqsubseteq I''$ we get $\mathsf{in}(f''(s)) \rightarrow \mathsf{in}(f(s))$ and $\mathsf{in}(f''(s)) \rightarrow \mathsf{in}(f'(s))$, from which the first condition follows. We also get $\mathsf{in}(f''(s)) \wedge f(s) \rightarrow f''(s)$ and $\mathsf{in}(f''(s)) \wedge f'(s) \rightarrow f''(s)$, therefore, $\mathsf{in}(f''(s)) \wedge \big(f(s) \vee f'(s)\big) \rightarrow f''(s)$, from which the second condition follows.

—$s \in (f \setminus f') \cap f''$: then the proof obligations become: $\mathsf{in}(f''(s)) \rightarrow \mathsf{in}(f(s))$ and $\mathsf{in}(f''(s)) \wedge f(s) \rightarrow f''(s)$, which hold from hypotheses $s \in f \cap f''$ and $I \sqsubseteq I''$.

—$s \in (f' \setminus f) \cap f''$: similar to the previous case.

□

Proof of Theorem 20. Let I be a well-formed interface with contract f. If I is Moore then $f(s)$ refers to no input variables, therefore, $\mathsf{in}(f(s))$ has no free variables, thus, it is equivalent to either true or false. I is well-formed, so $\mathsf{in}(f(s))$ must be true for all s. □

Proof of Theorem 21. Let $I = (X, Y, f)$ be an input-complete interface. Then $\mathsf{in}(f(s))$ is valid for all $s \in \mathcal{A}(X \cup Y)^*$, i.e., $\exists Y : f(s) \equiv \mathsf{true}$ for any assignment over X. Let a_X be an assignment over X (note that a_X is defined even when X is empty). Then there exists an assignment a_Y on Y such that the combined assignment (a_X, a_Y) on $X \cup Y$ satisfies $f(s)$. Thus, $f(s)$ is satisfiable, which means I is well-formed. □

Proof of Theorem 22. Let $I = (X, Y, f)$ and $\mathsf{IC}(I) = (X, Y, f_{ic})$. Let $s \in \mathcal{A}(X \cup Y)^*$.

(1) $\mathsf{in}(f_{ic}(s)) \equiv \exists Y : (f(s) \vee \neg\mathsf{in}(f(s))) \equiv (\exists Y : f(s)) \vee \neg\mathsf{in}(f(s)) \equiv \mathsf{in}(f(s)) \vee \neg\mathsf{in}(f(s)) \equiv \mathsf{true}$, thus, $\mathsf{IC}(I)$ is input-complete.

(2) Obviously, $\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f_{ic}(s))$. We need to show that $(\mathsf{in}(f(s)) \wedge (f(s) \vee \neg\mathsf{in}(f(s)))) \rightarrow f(s)$. The premise can be rewritten as $(\mathsf{in}(f(s)) \wedge f(s)) \vee (\mathsf{in}(f(s)) \wedge \neg\mathsf{in}(f(s))) \equiv \mathsf{in}(f(s)) \wedge f(s)$, which clearly implies $f(s)$.

□

Proof of Theorem 23. In f, the term Φ defined in Formula (11) is equivalent to true because $\mathsf{in}(f_2(s_2)) \equiv \mathsf{true}$. To see that $\theta(I_1, I_2)$ is input-complete, consider a state $s \in \mathcal{A}(X_{\theta(I_1,I_2)} \cup Y_{\theta(I_1,I_2)})^*$ and let a be an assignment over $X_{\theta(I_1,I_2)}$. Since $\mathsf{in}(f_1(s_1)) \equiv \mathsf{true}$, and $X_1 \subseteq X_{\theta(I_1,I_2)}$, there exists an assignment b over Y_1 such that $(a, b) \models f_1(s_1)$. Let c be an assignment over $\mathsf{InVars}(\theta)$ such that $(b, c) \models \rho_\theta$: such an assignment can always be found by setting $c(x)$ to the value that b assigns to y, where $(y, x) \in \theta$. Since $\mathsf{in}(f_2(s_2)) \equiv \mathsf{true}$, there exists an assignment d over Y_2 such that $(a, c, d) \models f_2(s_2)$. Combining the assignments we get $(a, b, c, d) \models f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta \equiv f(s)$, therefore, $\theta(I_1, I_2)$ is input-complete. □

Proof of Theorem 24. By definition, $\kappa(I) = (X \setminus \{x\}, Y \cup \{x\}, f_\kappa)$, where $f_\kappa(s) \equiv f(s) \wedge (x = y)$, for all $s \in \mathcal{A}(X \cup Y)^*$. Let $s \in \mathcal{A}(X \cup Y)^*$. We must show that $\mathsf{in}(f_\kappa(s)) \equiv \exists Y \cup \{x\} : f(s) \wedge (x = y)$ is valid. Because $f(s)$ does not refer to x, we have $\exists Y \cup \{x\} : f(s) \wedge (x = y) \equiv \exists Y : \exists x : f(s) \wedge (x = y) \equiv \exists Y : (f(s) \wedge (\exists x : x = y)) \equiv \exists Y : f(s) \equiv \mathsf{in}(f(s)) \equiv \mathsf{true}$. □

Proof of Theorem 25. I is input-complete means $\mathsf{in}(f(s))$ is valid for all $s \in \mathcal{A}(X \cup Y)^*$. We must show that $\exists Y \setminus \{y\} : (\exists y : f(s))$ is valid: the latter formula is equivalent to $\exists Y : f(s)$, i.e., $\mathsf{in}(f(s))$. □

Proof of Theorem 29. Following Definition 9, it suffices to prove that the formula

$$(f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta) \rightarrow \Big(\forall Y_{\theta(I_1,I_2)} : (f_1(s_1) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s_2))\Big)$$

is valid for any s_1, s_2. Let $a \in \mathcal{A}(X_1 \cup Y_1 \cup X_2 \cup Y_2)$ such that $a \models f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta$. We need to prove that $a \models \forall Y_{\theta(I_1,I_2)} : (f_1(s_1) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s_2))$. Let $b \in \mathcal{A}(Y_{\theta(I_1,I_2)})$ such that $(a|b) \models f_1(s_1) \wedge \rho_\theta$. Here, $(a|b)$ denotes the assignment obtained by replacing in a the values of all variables of b (i.e., variables in $Y_{\theta(I_1,I_2)}$) by the values assigned to them by b. We need to prove that $(a|b) \models \mathsf{in}(f_2(s_2))$. Observe that, because $X_1 \cap Y_{\theta(I_1,I_2)} = \emptyset$, for all $x_1 \in X_1$, we have $a(x_1) = (a|b)(x_1)$. This and the fact that I_1 is deterministic imply that for all $y_1 \in Y_1$, we have $a(y_1) = (a|b)(y_1)$. This and the facts $a \models \rho_\theta$ and $(a|b) \models \rho_\theta$ imply that for all $x_2 \in \mathsf{InVars}(\theta)$, we have $a(x_2) = (a|b)(x_2)$. Finally observe that, because $(X_2 \setminus \mathsf{InVars}(\theta)) \cap Y_{\theta(I_1,I_2)} = \emptyset$, for all $x_2' \in X_2 \setminus \mathsf{InVars}(\theta)$, we have $a(x_2') = (a|b)(x_2')$. Collecting the last two results, we get that for all $x_2 \in X_2$, we have $a(x_2) = (a|b)(x_2)$. This and $a \models f_2(s_2)$ imply $(a|b) \models \mathsf{in}(f_2(s_2))$. □

Proof of Theorem 30. Recall that $\mathsf{hide}(y, I) = (X, Y \setminus \{y\}, f')$, such that for any $s \in \mathcal{A}(X \cup Y \setminus \{y\})^*$, $f'(s) \equiv \exists y : f(s)$. If $Y = \{y\}$ then $\mathsf{hide}(y, I)$ is a sink, therefore, deterministic by Theorem 28. Otherwise, let $s \in f'$ and let $a_X \in \mathsf{in}(f'(s)) \equiv \exists Y \setminus \{y\} : \exists y : f(s) \equiv \mathsf{in}(f(s))$. Since I is deterministic, there is a unique $a_Y \in \mathcal{A}(Y)$ such that $(a_X, a_Y) \in f(s)$. Therefore, there is a unique $a_{Y\setminus\{y\}} \in \mathcal{A}(Y \setminus \{y\})$, where $a_{Y\setminus\{y\}}$ is the projection of a_Y to $Y \setminus \{y\}$, such that $(a_X, a_{Y\setminus\{y\}}) \in f'(s)$, which proves determinism of $\mathsf{hide}(y, I)$. □

Proof of Theorem 31. Let $I = (X, Y, f)$ and $I' = (X, Y, f')$.

First, suppose $I' \sqsubseteq I$. To prove $f \subseteq f'$, it suffices to show that for all $s \in f$, $f(s) \to f'(s)$ is valid. Let $a \in \mathcal{A}(X \cup Y)$ such that $a \in f(s)$. Let $a = (a_X, a_Y)$ where $a_X \in \mathcal{A}(X)$ and $a_Y \in \mathcal{A}(Y)$. Then $a_X \in \mathsf{in}(f(s))$, and by Definition 16, $a_X \in \mathsf{in}(f'(s))$. Therefore there exists $a'_Y \in \mathcal{A}(Y)$ such that $(a_X, a'_Y) \in f'(s)$. By Definition 16, $(a_X, a'_Y) \in f(s)$. Since I is deterministic, $a'_Y = a_Y$. Thus, $a = (a_X, a_Y) \in f'(s)$.

Conversely, suppose $f \subseteq f'$. To prove $I' \sqsubseteq I$, it suffices to show that for all $s \in f$, the formulas $\mathsf{in}(f(s)) \to \mathsf{in}(f'(s))$ and $\mathsf{in}(f(s)) \wedge f'(s) \to f(s)$ are valid. Let $a_X \in \mathsf{in}(f(s))$. Then there exists $a_Y \in \mathcal{A}(Y)$ such that $a := (a_X, a_Y) \in f(s)$. Thus, $s \cdot a \in f$, and by hypothesis, $s \cdot a \in f'$, therefore, $a \in f'(s)$. This implies $a_X \in \mathsf{in}(f'(s))$. This proves $\mathsf{in}(f(s)) \to \mathsf{in}(f'(s))$. Now consider $(a_X, a'_Y) \in f'(s)$ such that $a_X \in \mathsf{in}(f(s))$. The latter fact and determinism of I imply that $(a_X, a'_Y) \in f(s)$, which proves $\mathsf{in}(f(s)) \wedge f'(s) \to f(s)$. □

Proof of Theorem 32. Let $f := f(I)$, $f' := f(I')$, $f_\sqcap := f(I \sqcap I')$ and $f_\sqcup := f(I \sqcup I')$.

(1) The containment $f_\sqcap \subseteq f \cup f'$ follows from Lemma 7. The converse is proven by induction on the length of states. Basis: $\varepsilon \in f_\sqcap$. Induction step: Suppose $s \cdot a \in f \cup f'$. WLOG, assume $s \cdot a \in f$. Then $a \in f(s)$. Let $a = (a_X, a_Y)$ with $a_X \in \mathsf{in}(f(s))$. If $a_X \notin \mathsf{in}(f'(s))$, then clearly $a \in f_\sqcap(s)$. Otherwise, there exists a'_Y such that $(a_X, a'_Y) \in f'(s)$. Since I' is deterministic, and by the shared-refinability hypothesis, $a_Y = a'_Y$. Therefore $a \in f(s) \cap f'(s)$, or $s \cdot a \in f \cap f'$, thus, by Lemma 7, $s \cdot a \in f_\sqcap$.

(2) The containment $f \cap f' \subseteq f_\sqcup$ follows from Lemma 9. The converse is proven by induction on the length of states. Basis: $\varepsilon \in f \cap f'$. Induction step: Suppose $s \cdot a \in f_\sqcup$, thus, $a \in f_\sqcup(s)$. By the induction hypothesis, $s \in f_\sqcup$ implies $s \in f \cap f'$. Thus, $a \models \mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge \big(f(s) \vee f'(s)\big)$. Because I and I' are deterministic, this implies $a \models f(s) \wedge f'(s)$, therefore, $s \cdot a \in f \cap f'$.

□