A Theory of Synchronous Relational Interfaces

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

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-45
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-45.html

April 23, 2010

A Theory of Synchronous Relational Interfaces[*]

Stavros Tripakis, UC Berkeley
Ben Lickly, UC Berkeley
Thomas A. Henzinger, IST Austria
Edward A. Lee, UC Berkeley

April 23, 2010

Abstract

In a component-based design context, we propose a relational interface theory for synchronous systems. A component is abstracted by its interface, which consists of input and output variables, as well as one or more contracts. A contract is a relation between input and output assignments. In the stateless case, there is a single contract that holds at every synchronous round. In the general, stateful, case, the contract may depend on the state, modeled as the history of past observations. Interfaces can be composed by connection or feedback. Parallel composition is a special case of connection. Feedback is allowed only for Moore interfaces, where the contract does not depend on the current values of the input variables that are connected (although it may depend on past values of such variables). The theory includes explicit notions of environments, pluggability and substitutability. Environments are themselves interfaces. Pluggability means that the closed-loop system formed by an interface and an environment is well-formed, that is, a state with unsatisfiable contract is unreachable. Substitutability means that an interface can replace another interface in any environment. A refinement relation between interfaces is proposed, that has two main properties: first, it is preserved by composition; second, it is equivalent to substitutability for well-formed interfaces. 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.

1 Introduction

Compositional methods, that allow to assemble smaller components into larger systems both efficiently and correctly, are not simply a desirable feature in system design: they are a must for designing large and complex systems. It is not surprising, then, that a very large body of research has tackled compositionality in the past. Our work is situated in the context of *interface theories* [14, 15], which represent one such body of research. 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.

The type of information about a component that is exposed in an interface is likely to vary depending on the application. For instance, if we are interested simply in type checking, we might abstract a component

[*]This report is a revised version of [47, 48]. 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 #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), 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: Agilent, 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. 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.

(say, a C or Java function) simply by its type signature. If, on the other hand, we are interested in checking correctness properties, say, that a division component never attempts a division by zero, then simple types are not enough, and we would like to have a more detailed interface. Therefore, 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, ease of modeling, as well as tractability of the computational problems involved.

Our work has been motivated by the domains of embedded and cyber-physical systems [24, 30]. In order to build such systems reliably and efficiently, model-based design has been proposed as a paradigm, where formal models are heavily used at the design and analysis levels, and then semantics-preserving implementations are derived from these models as much as possible automatically. The models are often domain specific, since it is important for designers to reason at levels of abstraction appropriate for their domain. 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 [8] are important players in this field [38]. The semantics of the above models rely on the synchronous model of computation, which directly inspired this work.

In our theory, a component is captured by its interface, which contains a set of input variables, a set of output variables, and a set of *contracts*. A contract is simply a relation between assignments of values to inputs and output variables. Syntactically, we use a logical formalism such as first-order logic to represent and manipulate contracts. For example, if x_1 and x_2 are input variables and y is an output variable, then $x_2 \neq 0 \land y = \frac{x_1}{x_2}$ could be the contract of a component that performs division. A more abstract contract for the same component, that only specifies the sign of the output based on the inputs, is the following: $x_2 \neq 0 \land \big(y < 0 \equiv (x_1 < 0 < x_2 \lor x_2 < 0 < x_1)\big)$. An even more abstract contract is $x_2 \neq 0$.[4]

Interfaces govern the operation of a component, which is assumed to evolve 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. Interfaces can be *stateless* or *stateful*. In the stateless case, there is a single contract that holds at every round. In the general, stateful case, there is a different contract for every state. A state is modeled as a history of observations, that is, as a finite sequence of complete assignments. The set of states, as well as the set of contracts, can therefore be infinite, and our theory can handle that. But it is useful to consider also the special case of *finite-state interfaces*, where many different states have the same contract, and the set of contracts is finite. Note that the domains of variables could still be infinite. Finite-state interfaces are represented as finite automata whose locations are labeled by contracts (e.g., formulas).

Interfaces can be composed so that a new interface is obtained as the composition of other interfaces. We provide two composition operators, composition by *connection* and composition by *feedback*, studied in Section 5. Connection essentially corresponds to sequential composition, however, it can also capture parallel composition as a special case (empty connection). Importantly, composition by connection is *not* the same as composition of relations, except in the special case when the interface that provides the outputs is deterministic. This is because, similarly to other works, we use a *demonic* interpretation of non-determinism, corresponding to universal instead of existential quantification. 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).

Composition generates redundant output variables, in the sense that they are equal at every round. We propose a *hiding* operator (Section 6) that allows elimination of such 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

[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/.

[4] 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, as well as for illustration purposes, we often use such properties.

influence the evolution of the contract from one state to the next.

Our theory includes explicit notions of *environments*, *pluggability* and *substitutability* (see Section 7). 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. In general, we distinguish between well-formed and *well-formable* interfaces (the two notions coincide for stateless interfaces). Well-formable interfaces are not necessarily well-formed, but can be made well-formed by appropriately restricting their inputs. As in [15], *controller-synthesis* type of procedures can be used to check whether a given finite-state interface is well-formable and, if it is, transform it into a well-formed one. 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 theory includes a *refinement* relation between interfaces, studied in Section 8. Our refinement is similar in spirit to other refinement relations, such as *alternating refinement* [4], refinement of A/G interfaces [15, 19], *subcontracting* in Eiffel [37], or *function subtyping* in type theory [41, 32], which, roughly speaking, require that I' refines I iff I' accepts more inputs and produces less outputs than I. This requirement is easy to formalize as $\mathsf{in} \to \mathsf{in}' \wedge \mathsf{out}' \to \mathsf{out}$ when input assumptions in are separated from output guarantees out, but needs to be extended in our case, where constraints on inputs and outputs are mixed in the same contract ϕ. We do this by requiring $\mathsf{in}(\phi) \to (\mathsf{in}(\phi') \wedge (\phi' \to \phi))$, where $\mathsf{in}(\phi)$ is the projection of ϕ to the inputs, that is, the assumption part. This definition applies to the stateless case where an interface has a single contract ϕ, but can be easily extended to the stateful case.

Refinement is shown to be a partial order with the following main properties: first, it is preserved by composition; second, it is equivalent to substitutability for well-formed interfaces (more precisely: refinement always implies substitutability; the converse holds when the refined interface is well-formed). Refinement always preserves well-formability. Refinement does not preserve well-formedness in general, but it does so when the refining interface I' has no more legal inputs than the refined interface I.

Our theory supports *shared refinement* of two interfaces I and I' (Section 9). This is important for component reuse, as argued in [19]. Shared refinement, when defined, is shown to be the greatest lower bound with respect to refinement, and is therefore denoted $I \sqcap I'$. $I \sqcap I'$ is an interface that refines both I and I', therefore, it can replace both in any context. In this paper we also propose a corresponding *shared abstraction* operator which is shown to be the least upper bound with respect to refinement, denoted $I \sqcup I'$.

As a special case, we discuss *input-complete* interfaces, that impose no restrictions on inputs, and *deterministic* interfaces, where contracts are partial functions instead of relations. These two subclasses of interfaces are interesting, first, because the theory is greatly simplified in those cases: refinement is implication of contracts, composition is composition of relations, and so on. Second, there is an interesting duality between the two subclasses, as shown in Sections 10 and 11.

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. We provide such algorithms throughout the text, for instance, for composing interfaces, checking refinement, and so on. These algorithms compute some type of product of the automata that represent the interfaces and syntactically manipulate their contracts. Checking satisfiability is required for checking refinement, well-formability, and so on. Decidability of this problem will of course depend on the types of formulas used. Recent advances in *SAT modulo theories* and SMT solvers can be leveraged for this task.

2 Related work

Abstracting components in some mathematical framework that offers stepwise refinement and compositionality guarantees is an old idea. It goes back to the work of Floyd and Hoare on proving program correctness using pre- and post-condititions [20, 25] (a pair of pre- and post-conditions can be seen as a contract for a piece of sequential code) and the work of Dijkstra and Wirth on stepwise refinement as a method for gradually developing programs from their specifications [17, 49]. These ideas were used and developed further in a large number of works, including Abrial's Z notation [44] and B method [2], Liskov's work on CLU [31], Meyer's design-by-contract paradigm [37], as well as Back's work on the refinement calculus [5]. The latter work uses the term contract and its game interpretation, as well as demonic non-determinism, that we also use in our framework.

Using relations as program specifications is also not new, and goes back to the work of Parnas on DL relations [40]. A survey of this body of work is provided in [28]. Of relevance is the relational calculus developed in [21] for sequential programs, where a demonic interpretation of relations is also used.

In a reactive-system setting, Broy considers a relational framework where specifications are sets of stream-processing functions [10, 11]. This framework is more general than ours, in that it can capture stream-processing functions that are not necessarily length-preserving (ours are, because of synchrony of inputs and outputs). On the other hand, Broy uses the more standard definitions of refinement as logical implication (trace inclusion) and composition as composition of relations.

In Dill's trace theory, a component is described using a pair of sets of traces, for legal and illegal behaviors, respectively (*successes* and *failures*) [18]. The theory distinguishes between input and output symbols, but does not impose synchrony of inputs and outputs, since one of its goals is to capture asynchronous circuits. Dill's theory includes an explicit notion of environment, as the "mirror" of a trace structure with input and output symbols reversed. Refinement (called *conformation*) in that theory induces a lattice.

Like trace structures, the framework of interface automata [14] also has an asynchronous, operational flavor. It can capture input-output relations, but in a more explicit or enumerative manner. Our framework is of a more declarative, denotational and symbolic nature. I/O automata [34] are also related, but are by definition input-complete.

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 [39, 45, 27], some on a synchronous model, e.g., see [22, 36], while others are done within a temporal logic framework, e.g., see [6, 1]. 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, e.g., see [26, 45, 43, 23].

[15] defines *relational nets*, which are networks of processes that non-deterministically relate input values to output values. [15] 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 [15] 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. [15] studies *stateless* A/G interfaces, while [19] studies also *stateful* A/G interfaces, in a synchronous setting similar to the one considered in this paper. [19] 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 [19] to conclude that extended interfaces are not appropriate.

[13] 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 [15] and [19] can handle very general compositions of interfaces, that can be obtained via parallel composition and arbitrary connection (similar to the denotational composition framework of [29]). 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 13 (see also Section 12).

3 Preliminaries, notation

We use first-order logic (FOL) notation throughout the paper. For an introduction to FOL, see, for instance, [46]. 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 all 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 $s' \leq s$, $s' \in L$.

4 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 $\mathcal{A}(X \cup Y)^*$.*

We write $\mathsf{InVars}(I)$ for X, $\mathsf{OutVars}(I)$ for Y 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 $f : \mathcal{A}(X \cup Y)^* \rightarrow 2^{\mathcal{A}(X \cup Y)}$, 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 $\phi \to \mathsf{in}(\phi)$ 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. 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 $s \cdot (a_X, a_Y)$.

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.*

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 10 and 11, 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 *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:*

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

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 $\mathsf{in}(I_1) \equiv x > 0$.

Another possible stateless interface for this component is:

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

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. $C : L \to 2^{\mathcal{A}(X \cup Y)}$ 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 $\ell \in L$:*

$$C(\ell) \quad = \quad \bigcup_{(\ell,g,\ell') \in T} g \tag{1}$$

$$\forall (\ell, g_1, \ell_1), (\ell, g_2, \ell_2) \in T : \ell_1 \neq \ell_2 \to g_1 \cap g_2 = \emptyset \tag{2}$$

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. 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.

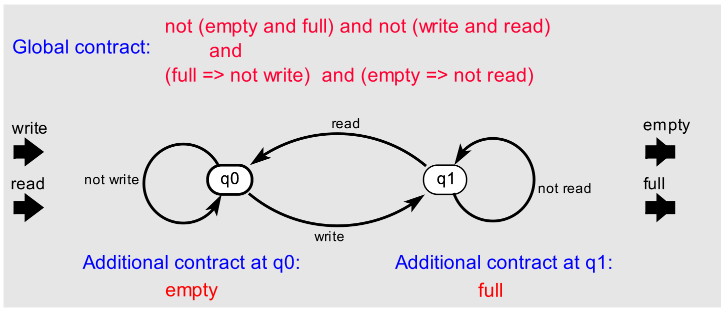
An example of a finite-state interface follows:

Example 2 (Buffer) *Figure 1 shows a finite-state automaton defining a finite-state interface for 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 holds at all location, and a* local contract*, specific to a location. In this example, 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. Both are relational specifications that link inputs and outputs. The local contract at ℓ_0 states that the buffer is empty and at ℓ_1 that it is full.*

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 2. I is assume to have two boolean variables, an input x, and an output y. I is not well-formed, because it*



Global contract:
not (empty and full) and not (write and read)
and
(full => not write) and (empty => not read)
write
read
not write
q0
read
write
q1
not read
empty
full
Additional contract at q0:
empty
Additional contract at q1:
full

Figure 1: Interface for a buffer of size 1.

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.*

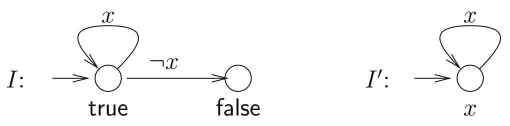


Figure 2: A well-formable interface I and its well-formed witness I'.

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 $I' = (X, Y, f')$ (called a* witness*) 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 $f' \subseteq f$.*

Proof: 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 $s \cdot a \in f$. ■

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.*

Proof: 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. ■

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 [14]. 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

$$C(\ell) \quad := \quad C(\ell) \wedge (\forall Y : C(\ell) \rightarrow \neg g) \tag{3}$$

$\forall Y : C(\ell) \rightarrow \neg g$ 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, (3) 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:

$$\begin{aligned} T \quad := \quad & \{((\ell_1, \ell_2), (C_1(\ell_1) \equiv C_2(\ell_2)) \wedge g_1 \wedge g_2, (\ell_1', \ell_2')) \mid (\ell_i, g_i, \ell_i') \in T_i, \text{ for } i = 1, 2\} \\ & \cup \{((\ell_1, \ell_2), C_1(\ell_1) \not\equiv C_2(\ell_2), \ell_{bad})\} \end{aligned} \tag{4}$$

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

5 Composition

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

5.1 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:

$$\begin{aligned}
\mathsf{OutVars}(\theta) &:= \{y \mid \exists (y,x) \in \theta\} && (5)\\
\mathsf{InVars}(\theta) &:= \{x \mid \exists (y,x) \in \theta\} && (6)\\
X_{\theta(I_1,I_2)} &:= (X_1 \cup X_2) \setminus \mathsf{InVars}(\theta) && (7)\\
Y_{\theta(I_1,I_2)} &:= Y_1 \cup Y_2 \cup \mathsf{InVars}(\theta) && (8)
\end{aligned}$$

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 $s \in \mathcal{A}(X_{\theta(I_1,I_2)} \cup Y_{\theta(I_1,I_2)})^*$:*

$$\begin{aligned}
f(s) &:= f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta \wedge \forall Y_{\theta(I_1,I_2)} : \Phi\\
\Phi &:= (f_1(s_1) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s_2)) && (9)\\
\rho_\theta &:= \bigwedge_{(y,x)\in\theta} y = x
\end{aligned}$$

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 $\theta(I_1, I_2)$.

Definition 9 may seem unnecessarily complex at first sight. In particular, the reader may doubt the necessity of the term $\forall Y_{\theta(I_1,I_2)} : \Phi$, in the definition of $f(s)$. Informally speaking, 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 the validity of our interface theory. Omitting this condition would result in a fundamental property of the theory, namely, preservation of refinement by composition (Theorem 13) not being true, as will be explained in Example 17. Because of this condition, composition by connection does *not* correspond to composition of relations, except in the special case when I_1 is deterministic – see Theorem 28 of Section 11.

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 (9), replacing $f_i(\ell_i)$ by $C_i(\ell_i)$, and T is defined as follows:

$$T := \{((\ell_1, \ell_2), g_1 \wedge g_2, (\ell_1', \ell_2')) \mid (\ell_i, g_i, \ell_i') \in T_i, \text{ for } i = 1, 2\} \qquad (10)$$

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

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 $X_i \cup Y_i$.*

Proof: 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,

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

Now, observe that $\phi \to \mathsf{in}(\phi)$ is a valid formula for any ϕ. Therefore, $f_2(s_2) \to \mathsf{in}(f_2(s_2)) \to \mathsf{in}(f_2(s_2)) \vee \neg\exists Y_1 \cup Y_2 : f_1(s_1)$, which gives

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

■

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

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

Proof: Follows from Lemma 2. ■

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:*

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

Proof: 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 3. 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_{11} 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:

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

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:

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

Let ϕ and ϕ' be the contracts of I and I', respectively. Simplifying, we get:

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

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

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

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

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

Proof of $\phi \to E$: 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. $(x, o) \models E$).

Let q' be an arbitrary assignment over Q such that $(x, q', o) \models \phi_2 \land \rho_{\theta_{23}}$. We want to show

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

Clearly, we have $(x, q', o) \models \phi_2 \land \rho_{\theta_{23}}$ by construction of q'. We also have $(x, o) \models \phi_1 \land \rho_{\theta_{13}} \land \rho_{\theta_{23}} \land C$ since no free variables are in Q are and $(x, q, o) \models A$. Thus 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 $(x, q', o') \models \phi_1 \land \rho_{\theta_{12}} \land \rho_{\theta_{23}}$. We want to now show that $(x, q', o') \models \mathsf{in}(\phi_{23})$. To do so, we first expand $\mathsf{in}(\phi_{23})$:

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

Thus we can reduce the proof to two parts:

(a) $(x, o') \models \exists Q(\phi_2 \land \phi_3 \land \rho_{\theta_{23}})$, and

(b) $(x, o') \models \forall Q(\phi_2 \land \rho_{\theta_{23}} \to \mathsf{in}(\phi_3))$

For part (a), we want to show that for any assignment q_a over Q: $(x, q_a, o') \models \phi_2 \land \rho_{\theta_{23}} \Rightarrow (x, q_a, o') \models \mathsf{in}(\phi_3)$. We start with such an assignment q_a. Combining this with the fact that $(x, o') \models \phi_1 \land \rho_{\theta_{12}} \land \rho_{\theta_{23}}$, we get $(x, q_a, o') \models \phi_1 \land \phi_2 \land \rho_\theta$. Combined with the fact that $x \models C$, we get $(x, q_a, o') \models \phi_1 \land \phi_2 \land \rho_\theta \land C$. 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 $\phi_2 \land \phi_3 \land \rho_{\theta_{23}}$. 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 $(x, o') \models \phi_1 \land \rho_{\theta_{12}} \land \rho_{\theta_{23}}$ 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:

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

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 $(x, q_{Y2}, q_{X3}, q_{Y3}, o') \models \phi_2 \land \phi_3 \land \rho_{\theta_{23}}$, 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 $(x, p', o) \models \phi_1 \land \rho_{\theta_{12}}$. Now take o' over $O \setminus P$ such that $(x, p', o') \models \phi_1 \land \rho_{\theta_{12}} \land \rho_{\theta_{13}}$. 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 $\phi' \to C$.

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 $(x, o') \models \phi_{12} \land \rho_{\theta_{13}} \land \rho_{\theta_{23}}$. Clearly $(x, o') \models \phi_1 \land \rho_{\theta_{12}} \land \rho_{\theta_{13}}$. 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 $\phi' \to D$. ∎

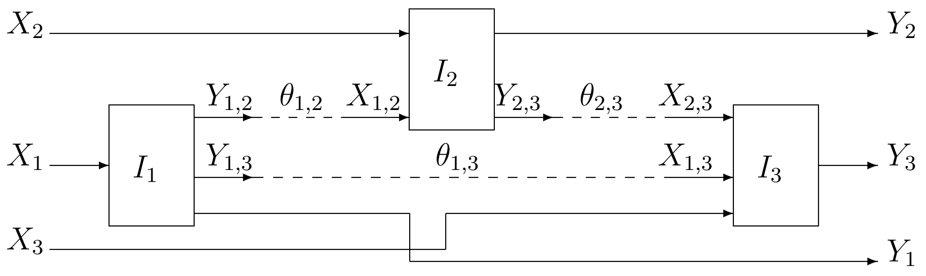


Figure 3: Setting used in the proof of associativity.

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

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

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

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

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 $(\theta_1 \cup \theta_2)((I_{id} \| I_{+1,2}), I_{\le})$. Define $\theta_3 := \theta_1 \cup \theta_2$. From Lemma 2 we get:

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

Then, for $\theta_3((I_{id} \| I_{+1,2}), I_{\le})$, Formula (9) gives:

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

By quantifier elimination, we get

$$\forall y_1, y_2, z_1, z_2 : \Phi \quad \equiv \quad x_1 \le x_2 + 1$$

therefore

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

Notice that $\mathsf{in}(\theta_3((I_{id} \| I_{+1}), I_{\le})) \equiv x_1 \le x_2 + 1$. 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 $x_1 \le x_2 + 2$ 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 5 *Consider the composite interface $\theta_3((I_{id} \| I_{+1,2}), I_{\le})$ from Example 4, 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 $x_1 > x_2 + 1 \wedge x_1 \le x_2 + 1$ is unsatisfiable.*

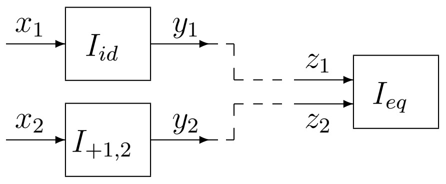


Figure 4: The interface diagram of Example 4.

Contrary to other works [14, 15, 19], we do not impose an a-priori *compatibility* condition on connections. We could easily impose well-formedness or well-formability as a compatibility condition. But we prefer not to do so, because this allows us to state more general results. In particular, Theorem 13 holds independently of whether the connection yields a well-formed interface or not. And together with Theorems 11 and 12, it guarantees that if the refined composite interface is well-formed/formable, then so is the refining one. Having said that, compatibility is a useful concept (see discussion in Section 10), 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.*

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.

5.2 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 [13]. Our definition is similar in spirit, but less restrictive than the one in [13]. 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 $x \in X$.*

Example 6 (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 a Moore interface with respect to some input variable $x \in X$. A* feedback connection *κ on I is a pair (y, x) such that $y \in Y$. Define $\rho_\kappa := (x = y)$. The feedback connection κ defines the interface:*

$$\kappa(I) := (X \setminus \{x\}, Y \cup \{x\}, f_\kappa) \tag{11}$$

$$f_\kappa(s) := f(s) \wedge \rho_\kappa, \quad \textit{for all } s \in \mathcal{A}(X \cup Y)^* \tag{12}$$

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*

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

Proof: 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 $s \in \mathcal{A}(X \cup Y)^*$

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

■

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 feedback connections on I_1, I_2, respectively, such that $\mathsf{InVars}(\kappa_2) \cap \mathsf{InVars}(\theta) = \emptyset$. Then:*

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

Proof: 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: $\kappa_1(\theta(I_1, I_2)) = \theta(\kappa_1(I_1), I_2)$

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. Let us call the contract of the left hand side f_{kt} and of the right hand side f_{tk}. For simplicity in the notation below, we will also name κ_1 as (y, x).

$$\begin{aligned} f_{tk}(s) = && (f_1(s) \wedge x = y) \wedge f_2(s) \wedge \forall Y \cup \{x\} : ((f_1(s) \wedge x = y \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s))) \\ = && (f_1(s) \wedge x = y) \wedge f_2(s) \wedge \forall Y \cup \{x\} : (\neg f_1(s) \vee x \neq y \vee \neg\rho_\theta \vee \mathsf{in}(f_2(s))) \\ = && (f_1(s) \wedge x = y) \wedge f_2(s) \wedge \forall Y : (\neg f_1(s) \vee \neg\rho_\theta \vee \mathsf{in}(f_2(s)) \vee \forall x : x \neq y) \\ = && (f_1(s) \wedge x = y) \wedge f_2(s) \wedge \forall Y : (\neg f_1(s) \vee \neg\rho_\theta \vee \mathsf{in}(f_2(s)) \vee false) \\ = && f_1(s) \wedge f_2(s) \wedge \forall Y : ((f_1(s) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s))) \wedge x = y \\ = && f_{kt}(s) \end{aligned}$$

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

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. As before, we will name the left hand side contract f_{kt}, the right hand side contract f_{tk}, and κ_2 as (y, x).

$$\begin{aligned}
f_{tk}(s) &= & f_1(s) \wedge (f_2(s) \wedge x = y) \wedge \forall Y \cup \{x\} : ((f_1(s) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s) \wedge x = y)) \\
&= & (f_1(s) \wedge x = y) \wedge f_2(s) \wedge \forall Y : ((f_1(s) \wedge \rho_\theta) \rightarrow \exists Y_2 \cup \{x\} : (f_2(s) \wedge x = y)) \\
&= & (f_1(s) \wedge x = y) \wedge f_2(s) \wedge \forall Y : ((f_1(s) \wedge \rho_\theta) \rightarrow \exists Y_2 : (f_2(s) \wedge \exists x : x = y)) \\
&= & (f_1(s) \wedge x = y) \wedge f_2(s) \wedge \forall Y : ((f_1(s) \wedge \rho_\theta) \rightarrow \exists Y_2 : (f_2(s) \wedge \mathit{true})) \\
&= & f_1(s) \wedge f_2(s) \wedge \forall Y : ((f_1(s) \wedge \rho_\theta) \rightarrow \mathsf{in}(f_2(s))) \wedge x = y \\
&= & f_{kt}(s)
\end{aligned}$$

■

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.*

Proof:

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.

■

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.

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

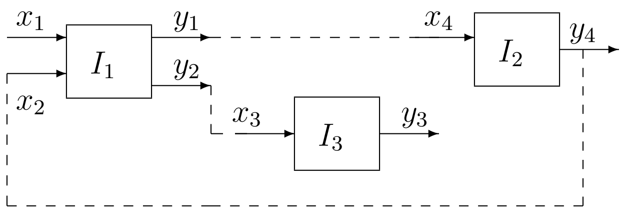


Figure 5: An interface diagram with feedback.

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 7 *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 be a Moore interface with respect to some of its input variables, and let κ be a feedback connection on I. Let $f := f(I)$ and $f_\kappa := f(\kappa(I))$. Then:*

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

Proof: Let $I = (X, Y, f)$ be Moore w.r.t. $x \in X$. Let $\kappa = (y, x)$.

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 $\kappa(I) = (X \setminus \{x\}, Y \cup \{y\}, f_\kappa)$. 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 $(a, p) \models \mathsf{in}(f_\kappa(s))$.

■

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 feedback connection on I. If I is well-formed then $\kappa(I)$ is well-formed.*

Proof: 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. ■

Feedback does not preserve well-formability:

Example 8 *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* $\mathsf{false} \wedge x = y \equiv \mathsf{false}$. *$\kappa(I_f)$ is not well-formable: indeed, $y = 0 \wedge x = y$ implies $x = 0$, therefore, state s_1 cannot be avoided.*

6 Hiding

As can be seen in Example 4, 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 is unclear how the contracts of different histories should be combined. To avoid this problem, we allow hiding only 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 variables in 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, f not referring to z does *not* imply that f is independent from z. Consider, for example, $f(\varepsilon) \equiv \mathsf{true}$, while $f(z = 0) \not\equiv f(z = 1)$, where $f(z = 0)$ denotes a state where $z = 0$, and similarly for $f(z = 1)$.

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.*

Proof: 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 $f(s) = f(s')$. ■

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 definition the follows:

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*

$$\mathsf{hide}(y, I) := (X, Y \setminus \{y\}, f') \tag{13}$$

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 $\ell \in L$.

7 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.

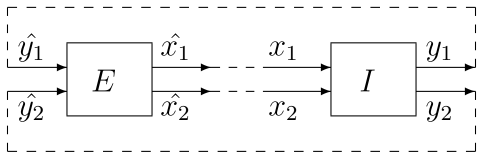


Figure 6: Illustration of pluggability.

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 5.2. 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.

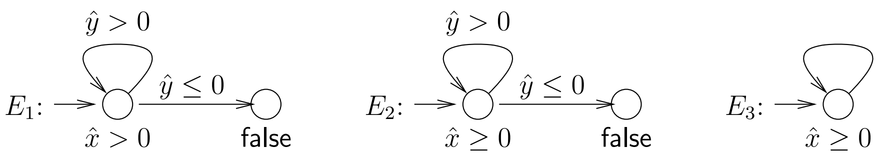


Figure 7: Three environments.

Example 9 *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.*

Proof:

- 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 and well-formed, therefore, by Theorem 19, it 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 a 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.

■

Example 10 *Consider interfaces I and E shown in Figure 8. Observe that I is Moore and $I \leftrightarrows E$. However, I is not well-formable.*

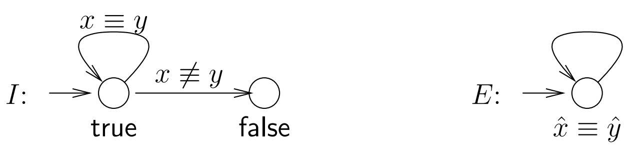


Figure 8: A Moore interface I and a non-Moore environment E.

Example 10 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 write $I \equiv_e I'$ iff both $I \rightarrow_e I'$ and $I' \rightarrow_e I$ hold.*

Theorem 9 *Let I, I' be well-formed interfaces. Then $I \equiv_e I'$ iff $I = I'$.*

Proof: By Theorem 15 of Section 8, $I \equiv_e I'$ implies $I' \sqsubseteq I$ and $I \sqsubseteq I'$. The result follows by antisymmetry of refinement (Theorem 10). ■

8 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:*

$$\mathsf{in}(f(s)) \rightarrow \Big(\mathsf{in}(f'(s)) \wedge \big(f'(s) \rightarrow f(s)\big)\Big) \tag{14}$$

Condition 14 can be rewritten equivalently as the conjunction of the following two conditions:

$$\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f'(s)) \tag{15}$$
$$\big(\mathsf{in}(f(s)) \wedge f'(s)\big) \rightarrow f(s) \tag{16}$$

Condition 15 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 16 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'.

The reader may wonder why Condition (16) could not be replaced with the simpler condition:

$$f'(s') \to f(s) \tag{17}$$

Indeed, as will be shown in Section 10, for input-complete interfaces, Condition (14) reduces to Condition (17) – see Theorem 25. In general, however, the two definitions are different in a profound way, as Example 15, at the end of this section, demonstrates.

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 14. Indeed, $\mathsf{in}(\phi) \to (\mathsf{in}(\phi') \wedge (\phi' \to \phi))$ 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 11 (Buffer that may fail) *This example builds on Example 2. Figure 9 depicts the interface of a single-place buffer that 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.*

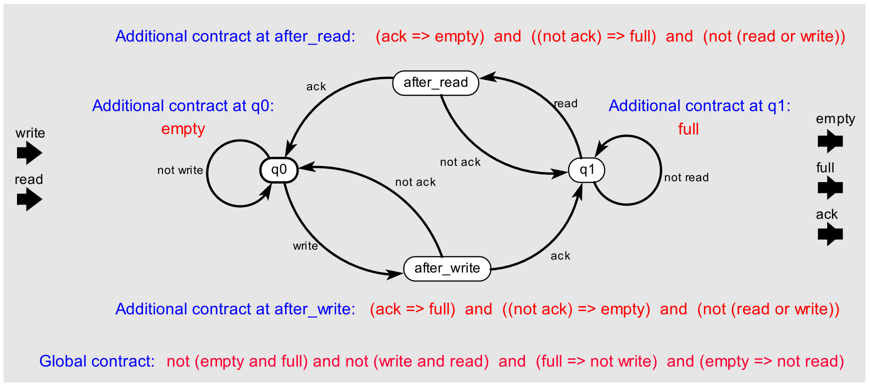

Additional contract at after_read: (ack => empty) and ((not ack) => full) and (not (read or write))
ack
after_read
read
Additional contract at q0:
empty
Additional contract at q1:
full
write
read
not write
q0
not ack
not ack
q1
not read
empty
full
ack
write
after_write
ack
Additional contract at after_write: (ack => full) and ((not ack) => empty) and (not (read or write))
Global contract: not (empty and full) and not (write and read) and (full => not write) and (empty => not read)

Figure 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 1 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.

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 $(\ell_1, \ell_2) \in L_1 \times L_2$, $C(\ell_{good}) := \mathsf{true}$, $C(\ell_{bad}) := \mathsf{false}$, and:

$$\begin{aligned}
T &:= \{((\ell_1, \ell_2), g_{both} \wedge g_1 \wedge g_2, (\ell_1', \ell_2')) \mid (\ell_i, g_i, \ell_i') \in T_i, \text{ for } i = 1, 2\} \\
&\quad \cup \{((\ell_1, \ell_2), g_{bad}, \ell_{bad}), ((\ell_1, \ell_2), g_{good}, \ell_{good}), (\ell_{good}, \mathsf{true}, \ell_{good})\} && (18) \\
g_{both} &:= C_1(\ell_1) \wedge C_2(\ell_2) && (19) \\
g_{good} &:= \mathsf{in}(C_1(\ell_1)) \wedge \mathsf{in}(C_2(\ell_2)) \wedge \neg C_2(\ell_2) && (20) \\
g_{bad} &:= \mathsf{in}(C_1(\ell_1)) \wedge \big(\neg \mathsf{in}(C_2(\ell_2)) \vee C_2(\ell_2) \wedge \neg C_1(\ell_1)\big) && (21)
\end{aligned}$$

Notice that guard g_{bad} encodes the negation of the refinement Condition (14). 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.

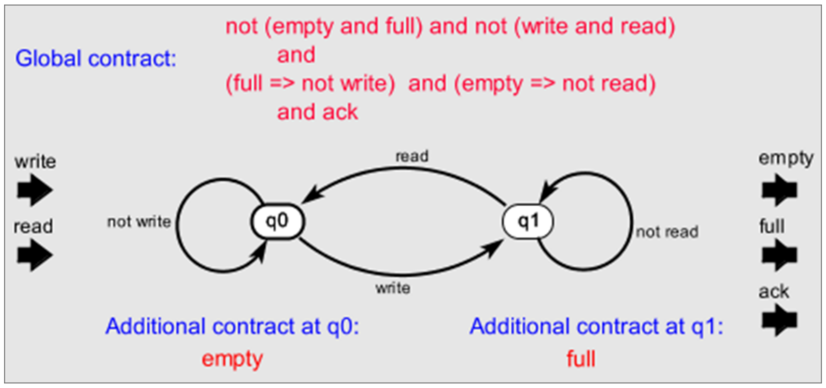

Global contract:
not (empty and full) and not (write and read)
and
(full => not write) and (empty => not read)
and ack
write
read
not write
q0
read
write
q1
not read
empty
full
ack
Additional contract at q0:
empty
Additional contract at q1:
full

Figure 10: Buffer interface of Figure 1 with additional output variable *ack*.

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 Theorem 15 below. The false interface is not pluggable to any environment.

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 $f \cap f'' \subseteq f'$.*

Proof: 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 $s \cdot a \in f'$. ■

Theorem 10 (Partial order) *$\sqsubseteq$ is a partial order, that is, a reflexive, antisymmetric and transitive relation.*

Proof: $\sqsubseteq$ is reflexive because Condition 14 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 $\mathsf{in}(f(s)) \to \mathsf{in}(f'(s))$, $\mathsf{in}(f(s)) \wedge f'(s) \to f(s)$, $\mathsf{in}(f'(s)) \to \mathsf{in}(f''(s))$, and $\mathsf{in}(f'(s)) \wedge f''(s) \to f'(s)$. These imply $\mathsf{in}(f(s)) \to \mathsf{in}(f''(s))$ and $\mathsf{in}(f(s)) \wedge f''(s) \to f(s)$. 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 $f' \subseteq f$. ■

Theorem 11 (Refinement preserves well-formedness for stateless interfaces) *Let I, I' be stateless interfaces such that $I' \sqsubseteq I$. If I is well-formed then I' is well-formed.*

Proof: 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 (15), $a_X \models \mathsf{in}(\phi') \equiv \exists Y' : \phi'$. Therefore, there exists $a'_{Y'}$ such that $(a_X, a'_{Y'}) \models \phi'$. Thus, ϕ' is satisfiable. Thus, I' is well-formed. ■

Theorem 11 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 12 (Refinement and well-formedness/-formability) *Let I, I' be interfaces such that $I' \sqsubseteq I$.*

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

Proof: 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 11, 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 and well-formed, therefore, it is *input-complete* as will be shown in Theorem 19. For input-complete interfaces, $I' \sqsubseteq I$ implies $f' \subseteq f$ (Theorem 25), 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 $\phi_s \to \mathsf{in}(f(s))$. 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 $\phi_s \to \mathsf{in}(f(s))$, $a \models \mathsf{in}(f(s)) \wedge f'(s)$. This and $I' \sqsubseteq I$ imply $a \models f(s)$, thus, $a \models f(s) \wedge \phi_s \equiv f_1(s)$. Thus, $s \cdot a \in f_1$.

   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 $s \in f \cap f'$, $\mathsf{in}(f(s)) \to \mathsf{in}(f'(s))$. Therefore

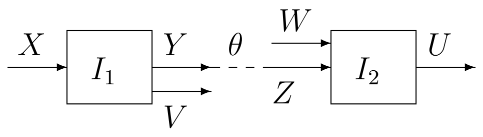


Figure 11: Setting used in Theorem 13.

$\mathsf{in}(f(s)) \wedge \phi_s \to \mathsf{in}(f'(s)) \wedge \phi_s$. The latter formula is equivalent to $\mathsf{in}(f_2(s))$ because $s \in f_1$. Also, $\mathsf{in}(f_1(s)) \wedge f_2(s) \equiv \mathsf{in}(f(s)) \wedge f'(s) \wedge \phi_s \to f(s) \wedge \phi_s \equiv f_1(s)$. 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.

■

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 $f_2 \subseteq f(I_2)$.*

Proof: 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. ■

Theorems 13 and 14 state a major property of our theory, namely, that refinement is preserved by composition.

Theorem 13 (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, $\theta(I_1', I_2') \sqsubseteq \theta(I_1, I_2)$.*

Proof: Let $I_1 = (X, Y \cup V, f_1)$ and $I_2 = (Z \cup W, U, f_2)$, so that $Y \cap V = Z \cap W = \emptyset$, $Y = \mathsf{OutVars}(\theta)$ and $Z = \mathsf{InVars}(\theta)$. 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 11.

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 $W \cup Z \cup U$:

$$\begin{aligned}
\theta(I_1, I_2) &:= (X \cup W, Y \cup V \cup Z \cup U, f) && (22)\\
f(s) &:= f_1(s_1) \wedge f_2(s_2) \wedge \rho_\theta \wedge \Psi && (23)\\
\Psi &:= \forall Y \cup V \cup Z \cup U : (f_1(s_1) \wedge \rho_\theta) \to \mathsf{in}(f_2(s_2)) && (24)\\
\theta(I_1', I_2') &:= (X \cup W, Y \cup V \cup Z \cup U, f') && (25)\\
f'(s) &:= f_1'(s_1) \wedge f_2'(s_2) \wedge \rho_\theta \wedge \Psi' && (26)\\
\Psi' &:= \forall Y \cup V \cup Z \cup U : (f_1'(s_1) \wedge \rho_\theta) \to \mathsf{in}(f_2'(s_2)) && (27)
\end{aligned}$$

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

$$\psi_1 := (\exists Y \cup V \cup Z \cup U : f_1(s_1) \land f_2(s_2) \land \rho_\theta \land \Psi) \land (\forall Y \cup V \cup Z \cup U : \neg f_1'(s_1) \lor \neg f_2'(s_2) \lor \neg \rho_\theta \lor \neg \Psi')$$

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

$$\psi_2 := \Psi \land (\exists Y \cup V \cup Z \cup U : f_1(s_1) \land f_2(s_2) \land \rho_\theta) \land \Big(\neg \Psi' \lor (\forall Y \cup V \cup Z \cup U : \neg f_1'(s_1) \lor \neg f_2'(s_2) \lor \neg \rho_\theta)\Big)$$

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 $a \models \forall Y \cup V \cup Z \cup U : \neg f_1'(s_1) \lor \neg f_2'(s_2) \lor \neg \rho_\theta$. 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 $(a, c, d) \models f_1'(s_1) \land \rho_\theta$. 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 $(a, c, d, e) \models f_1'(s_1) \land f_2'(s_2) \land \rho_\theta$, which contradicts $a \models \forall Y \cup V \cup Z \cup U : \neg f_1'(s_1) \lor \neg f_2'(s_2) \lor \neg\rho_\theta$. Thus, the claim $a \models \neg\Psi'$ is proven and we have that a satisfies:

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

Since a does not satisfy Ψ', there exists an assignment b over $Y \cup V \cup Z \cup U$, such that $(a, b) \models f_1'(s_1) \land \rho_\theta \land \neg\mathsf{in}(f_2'(s_2))$. Since $I_2' \sqsubseteq I_2$, $\mathsf{in}(f_2(s_2)) \to \mathsf{in}(f_2'(s_2))$, or $\neg\mathsf{in}(f_2'(s_2)) \to \neg\mathsf{in}(f_2(s_2))$. 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 $\mathsf{in}(f_1(s_1)) \land f_1'(s_1) \to f_1(s_1)$. 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): $(\mathsf{in}(f(s)) \land f'(s)) \to f(s)$ is valid: Suppose the result does not hold. This means that $\mathsf{in}(f(s)) \land f'(s) \land \neg f(s)$ is satisfiable, i.e.,

$$\psi_4 := (\exists Y \cup V \cup Z \cup U : f_1(s_1) \land f_2(s_2) \land \rho_\theta \land \Psi) \land (f_1'(s_1) \land f_2'(s_2) \land \rho_\theta \land \Psi') \land (\neg f_1(s_1) \lor \neg f_2(s_2) \lor \neg\rho_\theta \lor \neg\Psi)$$

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

$$\psi_5 := \Psi \land \Psi' \land (\exists Y \cup V \cup Z \cup U : f_1(s_1) \land f_2(s_2) \land \rho_\theta) \land (f_1'(s_1) \land f_2'(s_2) \land \rho_\theta) \land (\neg f_1(s_1) \lor \neg f_2(s_2))$$

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

Theorem 14 (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 $\kappa(I') \sqsubseteq \kappa(I)$.*

Proof: 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 $f_\kappa(s) := f(s) \land x = y$ and $f_\kappa'(s) := f'(s) \land x = y$, 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:

$$\mathsf{in}(f_\kappa(s)) \to \mathsf{in}(f_\kappa'(s))$$
$$\big(\mathsf{in}(f_\kappa(s)) \land f_\kappa'(s)\big) \to f_\kappa(s)$$

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:

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

■

Note that the assumption that I' be Moore w.r.t. x in Theorem 14 is essential. Indeed, Mooreness is not generally preserved by refinement:

Example 12 *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 13 *This example is borrowed from [19]. 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 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*.*

Proof:

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 13 and 14, $K(\theta(E, I')) \sqsubseteq K(\theta(E, I))$. Both $K(\theta(E, I))$ and $K(\theta(E, I'))$ are source interfaces, therefore, by part 2 of Theorem 12, $K(\theta(E, I'))$ 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 (14) 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:
   - $\mathsf{in}(f(s)) \not\rightarrow \mathsf{in}(f'(s))$: Observe that, in the contract of the connection of E and I', the term Φ of (9) evaluates to false at state s: this is because $f_e(s) \not\rightarrow \mathsf{in}(f'(s))$. 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.
   - $\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f'(s))$ but $\mathsf{in}(f(s)) \wedge f'(s) \not\rightarrow f(s)$: 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.

■

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

Example 14 *Consider the finite-state interfaces I and I' defined by the automata shown in Figure 2. 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 $x = \mathsf{false}$ is legal for I but not for I'. But there is no environment E such that $I \models E$ but $I' \not\models E$.*

Collecting the results of this section allows us to state the main benefits of our theory, namely, substitutability and incrementality. In particular, let I be an interface formed by some composition of interfaces $I_1, I_2, \ldots$. Suppose we want to replace I_1 by I_1'. We only need to ensure that I_1' refines I_1. This, together with Theorems 13 and 14, guarantees that the new composition, call it I', obtained by using I_1' instead of I_1, refines the old composition I. Moreover, Theorems 11 and 12, guarantee that if I is well-formed or well-formable then so is I'. This means that it suffices to check, say, well-formability, at the level of I and not have to repeat the check at the level of I'. This is very useful when I represents compact specifications while I' is about detailed implementations, that are more difficult to verify. Finally, thanks to Theorem 15, I' can be plugged to any context (i.e., environment) that I can be plugged to, that is, the method is incremental.

We end this section with an additional remark on the definition of refinement. As mentioned above, replacing Condition (16) with the simpler Condition (17) changes the meaning of refinement in a profound way. In particular, part 2 of Theorem 15 no longer holds, as the following example demonstrates:

Example 15 *Consider interface I_1 from Example 1 and interface $I_{id} := (\{x\}, \{y\}, x = y)$. It can be checked that $I_{id} \sqsubseteq I_1$. If we used Condition (17) instead of Condition (16) in the definition of refinement, then I_{id} would not refine I_1: this is because $x = y \not\to x > 0$. Yet there is no environment E such that $I_1 \models E$ but $I_{id} \not\models E$: this follows from Theorem 15.*

9 Shared refinement and shared abstraction

A *shared refinement* operator $\sqcap$ is introduced in [19] 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 [19], is important for component reuse. A similar mechanism called *fusion* has also been proposed in [7].

[19] 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 $s \in f \cap f'$:*

$$\forall X : \big(\mathsf{in}(f(s)) \land \mathsf{in}(f'(s))\big) \to \exists Y : (f(s) \land f'(s)) \tag{28}$$

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

$$\begin{aligned} I \sqcap I' \quad &:= \quad (X, Y, f_\sqcap) \\ f_\sqcap(s) \quad &:= \quad \big(\mathsf{in}(f(s)) \lor \mathsf{in}(f'(s))\big) \land \big(\mathsf{in}(f(s)) \to f(s)\big) \land \big(\mathsf{in}(f'(s)) \to f'(s)\big) \end{aligned} \tag{29}$$

Example 16 *Consider interfaces $I_{00} := (\{x\}, \{y\}, x = 0 \to y = 0)$ and $I_{01} := (\{x\}, \{y\}, x = 0 \to y = 1)$. I_{00} and I_{01} are not shared-refinable because there is no way to satisfy $y = 0 \land y = 1$ 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 (19)):

$$C(\ell) \;:=\; \begin{cases} \big(\mathsf{in}(C_1(\ell_1)) \vee \mathsf{in}(C_2(\ell_2))\big) \wedge \big(\mathsf{in}(C_1(\ell_1)) \to C_1(\ell_1)\big) \wedge \big(\mathsf{in}(C_2(\ell_2)) \to C_2(\ell_2)\big), & \text{if } \ell = (\ell_1, \ell_2) \in L_1 \times L_2 \\ C_1(\ell), & \text{if } \ell \in L_1 \\ C_2(\ell), & \text{if } \ell \in L_2 \end{cases} \tag{30}$$

$$\begin{aligned} T \;:=\; & \{((\ell_1, \ell_2), g_{both} \wedge g_1 \wedge g_2, (\ell_1', \ell_2')) \mid (\ell_i, g_i, \ell_i') \in T_i, \text{ for } i = 1, 2\} \\ & \cup \{((\ell_1, \ell_2), \neg C_2(\ell_2) \wedge g_1, \ell_1') \mid (\ell_1, g_1, \ell_1') \in T_1\} \cup T_1 \\ & \cup \{((\ell_1, \ell_2), \neg C_1(\ell_1) \wedge g_2, \ell_2') \mid (\ell_2, g_2, \ell_2') \in T_2\} \cup T_2 \end{aligned} \tag{31}$$

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*

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

Proof: 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, $s \cdot a \in f$. ■

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:*

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

Proof: 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:

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

Clearly, the last formula implies $\mathsf{in}(f(s)) \vee \mathsf{in}(f'(s))$. The converse also holds, thanks to shared-refinability Condition (28). ■

Theorem 16 (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 $I'' \sqsubseteq (I \sqcap I')$.*

Proof: 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

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

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:

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

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. ■

Theorem 17 (Shared-refinement preserves well-formedness) *If I and I' are shared-refinable interfaces and both are well-formed, then $I \sqcap I'$ is well-formed.*

Proof: 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 (29), 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 (28), 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 (29), thus $(a_X, a'_Y) \in f_\sqcap(s)$. The case $s \in f'$ is symmetric. ■

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:*

$$\begin{aligned} I \sqcup I' &:= (X, Y, f_\sqcup) \\ f_\sqcup(s) &:= \begin{cases} \mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge \big(f(s) \vee f'(s)\big) & \textit{if } s \in f \cap f' \\ f(s) & \textit{if } s \in f \setminus f' \\ f'(s) & \textit{if } s \in f' \setminus f \end{cases} \end{aligned} \tag{32}$$

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*

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

Proof: 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 $s \cdot a \in f_\sqcup$. ■

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 (19)):

$$C(\ell) := \begin{cases} \mathsf{in}(C_1(\ell_1)) \wedge \mathsf{in}(C_2(\ell_2)) \wedge \big(L_1(\ell_1) \vee C_2(\ell_2)\big), & \text{if } \ell = (\ell_1, \ell_2) \in L_1 \times L_2 \\ C_1(\ell), & \text{if } \ell \in L_1 \\ C_2(\ell), & \text{if } \ell \in L_2 \end{cases} \tag{33}$$

$$\begin{aligned} T := {} & \{((\ell_1, \ell_2), g_{both} \wedge g_1 \wedge g_2, (\ell_1', \ell_2')) \mid (\ell_i, g_i, \ell_i') \in T_i, \text{ for } i = 1, 2\} \\ & \cup \{((\ell_1, \ell_2), \mathsf{in}(C_1(\ell_1)) \wedge \mathsf{in}(C_2(\ell_2)) \wedge \neg C_2(\ell_2) \wedge g_1, \ell_1') \mid (\ell_1, g_1, \ell_1') \in T_1\} \cup T_1 \\ & \cup \{((\ell_1, \ell_2), \mathsf{in}(C_1(\ell_1)) \wedge \mathsf{in}(C_2(\ell_2)) \wedge \neg C_1(\ell_1) \wedge g_2, \ell_2') \mid (\ell_2, g_2, \ell_2') \in T_2\} \cup T_2 \end{aligned} \tag{34}$$

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 18 (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 $(I \sqcup I') \sqsubseteq I''$.*

Proof: 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

  $$\begin{aligned} \big(\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s))\big) & \rightarrow \mathsf{in}(f(s)) \\ \big(\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge f(s)\big) & \rightarrow \Big(\mathsf{in}(f(s)) \wedge \mathsf{in}(f'(s)) \wedge \big(f(s) \vee f'(s)\big)\Big) \end{aligned}$$

  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: $\mathsf{in}(f''(s)) \to \mathsf{in}(f(s))$ and $\mathsf{in}(f''(s)) \wedge f(s) \to f''(s)$, 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.

■

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 18 since false is refined by any contract, as observed earlier.

10 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 19 *Every well-formed source interface is input-complete. So is every well-formed Moore interface.*

Proof: Let I be a well-formed interface with contract f. If I is a source interface then it has no input variables. In that case, $\mathsf{in}(f(s))$ is a formula with no free variables, therefore, it is equivalent to either true or false. I is well-formed, so $\mathsf{in}(f(s))$ must be true for all s. If I is Moore then $f(s)$ refers to no input variables, therefore, again $\mathsf{in}(f(s))$ has no free variables. ■

Theorem 20 *Every input-complete interface is well-formed.*

Proof: 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. ■

Every interface I can be turned into an input-complete interface $\mathsf{IC}(I)$ that refines I:

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 21 (Input-completion refines original) *If I is an interface then:*

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

Proof: 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, $\mathsf{in}(f(s)) \to \mathsf{in}(f_{ic}(s))$. We need to show that $(\mathsf{in}(f(s)) \wedge (f(s) \vee \neg\mathsf{in}(f(s)))) \to f(s)$. 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 $f(s)$.

■

Theorems 21 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 9, 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 22 (Connection preserves input-completeness) *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 $s \in \mathcal{A}(X_{\theta(I_1,I_2)} \cup Y_{\theta(I_1,I_2)})^*$*

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

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

Proof: In f, the term Φ defined in Formula (9) 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. ■

It is important to note that the "demonic" interpretation of non-determinism used in our definition of connection is necessary in order for connection to preserve refinement (Theorem 13). In particular, adopting the "angelic" interpretation of non-determinism would result in the standard definition of connection as composition of relations: $f_1(s) \wedge f_2(s) \wedge \rho_\theta$. This works for input-complete interfaces, as shown above, but not for general interfaces, as illustrated in the following example.

Example 17 *Let*

$$\begin{array}{rcl} I_{10} & := & \big(\{x\}, \{y\}, x = 0 \wedge (y = 0 \vee y = 1)\big) \\ I_{12} & := & (\{z\}, \{w\}, z = 0 \wedge w = 0) \end{array}$$

Let $\theta := \{(y, z)\}$. The conjunction of the contracts of I_{10} and I_{12}, together with the equality $y = z$ imposed by the connection θ, gives the contract $x = 0 \wedge (y = 0 \vee y = 1) \wedge z = 0 \wedge w = 0 \wedge y = z$, which is equivalent to $x = y = z = w = 0$, which is clearly satisfiable. Therefore, we could interpret the composite interface $\theta(I_{10}, I_{12})$ as the interface

$$(\{x\}, \{y, z, w\}, x = y = z = w = 0)$$

Now, consider the interface:

$$I_{11} \quad := \quad (\{x\}, \{y\}, x = 0 \wedge y = 1)$$

It can be checked that $I_{11} \sqsubseteq I_{10}$. But if we connect I_{11} to I_{12}, we find that the conjunction of their contracts (with the connection $y = z$) is unsatisfiable. Therefore, if we used conjunction for composition by connection, then the composite interface $\theta(I_{11}, I_{12})$ would not refine $\theta(I_{10}, I_{12})$, even though I_{11} refines I_{10}, i.e., Theorem 13 would not hold.

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.[5] 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.

[5] 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 23 (Feedback preserves input-completeness) *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.*

Proof: 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 $\exists Y \cup \{x\} : f(s) \wedge (x = y) \equiv \exists Y : \exists x : f(s) \wedge (x = y) \equiv \exists Y : (f(s) \wedge (\exists x : x = y)) \equiv \exists Y : f(s) \equiv \mathsf{in}(f(s)) \equiv \mathsf{true}$. ■

Theorem 24 (Hiding preserves input-completeness) *Let $I = (X, Y, f)$ be an input-complete interface and let $Y' \subseteq Y$, such that f is independent from Y'. Then, $\mathsf{hide}(Y', I)$ is input-complete.*

Proof: 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., $\mathsf{in}(f(s))$. ■

Theorem 25 (Refinement for input-complete interfaces) *Let I and I' be input-complete interfaces. Then $I' \sqsubseteq I$ iff $f(I') \subseteq f(I)$.*

Proof: Follows directly from Definitions 16 and 3. ■

For input-complete interfaces, the shared-refinability condition, i.e., Condition (28), simplifies to

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

Clearly, this condition does *not* always hold. Indeed, the interfaces of Example 16 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 26 (Shared refinement and abstraction for input-complete interfaces) *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 $f(I \sqcup I') = f(I) \cup f(I')$.*

Proof: Follows directly from Definitions 17, 18 and 3. ■

As the above presentation shows, input-complete interfaces are much simpler than general interfaces: refinement is implication of contracts, composition is conjunction, and so on. Then, a legitimate question is, why consider non-input-complete interfaces at all? There are mainly two reasons.

First, non-input-complete interfaces can be used to model situations that cannot be modeled by input-complete interfaces. For example, consider modeling a component implementing some procedure that requires certain conditions on its inputs to be satisfied, otherwise it may not terminate. We can capture the specification of this component as an interface, by imposing these conditions in the contract of the interface. But we cannot capture the same specification as an input-complete interface: for what would the output be when the input conditions are violated? We cannot simply add an extra output taking values in $\{T, NT\}$, for "terminates" and "does not terminate", since non-termination is not an observable property.

Second, even in the case where we could use input-complete interfaces to capture a specification, we may decide not to do so, in order to allow for *local compatibility* checks. In particular, when connecting two interfaces I and I', we may want to check that their composition is well-formed *before* proceeding to form an entire interface diagram. Input-complete interfaces are always well-formed and so are their compositions (Theorems 20, 22 and 23), therefore, local compatibility checks provide useful information only in the non-input-complete case.

11 The deterministic case

In this section we state some properties of the theory in the case of deterministic interfaces. First, sink interfaces are by definition deterministic:

Theorem 27 *All sink interfaces are deterministic.*

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

Theorem 28 *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.*

Proof: 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 $(a|b) \models \mathsf{in}(f_2(s_2))$. ■

Theorem 29 (Hiding preserves determinism) *Let $I = (X, Y, f)$ be a deterministic interface and let $Y' \subseteq Y$, such that f is independent from Y'. Then, $\mathsf{hide}(Y', I)$ is deterministic.*

Proof: 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 27. 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 $a_{Y \setminus Y'} \in \mathcal{A}(Y \setminus Y')$ such that $(a_X, a_{Y \setminus Y'}) \in f'(s)$, which proves determinism of $\mathsf{hide}(Y', I)$. ■

Theorem 30 (Refinement for deterministic interfaces) *Let I and I' be deterministic interfaces. Then $I' \sqsubseteq I$ iff $f(I') \supseteq f(I)$.*

Proof: 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 $s \in f$, $f(s) \rightarrow f'(s)$ 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 $\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f'(s))$ and $\mathsf{in}(f(s)) \wedge f'(s) \rightarrow f(s)$ 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 $\mathsf{in}(f(s)) \rightarrow \mathsf{in}(f'(s))$. 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 $\mathsf{in}(f(s)) \wedge f'(s) \rightarrow f(s)$. ■

A corollary of Theorems 25 and 30 is that refinement for input-complete and deterministic interfaces is equality.

For deterministic interfaces, the shared-refinability condition, i.e., Condition (28), 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 31 (Shared refinement and abstraction for deterministic interfaces) *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 $f(I \sqcup I') = f(I) \cap f(I')$.*

Proof: 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, $s \cdot a \in f \cap f'$.

■

Notice that Theorems 30 and 31 are duals of Theorems 25 and 26.

12 Conclusion and perspectives

We have proposed a compositional theory that allows to reason formally about components in a synchronous setting, and offers guarantees of substitutability. The theory is directly applicable to the class of applications captured in synchronous embedded software environments like Simulink, SCADE or Ptolemy, mentioned in the introduction (e.g., see [42] for an example of possible applications). But our framework should be also applicable to more general-purpose software. For example, stateless interfaces can be used as extended types, that are able to express constraints on the outputs based on information about the inputs of a given function. Synchronous hardware is another important application domain for our work. We are currently building an implementation of our theory on Ptolemy and experimenting with different kinds of applications. Reports on such experiments will be provided as part of future work.

Another 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 [35, 9]. 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 [12].

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. This would allow to express, for example, 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 [50], input-output dependency information such as that used in reactive modules [3], or the coarser *profiles* of [33]. A more general solution could involve studying fixpoints in a relational context, as is done, for instance, in [16].

Acknowledgments

We are grateful to Marc Geilen, who observed that the deterministic case is a dual of the input-complete case, to Jan Reineke, whose suggestions improved the definition of environments, 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.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, 1995.

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

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

[4] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *CONCUR'98*, volume 1466 of *LNCS*. Springer, 1998.

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

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

[7] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In *Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007*, pages 200–225. Springer, 2008.

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

[9] G. Berry. The Constructive Semantics of Pure Esterel, 1999.

[10] M. Broy. Compositional refinement of interactive systems. *J. ACM*, 44(6):850–891, 1997.

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

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

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

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

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

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

[17] E.W. Dijkstra. Notes on structured programming. pages 1–82, 1972.

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

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

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

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

[22] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

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

[24] T. Henzinger and J. Sifakis. The discipline of embedded systems design. *IEEE Computer*, 40(10):32–40, 2007.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

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

[27] Bengt Jonsson. Compositional specification and verification of distributed systems. *ACM Trans. Program. Lang. Syst.*, 16(2):259–303, 1994.

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

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

[30] E.A. Lee. Cyber physical systems: Design challenges. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, Jan 2008.

[31] B. Liskov. Modular program construction using abstractions. In *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 354–389. Springer, 1979.

[32] B.H. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

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

[34] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[35] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, 1994.

[36] K.L. McMillan. A compositional rule for hardware design refinement. In *Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*. Springer-Verlag, 1997.

[37] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[38] S.P. Miller, M.W. Whalen, and D.D. Cofer. Software model checking takes off. *Comm. ACM*, 53(2):58–64, 2010.

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

[40] D.L. Parnas. A generalized control structure and its formal definition. *Commun. ACM*, 26(8):572–581, 1983.

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

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

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

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

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

[46] G. Tourlakis. *Mathematical Logic.* Wiley, 2008.

[47] S. Tripakis, B. Lickly, T.A. Henzinger, and E.A. Lee. On relational interfaces. Technical Report UCB/EECS-2009-60, EECS Department, University of California, Berkeley, May 2009.

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

[49] N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 14(4):221–227, 1971.

[50] Y. Zhou and E.A. Lee. Causality interfaces for actor networks. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–35, 2008.