Interface Theories for Component-based Design[⋆,⋆⋆]

Luca de Alfaro[1] Thomas A. Henzinger[2]

[1] University of California, Santa Cruz
[2] University of California, Berkeley

Abstract. We classify component-based models of computation into component models and interface models. A component model specifies for each component how the component behaves in an arbitrary environment; an interface model specifies for each component what the component expects from the environment. Component models support compositional abstraction, and therefore component-based verification. Interface models support compositional refinement, and therefore component-based design. Many aspects of interface models, such as compatibility and refinement checking between interfaces, are properly viewed in a game-theoretic setting, where the input and output values of an interface are chosen by different players.

1 Interfaces vs. Components, Informally

A generic way of depicting system structure is the block diagram. A block diagram consists of entities called blocks related by an interconnect, which specifies a topology for communication between the blocks. A block may represent a physical or logical component, such as a piece of hardware or software, but it more often represents either an abstract description of the *component*, or a description of the component *interface*. A component description answers the question *What does it do?*; an interface description answers the question *How can it be used?*. A component description may be very close to the underlying component, or it may specify as little as a single property of the underlying component. Interface descriptions, too, can be more or less detailed, but they must contain enough information for determining how the underlying components can be composed and connected, and like any good abstraction, they should not contain more information. Component designers often make assumptions about the environment in which a component is to be deployed, and such assumptions, while not part of the component itself and therefore not part of any abstract component description, can be part of an interface description.

[⋆] This research was supported in part by the AFOSR MURI grant F49620-00-1-0327, the DARPA ITO grant F33615-00-C-1693, the MARCO grant 98-DT-660, and the NSF ITR grant CCR-0085949.

[⋆⋆] An abbreviated version of this paper will appear in the *Proceedings of the First International Workshop on Embedded Software* (EMSOFT), 2001.

Components do not constrain the environment; interfaces do. Consider, for example, a division component with inputs x and y and output z. Suppose that our description language is the predicate calculus. The predicate

$$x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \Rightarrow z = x/y \tag{1}$$

is a description of the division component. It does not constrain the environment, but describes the behavior of the component in an arbitrary environment: "for all inputs x and y, if $y \neq 0$, then the output is $z = x/y$." A more abstract description of the same component might be

$$x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \Rightarrow z \in \mathbb{R}.$$

By contrast, the predicate

$$x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \wedge z \in \mathbb{R}$$

is a description of the interface for the division component. It describes an expectation the component has about its environment: "input x is expected to be a real, and input y is expected to be a real different from 0." Such a constraint on the environment is called an *input assumption.* Since the environment consists of other components, a useful interface description not only constrains the environment, but also offers symmetric information about the underlying component, which can then be compared against the input assumptions of interfaces for the environment components. The reciprocal information is called *output guarantee*; in our example, it is "output z is guaranteed to be a real." A more detailed interface description for the division component might be

$$x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \wedge z = x/y. \tag{2}$$

While the component description (1) asserts that "*if* the environment provides proper inputs, *then* the component produces the desired result," the interface description (2) asserts that "the environment provides proper inputs *and* the component produces the desired result."

The informal litmus test *Does it constrain the environment?* applies naturally to many open systems (i.e., systems with free inputs) and open-system descriptions. For instance, a physical circuit is a component, because it behaves (or misbehaves) somehow in all environments; the pin assignment of a hardware chip is an interface, because it puts an expectation on the way in which the chip is deployed. The body of a Pascal procedure is a component; its parameter declaration is an interface. An I/O automaton [13] is a component; an interface automaton [5], which has the same syntax as an I/O automaton, is an interface.

Components and interfaces have different well-formedness criteria. The well-formedness criterion for components is *input-universal*:

$$(\forall x, y)(\exists z)(x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \Rightarrow z = x/y).$$

In other words, a component predicate is well-formed if it is true in *all* environments. This criterion expresses the fact that the component does not constrain the environment and produces an output in an arbitrary environment. The well-formedness criterion for interfaces is *input-existential*:

$$(\exists x, y)(\exists z)(x \in \mathbb{R} \wedge y \in \mathbb{R} \backslash \{0\} \wedge z = x/y).$$

In other words, an interface predicate is well-formed if it is true in *some* environment. An environment that makes the interface predicate true, and thus enables the production of an output, is called a "helpful" environment. The well-formedness criterion expresses the input assumption that the environment is helpful. Input-universal vs. input-existential interpretations of open-system descriptions have also been called *pessimistic* (the environment is allowed to be arbitrary) vs. *optimistic* (the environment is expected to be helpful) approaches to the modeling of open systems [5].

Composition. Composition is a partial function on components, because the result of putting together two well-formed components may not be well-formed. However, as neither component constrains its environment, in many component models there are simple *compatibility* criteria, which ensure that any two compatible components can be composed. For synchronous component models, composition can be more involved because of the possibility of circular I/O dependencies [6, 7] (for example, an inverter component $y = \neg x$ cannot be composed with an identity component $x = y$, despite the fact that both are well-formed). The composition of *interfaces* must resolve both input assumptions and output guarantees: two well-formed interfaces can be composed if (1) they mutually satisfy their respective input assumptions, and (2) the composition (which may still have free inputs) is again a well-formed interface. Interface compatibility can be viewed as a game between the two interfaces and their environment: the environment attempts to be helpful and meet the input assumptions of both interfaces; the interfaces attempt to prevent this. If the environment has a winning strategy in this game, then the the two interfaces can be composed, because then the composition has again a helpful environment.

In the formal treatment below, we split composition into two operations, one for collecting sets of blocks (i.e., components or interfaces), and the other for relating them by an interconnect. This separation, which is inspired by block-diagram languages [12], orthogonalizes concerns, and thus guides and simplifies the presentation.

Hierarchy. Hierarchical block diagrams support abstraction and refinement. Abstraction allows a block diagram to be compressed into a single block; refinement allows a block to be expanded into a block diagram. The hierarchical relationship between blocks is *contravariant* on inputs and outputs: a more refined description of an open system may make weaker input assumptions and stronger output guarantees than a more abstract description of the same system. For components, which make no input assumptions, abstraction is therefore weakening, such as implication or trace containment or simulation, and refinement is strengthening. For interfaces, which do make input assumptions, hierarchy maintains its contravariant character, and can be defined as alternating trace containment or alternating simulation [3].

Components typically support *compositional abstraction*: for two compatible components f' and g', if f' can be abstracted to f, and g' can be abstracted to g, then f and g are again compatible. This is because for components, abstraction is weakening (say, implication) and compatibility (say, nonemptiness of conjunction) is made more likely by weakening. Compositional abstraction is the basis for component-based *verification* methods, which proceed bottom-up from a system description and require the following: in order to prove that the given system $f'\|g'$ refines the specification $f\|g$, it suffices to *independently* prove that the component f' refines the partial specification f, and that g' refines g. Assume-guarantee rules for compositional verification are more elaborate, but have the same "direction" [1, 9]: given the composite system, if we establish properties of the components, then we can conclude properties of the composite system.

Interfaces, by contrast, can be made to support *compositional refinement*, which is the "opposite direction" of abstraction: for two compatible interfaces F and G, if F can be refined to F', and G can be refined to G', then F' and G' are again compatible. This is possible because for interfaces, refinement weakens input assumptions and thus can make compatibility more likely. Compositional refinement is the basis for component-based *design*, which proceeds top-down from an interface description and requires the following: in order to refine the given interface $F\|G$ towards an implementation, it suffices to *independently* refine F and G, say, to F' and G', respectively; then the refinements F' and G' are compatible and their composition refines the interface $F\|G$. We present several interface formalisms that permit component-based design in this sense.

Formalism. Formalism enables tool support. Formal component models can support compositional verification with algorithmic tools for component-abstraction checking (e.g., "model checking" [4]). Formal interface models can support compositional design with algorithmic tools for interface-compatibility checking and interface-refinement checking. Interface models are typically less ambitious and smaller than component models, which often attempt to capture behavioral aspects of the underlying components for verification. Formal interface models in general, and automatic compatibility checking in particular, therefore offer an opportunity for formal methods to succeed and have practical impact on the design process.

While formal component models (e.g., [1, 10, 13]) and informal interface models (e.g., [14]) abound, formal interface models are less common. Notable examples are *trace theory* [8] and *lazy composition* [15], both of which combine, according to our terminology, component and interface aspects. Most formalisms, however, limit the input assumptions to assumptions about the types of input values. In practice, on the other hand, designers make much richer input assumptions. For example, an object-oriented software designer of a class with an initialization method and other methods may require that the initialization method is called before any of the other methods is called. Such temporal-ordering assumptions can be captured by *interface automata*, a formal interface model for asynchronous component interaction [5]. An embedded software designer of a control task may assume that a sensor input is updated with a certain frequency (an assumption on the plant), or that the task has a certain worst-case execution time (an assumption on the hardware), or that the task finishes execution within a certain time bound (an assumption on hardware and scheduler). Such real-time assumptions require a rich interface model with timing assumptions and timing guarantees, which has to wait for later work. In this paper, we lay the foundation for interface formalisms by defining the framework and presenting a few formal interface models, called *interface theories*, for simple kinds of synchronous component interaction.

2 Interfaces vs. Components, Formally

We capture hierarchical block diagrams formally by a mathematical object called block algebra. A *block algebra* consists of:

- A set of *blocks*.
- For each block F, a set P_F of *ports*. A *port* is a typed variable. We assume that all types are nonempty and write $\mathbb{T}_x$ for the type of port x.
- A partial binary function, called *composition*, mapping two blocks F and G to a block $F\|G$. We require that if the composition $F\|G$ is defined, then $P_{F\|G} = P_F \cup P_G$. We also require that composition is commutative and associative: (1) if $F\|G$ is defined, then $G\|F$ is defined and equal to $F\|G$; (2) if $(F\|G)\|H$ is defined, then $F\|(G\|H)$ is defined and equal to $(F\|G)\|H$. In other words, composition is a partial function that maps a set of blocks to a single block.
- A partial binary function, called *connection*, mapping a block F and an interconnect θ to a block $F\theta$. An *interconnect* is a set of channels, and a *channel* (x, y) is a pair consisting of a port x called *source*, and a port y called *target*, such that $\mathbb{T}_x = \mathbb{T}_y$. Given an interconnect θ, we write $I_\theta = \{x \mid (\exists y)(x, y) \in \theta\}$ for the set of sources, $O_\theta = \{y \mid (\exists x)(x, y) \in \theta\}$ for the set of targets, and ρ_θ for the predicate $\bigwedge_{(x,y)\in\theta}(x = y)$. We require that if the connection $F\theta$ is defined, then $P_{F\theta} = P_F \cup I_\theta \cup O_\theta$. We also require that if $\theta = \emptyset$, then $F\theta$ is defined and equal to F.
- A binary relation $\preceq$, called *hierarchy*, between blocks. If $F' \preceq F$, then the block F' is said to *refine* the block F, and F is said to *abstract* F'. We require that $\preceq$ is reflexive and transitive.

We distinguish between block algebras whose blocks represent interfaces, and block algebras whose blocks represent components: interface algebras support top-down design; component algebras support bottom-up verification. Top-down design iteratively refines a block F into a block F' such that $F' \preceq F$; bottom-up verification iteratively abstracts a block f' into a block f such that $f' \preceq f$. A block algebra is an *interface algebra*, and the blocks are called *interfaces*, if both of the following:

1. For all interfaces F, G, and F', if $F \| G$ is defined and $F' \preceq F$, then $F' \| G$ is defined and $F' \| G \preceq F \| G$.
2. For all interfaces F and F', and all interconnects θ, if $F\theta$ is defined and $F' \preceq F$, then $F'\theta$ is defined and $F'\theta \preceq F\theta$.

A block algebra is a *component algebra*, and the blocks are called *components*, if both of the following:

1. For all components f', g', and f, if $f' \| g'$ is defined and $f' \preceq f$, then $f \| g'$ is defined and $f' \| g' \preceq f \| g'$.
2. For all components f' and f, and all interconnects θ, if $f'\theta$ is defined and $f' \preceq f$, then $f\theta$ is defined and $f'\theta \preceq f\theta$.

Interfaces and components are related by implementations. Given an interface algebra $\mathcal{A}$ and a component algebra $\mathcal{B}$, an *implementation of $\mathcal{A}$ by $\mathcal{B}$* is a binary relation $\lhd$ between the components of $\mathcal{B}$ and the interfaces of $\mathcal{A}$ such that for every interface F, there exists a component f with $f \lhd F$; that is, every interface can be implemented. If $f \lhd F$, then the component f is said to *implement* the interface F, and F is called an *interface for f*. Component-based design is supported by compositional implementations. The implementation $\lhd$ is *compositional* if all of the following:

1. For all components f and g of $\mathcal{B}$, and all interfaces F and G of $\mathcal{A}$, if $f \lhd F$ and $g \lhd G$ and $F \| G$ is defined, then $f \| g$ is defined and $f \| g \lhd F \| G$.
2. For all components f of $\mathcal{B}$, all interfaces F of $\mathcal{A}$, and all interconnects θ, if $f \lhd F$ and $F\theta$ is defined, then $f\theta$ is defined and $f\theta \lhd F\theta$.
3. For all components f of $\mathcal{B}$, and all interfaces F' and F of $\mathcal{A}$, if $f \lhd F'$ and $F' \preceq F$, then $f \lhd F$.

If $\mathcal{A}$ is an interface algebra, $\mathcal{B}$ a component algebra, and $\lhd$ a compositional implementation of $\mathcal{A}$ by $\mathcal{B}$, then the pair $(\mathcal{A}, \lhd)$ is called an *interface theory* for $\mathcal{B}$. There may be several interface theories for a component algebra. Given two interface theories $(\mathcal{A}_1, \lhd_1)$ and $(\mathcal{A}_2, \lhd_2)$ for the component algebra $\mathcal{B}$, the theory $(\mathcal{A}_2, \lhd_2)$ is *as expressive for $\mathcal{B}$ as* the theory $(\mathcal{A}_1, \lhd_1)$ if there is a function α from the interfaces of $\mathcal{A}_1$ to the interfaces of $\mathcal{A}_2$ such that all of the following:

1. For all interfaces F and G of $\mathcal{A}_1$, if $F \| G$ is defined, then $\alpha(F) \| \alpha(G)$ is defined and equal to $\alpha(F \| G)$.
2. For all interfaces F of $\mathcal{A}_1$, and all interconnects θ, if $F\theta$ is defined, then $\alpha(F)\theta$ is defined and equal to $\alpha(F\theta)$.

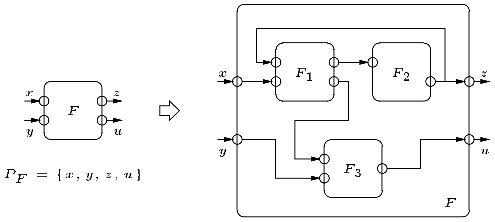

Fig. 1. Component-based design refinement using an interface theory

3. For all interfaces F' and F of $\mathcal{A}_1$, if $F' \preceq F$, then $\alpha(F') \preceq \alpha(F)$.
4. For all interfaces F of $\mathcal{A}_1$, and all components f of $\mathcal{B}$, if $f \lhd_1 F$, then $f \lhd_2 \alpha(F)$.

Interface theories support compositional design. Suppose that we want to design a component that implements a given interface F. An interface theory allows us to split the design task into a number of subtasks handled by independent designers. We can refine the interface F into an interface of the form $(F_1 \| \cdots \| F_k)\theta$; that is, the block F is refined into k blocks that are connected by a set θ of channels (cf. Figure 1). The new interfaces F_1 to F_k can be handed off to k different designers. Suppose that the first designer builds a component f_1 that implements the interface F_1, the second designer buys off the shelf a component f_2 that implements F_2, etc. Then the interface theory guarantees that (1) the k components can be composed and connected to form the system $(f_1 \| \cdots \| f_k)\theta$, and (2) this system implements the given interface F. Mathematically, if $(F_1 \| \cdots \| F_k)\theta \preceq F$, and $f_j \lhd F_j$ for all $1 \leq j \leq k$, then $(f_1 \| \cdots \| f_k)\theta \lhd F$. More generally, such a compositional design process can proceed through multiple levels of refinement. Given two interface theories $(\mathcal{A}_1, \lhd_1)$ and $(\mathcal{A}_2, \lhd_2)$ for a component algebra $\mathcal{B}$, if $(\mathcal{A}_2, \lhd_2)$ is as expressive for $\mathcal{B}$ as $(\mathcal{A}_1, \lhd_1)$, than every compositional design process that is carried out in the interface theory $(\mathcal{A}_1, \lhd_1)$ can also be carried out in the interface theory $(\mathcal{A}_2, \lhd_2)$.

An interface theory also supports the component-wise evolution of a design. Suppose that the system $(f_1 \| \cdots \| f_k)\theta$ implements the interface F. If we want to replace a component f_j by another component f'_j, and we are given the corresponding interface F_j, then we need to ensure only $f'_j \lhd F_j$ in order to put together a revised system $(f_1 \| \cdots \| f'_j \| \cdots \| f_k)\theta$ that implements F.

Component algebras support compositional verification. Suppose that we want to verify that a given system $(f'_1 \| \cdots \| f'_k)\theta$ satisfies a specification f, which may be a more abstract description of the system, or a property of the system. A component algebra allows us to split the verification task into a number of subtasks of smaller complexity. We can establish independently the k proof obligations that each component f'_j satisfies a corresponding specification f_j. Then the component algebra guarantees that the k partial specifications can be composed and connected to form a single specification $(f_1 \| \cdots \| f_k)\theta$, which, it remains to be shown, implies the given specification f. Mathematically, if $f'_j \preceq f_j$ for all $1 \leq j \leq k$, and $(f_1 \| \cdots \| f_k)\theta \preceq f$, then $(f'_1 \| \cdots \| f'_k)\theta \preceq f$. More generally, such a compositional verification process can proceed through multiple levels of abstraction.

3 Some Stateless Interface Algebras

We begin by considering some examples of interfaces without state, and defer interfaces with state to Section 5. Stateless interfaces may of course be implemented by stateful components. For example, while the parameter declaration of a Pascal procedure is a stateless interface similar to the stateless I/O interfaces defined below, the body of a Pascal procedure is a component that typically has state (e.g., local variables). We discuss three classes of stateless interfaces:

1. An *input/output* (I/O) *interface* constrains the environment of a component by specifying the names and types of input ports. Symmetrically, it provides the same information about output ports.
2. An *assume/guarantee* (A/G) *interface* is an I/O interface that constrains, in addition, the ranges of values expected at input ports. Symmetrically, it provides range information about output ports. The example

$$x \in \mathbb{R} \wedge y \in \mathbb{R}\backslash\{0\} \wedge z \in \mathbb{R}$$

   from the introduction is a stateless A/G interface.
3. A *port-dependency* (PD) *interface* is an I/O interface that constrains which output ports may influence the values at which input ports. Symmetrically, it provides dependency information between input and output ports.

3.1 The stateless input/output interfaces

A *stateless I/O interface* F consists of a set I_F of *input ports*, a set O_F^+ of *available ports* disjoint from I_F, and a set $O_F \subseteq O_F^+$ of *output ports*. The available ports are reserved names for choosing output ports when refining the interface; they are used to ensure that whenever two interfaces are implemented independently by two components, then the components have different output ports. Let $P_F = I_F \cup O_F$ and $P_F^+ = I_F \cup O_F^+$.

`Composition` $F \| G$ is defined iff $P_F^+ \cap P_G^+ = \emptyset$. Then, $I_{F\|G} = I_F \cup I_G$ and $O_{F\|G}^+ = O_F^+ \cup O_G^+$ and $O_{F\|G} = O_F \cup O_G$.

`Connection` $F\theta$ is defined iff (1) $I_\theta \subseteq O_F$, (2) $O_\theta \cap O_F^+ = \emptyset$, and (3) for all channels $(x,y),(x',y') \in \theta$, if $x \neq x'$, then $y \neq y'$; that is, two channels cannot have the same target. Then, $I_{F\theta} = I_F \backslash O_\theta$ and $O_{F\theta}^+ = O_F^+ \cup O_\theta$ and $O_{F\theta} = O_F \cup O_\theta$.

`Hierarchy` $F' \preceq F$ iff $I_{F'} \subseteq I_F$ and $O^+_{F'} \subseteq O^+_F$ and $O_{F'} \supseteq O_F$. Note the contravariance between inputs and outputs: a component that implements a refinement of the interface F may not use all input ports in I_F, but it must provide values at all output ports in O_F, because those may be expected by the environment.

Proposition 1. *The stateless I/O interfaces are an interface algebra.*

3.2 The stateless assume/guarantee interfaces

A *stateless A/G interface* F consists of a stateless I/O interface $\Pi_F = (I_F, O^+_F, O_F)$, a satisfiable predicate ϕ_F on I_F called *input assumption*, and a satisfiable predicate ψ_F on O_F called *output guarantee*. The input assumption specifies the value combinations at the input ports which a component that implements the interface must accept, and the output guarantee specifies the value combinations at the output ports which such a component may produce.

`Composition` $F \| G$ is defined iff $\Pi_F \| \Pi_G$ is defined. Then, $\phi_{F\|G} = \phi_F \wedge \phi_G$ and $\psi_{F\|G} = \psi_F \wedge \psi_G$.

`Connection` $F\theta$ is defined iff (1) $\Pi_F\theta$ is defined and (2) the input assumption $\phi_{F\theta}$, as defined next, is satisfiable. Let $\phi_{F\theta} = (\forall O_{F\theta})(\psi_F \wedge \rho_\theta \Rightarrow \phi_F)$ and $\psi_{F\theta} = \psi_F \wedge \rho_\theta$. The input assumption $\phi_{F\theta}$ states that a component that implements the interface $F\theta$ expects inputs so that the input assumption of F is satisfied, provided the outputs of F, some of which may be connected to inputs by θ, do not violate the output guarantee of F.

`Hierarchy` $F' \preceq F$ iff (1) $\Pi_{F'} \preceq \Pi_F$, (2) the input assumption $\phi_{F'}$ is implied by ϕ_F, and (3) the output guarantee $\psi_{F'}$ implies ψ_F. Note the contravariance between input assumptions and output guarantees: a component that implements a refinement of the interface F must be prepared to accept all inputs that satisfy the input assumption ϕ_F, and it may produce only outputs that satisfy the output guarantee ψ_F.

Proposition 2. *The stateless A/G interfaces are an interface algebra.*

Example 1. Consider the stateless A/G interface F with two input ports x and y, and an output port z, all integer-valued. The input assumption is $\phi_F = (x = 0 \Rightarrow y = 0)$; that is, the environment is expected to provide input values so that either the x value is different from 0, or both input values are 0. The output guarantee is $\psi_F = \text{TRUE}$; that is, a component that implements F may produce any integer output. The connection $F\theta$ with $\theta = \{(z, x)\}$ is legal (i.e., defined): as the environment of $F\theta$ does not know whether or not the value at x is 0, to be on the safe side, it must provide the value 0 at the remaining input port y; that is, $\phi_{F\theta} = (y = 0)$. Since now both z and x are output ports, we have $\psi_{F\theta} = (z = x)$. The connection $F\theta'$ with $\theta' = \{(z, y)\}$ is also legal: to be on the safe side, the environment must provide a value different from 0 at the remaining input port x; that is, $\phi_{F\theta'} = (x \neq 0)$ and $\psi_{F\theta'} = (z = y)$.

The stateless A/G interface F' is just like F, except that it has the stronger output guarantee $\psi_{F'} = (z \neq 0)$; that is, a component that implements F' is guaranteed to produce an output value different from 0. The interface F' has fewer implementations than F; indeed, F' refines F. Consequently, the connection $F'\theta$ with $\theta = \{(z, x)\}$ is still legal: in fact, as x is guaranteed to be different from 0, the environment is free to provide any input value at y; that is, $\phi_{F'\theta} = \text{TRUE}$ and $\psi_{F'\theta} = (z \neq 0 \wedge z = x)$. Also the connection $F'\theta'$ with $\theta' = \{(z, y)\}$ is still legal: $\phi_{F'\theta'} = (x \neq 0)$ and $\psi_{F'\theta'} = (z \neq 0 \wedge z = y)$. Note that both $F'\theta \preceq F\theta$ and $F'\theta' \preceq F\theta'$, as predicted. □

Example 2. Consider the stateless A/G interface G with two input ports x and y, and two output ports z and u, all boolean-valued. The input assumption is $\phi_G = (x = y)$ and the output guarantee is $\psi_G = (z = u)$; that is, the environment is expected to provide equal input values at x and y, and the component guarantees the output values at z and u to be equal. The connection $G\theta$ with $\theta = \{(z, x), (u, y)\}$ is legal: $\phi_{G\theta} = \text{TRUE}$ (there are no more inputs) and $\psi_{G\theta} = (z = u \wedge z = x \wedge u = y)$. However, the connection $G\theta_1$ with $\theta_1 = \{(z, x)\}$ is illegal (i.e., undefined), because there is no condition on the remaining input y that would guarantee that $x = y$. The connection $G\theta_2$ with $\theta_2 = \{(u, y)\}$ is similarly illegal. This shows the need for considering *sets* of channels as interconnects, rather than considering individual channels one at a time. □

3.3 The stateless port-dependency interfaces

A *stateless PD interface F* consists of a stateless I/O interface $\Pi_F = (I_F, O_F^+, O_F)$ and an *I/O-dependency relation* $\kappa_F \subseteq I_F \times O_F$. Intuitively, $(x, y) \in \kappa_F$ means that the value at input port x can influence the value at output port y.

`Composition` $F \| G$ is defined iff $\Pi_F \| \Pi_G$ is defined. Then, $\kappa_{F\|G} = \kappa_F \cup \kappa_G$.

`Connection` $F\theta$ is defined iff (1) $\Pi_F\theta$ is defined and (2) for all channels $(x, y) \in \kappa_F$, we have $(y, x) \notin \theta$. In other words, the port dependencies introduced by the interconnect θ must not close a dependency cycle. Let $\kappa^*_{F\theta} \subseteq P_{F\theta} \times O_{F\theta}$ be the smallest transitive relation such that $\kappa_F \subseteq \kappa^*_{F\theta}$ and $\theta \subseteq \kappa^*_{F\theta}$. Then, $\kappa_{F\theta} = \kappa^*_{F\theta} \cap (I_{F\theta} \times O_{F\theta})$.

`Hierarchy` $F' \preceq F$ iff $\Pi_{F'} \preceq \Pi_F$ and $\kappa_{F'} \cap (I_F \times O_F) \subseteq \kappa_F$. In other words, a component that implements a refinement of the interface F must not have more I/O dependencies than permitted by the I/O dependency relation κ_F.[1]

Proposition 3. *The stateless PD interfaces are an interface algebra.*

[1] Exactly the opposite condition is required in *component* algebras with dependency relations: a more abstract component may have fewer I/O dependencies (cf. [1]).

4 Some Component Algebras

For each interface algebra $\mathcal{A}$ from the previous section, we give an example of a component algebra $\mathcal{B}$ such that there is a compositional implementation $\lhd$ of $\mathcal{A}$ by $\mathcal{B}$. All examples of component algebras presented here are *netlists*, i.e., sets of atomic blocks called processes which are connected by channels. In the most general case, each process specifies a nonempty relation between input and output ports. Such a relational net is well-formed if there exist port values that satisfy the I/O relations of all processes, and all identities enforced by channels. The stateless I/O interfaces, the stateless A/G interfaces, and the stateless PD interfaces each capture sufficient conditions on the well-formedness of a subclass of relational nets, and thus provide an interface theory for that subclass.

Notation. A *valuation* p on a set P of ports is a function that maps each port $x \in P$ to a value $p(x) \in \mathbb{T}_x$. We write $[P]$ for the set of valuations on P. Given a valuation $p \in [P]$ and a predicate ρ on P, by $\rho@p$ we denote the truth value of ρ evaluated at p. Consider two sets P_1 and P_2 of ports, and two valuations $p_1 \in [P_1]$ and $p_2 \in [P_2]$. We write $p_1 \cong p_2$ if $p_1(x) = p_2(x)$ for all ports in $P_1 \cap P_2$. If $p_1 \cong p_2$, then $p_1 \uplus p_2$ denotes the valuation in $[P_1 \cup P_2]$ such that $(p_1 \uplus p_2)(x) = p_1(x)$ for all $x \in P_1$, and $(p_1 \uplus p_2)(x) = p_2(x)$ for all $x \in P_2$.

4.1 The relational nets

A *process* a consists of a set I_a of *input ports*, a set O_a of *output ports* disjoint from I_a, and a satisfiable predicate ρ_a on $I_a \cup O_a$ called *I/O relation*. For a process a, let $P_a = I_a \cup O_a$. A *relational net* f consists of a set A_f of processes and a set C_f of channels, such that all of the following:

1. For all processes $a, b \in A_f$, if $a \neq b$, then $P_a \cap P_b = \emptyset$.
2. For all processes $a \in A_f$ and all channels $(x, y) \in C_f$, we have $y \notin O_a$.
3. For all channels $(x, y), (x', y') \in C_f$, if $x \neq x'$, then $y \neq y'$.
4. Let $P_f = \bigcup_{a \in A_f} P_a \cup \bigcup_{(x,y) \in C_f} \{x, y\}$. There is an I/O valuation $p \in [P_f]$ such that (a) $\rho_a@p$ is true for all processes $a \in A_f$, and (b) $p(x) = p(y)$ for all channels $(x, y) \in C_f$.

If (a) and (b), then the I/O valuation p is called *consistent* with the relational net f. A port $x \in P_f$ of a relational net f is a *primary input port* if (1) there is no process $a \in A_f$ with $x \in O_a$, and (2) there is no port $y \in P_f$ with $(y, x) \in C_f$. We write I_f for the primary input ports of the relational net f, and $O_f = P_f \backslash I_f$ for the other ports.

`Composition` $f \| g$ is defined iff $P_f \cap P_g = \emptyset$. Then, $A_{f\|g} = A_f \cup A_g$ and $C_{f\|g} = C_f \cup C_g$.

`Connection` $f\theta$ is defined iff the result $(A_{f\theta}, C_{f\theta})$, as defined next, is a relational net. Let $A_{f\theta} = A_f$ and $C_{f\theta} = C_f \cup \theta$.

`Hierarchy` $f' \preceq f$ iff (1) $P_{f'} \supseteq P_f$, (2) $O_{f'} \supseteq O_f$, and (3) for every I/O valuation $p \in [P_{f'}]$, if p is consistent with f', then p is consistent with f. Note the covariance between inputs and outputs: a more abstract relational net f may have fewer input and output ports than f', and it may leave some output ports of f' unconstrained by treating them as free inputs.

Proposition 4. *The relational nets are a component algebra.*

The relational nets are very general, in that they admit processes with (1) partial I/O relations, which do not accept all input valuations, and (2) arbitrary dependencies between input and output valuations. An example of relational processes are the input-constraining Mealy machines whose combinational I/O dependencies do not change in time. Instead of a formal account of this statement, we only give an intuitive explanation. First, a state machine with inputs and outputs, such as a Moore or Mealy machine, is *input-enabling* if in every state, the machine accepts all possible inputs. We refer to a class of state machines as *input-constraining* if its members are not necessarily input-enabling. Second, stateless interfaces consider only the *combinational* I/O dependencies of a state machine, which assert how an output value depends on the *concurrent* input values. In particular, a Moore machine has *no* combinational I/O dependencies, because an output value may depend only on *previous* input values. Thus, to view a state machine M as a relational process a, we construct the I/O relation ρ_a from the combinational I/O dependencies of M, and abstract away all other detail, such as sequential I/O dependencies.

We have no interface theory for the relational nets. Instead, we consider three restricted classes of relational nets for which simple interface theories exist:

1. In the first restricted class, the *rectangular nets*, the processes can still restrict the accepted input valuations, but there are no I/O dependencies. An example of such processes are the input-constraining Moore machines whose input assumptions do not change in time. An appropriate interface theory are the stateless A/G interfaces. (To capture input assumptions that that may change in time, *stateful* A/G interfaces are needed; see Section 5.)
2. In the second restricted class, the *total nets*, the processes do not restrict the accepted input valuations, but there can be I/O dependencies. An example of such processes are the input-enabling Mealy machines whose combinational I/O dependencies do not change in time. An appropriate interface theory are the stateless PD interfaces, which rule out cycles of combinational I/O dependencies—a common restriction in hardware design. (To capture I/O dependencies that that may change in time, *stateful* PD interfaces are needed; cf. [6].)
3. In the third restricted class, the *total-and-rectangular nets*, the processes do not restrict the accepted input valuations, and there are no I/O dependencies. An example of such processes are the input-enabling Moore machines. An appropriate interface theory are the stateless I/O interfaces.

4.2 The rectangular nets

A process a is *rectangular* if there exist a predicate ϕ_a on I_a and a predicate ψ_a on O_a such that ρ_a is equivalent to $\phi_a \wedge \psi_a$. In other words, the I/O relation of a rectangular process may constrain input and output values, but it cannot relate them. It follows that there are no dependencies between input and output values. A relational net f is a *rectangular net* if all atoms in A_f are rectangular. The rectangular nets are closed under composition and connection.

For a relational net f, let $\rho_f^A = \bigwedge_{a \in A_f} \rho_a$ and $\rho_f^C = \bigwedge_{(x,y) \in C_f} (x = y)$ and $\rho_f = \rho_f^A \wedge \rho_f^C$. The predicate ρ_f is true precisely at the I/O valuations in $[P_f]$ that are consistent with f. Given a relational net f and a stateless I/O interface F, define $f \lhd_{I/O} F$ iff $I_f \subseteq I_F$ and $O_f \subseteq O_F^+$ and $O_f \supseteq O_F$. Given a relational net f and a stateless A/G interface F, define $f \lhd_{A/G} F$ iff (1) $f \lhd_{I/O} \Pi_F$, (2) $(\exists O_f)\rho_f$ is implied by the input assumption ϕ_F, and (3) $(\exists I_f)\rho_f$ implies the output guarantee ψ_F.

Proposition 5. *The stateless A/G interfaces with $\lhd_{A/G}$ are an interface theory for the rectangular nets, but not for the relational nets.*

To see the second part of the proposition, consider the inverter process $a_{\neq}$ with boolean input port x and boolean output port y and I/O relation $x \neq y$. This process is not rectangular. The illegal connection $a_{\neq}\theta$ for $\theta = \{(y, x)\}$ is permitted by the stateless A/G interfaces with $\lhd_{A/G}$.

4.3 The total nets

A process a is *total* if $(\forall I_a)(\exists O_a)\rho_a$. In other words, a total process accepts all possible input values. A relational net f is a *total net* if all processes in A_f are total. The total nets are closed under composition and connection.

For a relational net f, the *transitive-dependency relation* κ_f^* is a binary relation on the ports P_f, namely, the smallest transitive relation such that (1) for all processes $a \in A_f$, if $x \in I_f$ and $y \in O_f$, then $(x, y) \in \kappa_f^*$, and (2) $C_f \subseteq \kappa_f^*$. Given a relational net f and a stateless PD interface F, define $f \lhd_{PD} F$ iff $f \lhd_{I/O} \Pi_F$ and $\kappa_f^* \cap (I_F \times O_F) \subseteq \kappa_F$.

Proposition 6. *The stateless PD interfaces with $\lhd_{PD}$ are an interface theory for the total nets, but not for the relational nets.*

To see the second part of the proposition, consider the relational net f with two processes a and b, and no channels. Suppose that a has a boolean output port x and the I/O relation $x = 0$, and b has a boolean input port y and the I/O relation $y = 1$. The process b is not total. The illegal connection $f\theta$ for $\theta = \{(x, y)\}$ is permitted by the stateless PD interfaces with $\lhd_{PD}$.

4.4 The total-and-rectangular nets

A relational net f is a *total-and-rectangular net* if f is both a total net and a rectangular net. Note that a process a is both total and rectangular iff the I/O relation ρ_a contains no input ports from I_a. In other words, the I/O relation of a total rectangular process constrains only the output values. The total-and-rectangular nets are closed under composition and connection.

Proposition 7.

1. *The stateless I/O interfaces with $\lhd_{I/O}$ are an interface theory for the total-and-rectangular nets, equally expressive as the stateless A/G interfaces with $\lhd_{A/G}$, and more expressive than the stateless PD interfaces with $\lhd_{PD}$.*
2. *The stateless I/O interfaces with $\lhd_{I/O}$ are not an interface theory for the total nets, nor for the rectangular nets.*

To see that the stateless PD interfaces with $\lhd_{PD}$ are not as expressive for the total-and-rectangular nets as the stateless I/O interfaces with $\lhd_{I/O}$, consider the constant process a_0 with boolean input port x and boolean output port y and I/O relation $y = 0$. This process is both total and rectangular. The connection $a_0\theta$ with $\theta = \{(y, x)\}$ is permitted by the stateless I/O interfaces with $\lhd_{I/O}$, but is not permitted by the stateless PD interfaces with $\lhd_{PD}$. To see the second part of the proposition, first recall the inverter process $a_{\neq}$ from the proof of Proposition 5. The process $a_{\neq}$ is total. The illegal connection $a_{\neq}\theta$ for $\theta = \{(y, x)\}$ is permitted by the stateless I/O interfaces with $\lhd_{I/O}$. Second, recall the two-process net f from the proof of Proposition 6. Both processes $a, b \in A_f$ are rectangular. The illegal connection $f\theta$ for $\theta = \{(x, y)\}$ is permitted by the stateless I/O interfaces with $\lhd_{I/O}$.

5 Stateful Interfaces are Games

We present two interface algebras with state. A stateful interface F proceeds in steps through a state space, and with every step, the valuation on the set P_F of ports may change. The stateful interfaces we consider are *deterministic*, in that for every state the port valuation, which is observable, uniquely determines the successor state. Deterministic interfaces may of course be implemented by nondeterministic components. Nondeterminism in interfaces, however, seems unnecessary and is expensive: if an interface is not at all times aware of the state of the other interfaces within a component-based system, then compatibility checking for interfaces becomes difficult [5] (an exponential subset construction is needed to track the possible states of interfaces).

Our first example of stateful interfaces are the deterministic A/G interfaces. They have input assumptions and output guarantees that depend on the state of the interface. A stateful interface is naturally viewed as a two-player concurrent game [2]. The two players are *component*, representing a component that implements the interface, and *environment*, representing the environment. In the case of a stateful A/G interface, with every step, the component chooses an output valuation that satisfies the current output guarantee, and *independently* the environment chooses an input valuation that satisfies the current input assumption. The resulting I/O valuation determines the successor state, and the game repeats. The objective of the environment is to be helpful by always providing inputs that are accepted by the component.

There are two ways for the environment to win the game: either the game continues ad infinitum, or it enters a state in which the component cannot produce an output, because the output guarantee is unsatisfiable. Such a state is called a *termination state*, because it represents the fact that the component terminates. Conversely, the environment loses the game if the game enters a state in which the environment cannot provide an acceptable input, because the input assumption is unsatisfiable, whereas the component is not ready to terminate. Such a state is called an *immediate error state.* A helpful environment tries to prevent an immediate error state from being entered. The states from which the environment cannot prevent this are called *derived error states.* In other words, the derived error states are those states from which the environment has no strategy to avoid an immediate error state. As the objective is to avoid a set of states, this is a *safety* game [2]. The interface that gives rise to the game is well-formed iff the initial state is not an (immediate or derived) error state: if this is the case, then there is a helpful environment that can provide acceptable inputs either ad infinitum, or until the component terminates.

Given an interface F, a connection $F\theta$ is defined iff the resulting interface is well-formed. In particular, checking if $F\theta$ is defined requires computing the derived error states, i.e., solving a concurrent safety game. This can be done by backward-search from the immediate error states in time linear in the number of states. The game-theoretic view also motivates the definition of hierarchy: if the interface F' refines the interface F, then the well-formedness of $F\theta$ must imply the well-formedness of $F'\theta$; that is, if the environment has a strategy to win the $F\theta$ game, then it must also have a strategy to win the $F'\theta$ game. This preservation of strategies is captured by *alternating* refinement relations [3], such as alternating simulation (an alternative choice is refinement as alternating trace containment, but this would be more expensive to check).

The stateful A/G interfaces can be viewed as *synchronous interface automata*, thus complementing the asynchronous variety defined in [5]. In a similar fashion, state can be added also to the stateless I/O interfaces and to the stateless PD interfaces. Instead, we present as second example a more generic stateful interface algebra, the deterministic game interfaces, which make the game-theoretic view explicit. In a deterministic game interface, at every state, certain moves are available to each player. These moves can be very general, such as functions from input valuations to output valuations for the component, and functions from output valuations to input valuations for the environment. With every step, each player independently chooses an available move, and the combination of both moves determines the current port valuation, which in turn determines the successor state.

5.1 The deterministic assume/guarantee interfaces

A *deterministic A/G interface* F consists of the following:

- A stateless I/O interface Π_F.[2]
- A finite set Q_F of *states*, including an *initial state* $\hat{q}_F \in Q_F$.
- For every state $q \in Q_F$, a predicate $\phi_F(q)$ on I_F called *input assumption*, and a predicate $\psi_F(q)$ on O_F called *output guarantee*. A state $q \in Q_F$ is a *termination state* if the output guarantee $\psi_F(q)$ is unsatisfiable. A state $q \in Q_F$ is an *error state* if the input assumption $\phi_F(q)$ is unsatisfiable and the output guarantee $\psi_F(q)$ is satisfiable. We require that the initial state $\hat{q}_F$ is not an error state.
- For every pair $q, r \in Q_F$ of states, a predicate $\rho_F(q,r)$ on P_F called *transition guard*. We require that for every state $q \in Q_F$, (1) the disjunction $\bigvee_{r \in Q_F} \rho_F(q,r)$ is valid, and (2) for all states $r, r' \in Q_F$, if $r \neq r'$, then the conjunction $\rho_F(q,r) \wedge \rho_F(q,r')$ is unsatisfiable. Condition (2) ensures determinism. In other words, for every state $q \in Q_F$ and every I/O valuation $p \in [P_F]$, there is a unique state $r \in Q_F$ with $\rho_F(q,r)@p$. If $\rho_F(q,r)@p$, then r is called the *p-successor* of q and denoted $\delta_F(q,p)$.

For the deterministic A/G interfaces, composition, connection, and hierarchy are defined as follows:

`Composition` $F \| G$ of two deterministic A/G interfaces F and G is defined iff $\Pi_F \| \Pi_G$ is defined. Then, $Q_{F\|G} = Q_F \times Q_G$ and $\hat{q}_{F\|G} = (\hat{q}_F, \hat{q}_G)$ and for all states $q_F, r_F \in Q_F$ and $q_G, r_G \in Q_G$, let $\phi_{F\|G}(q_F, q_G) = \phi_F(q_F) \wedge \phi_G(q_G)$ and $\psi_{F\|G}(q_F, q_G) = \psi_F(q_F) \wedge \psi_G(q_G)$ and $\rho_{F\|G}((q_F, q_G), (r_F, r_G)) = \rho_F(q_F, r_F) \wedge \rho_G(q_G, r_G)$.

`Connection` $F\theta$ for a deterministic A/G interface F is defined iff (1) $\Pi_F\theta$ is defined and (2) the initial state of $F\theta$, as defined next, is not an error state. Let $Q_{F\theta} = Q_F$ and $\hat{q}_{F\theta} = \hat{q}_F$ and for all states $q, r \in Q_{F\theta}$, let $\psi_{F\theta}(q) = \psi_F(q) \wedge \rho_\theta$ and $\rho_{F\theta}(q,r) = \rho_F(q,r)$. The input assumptions $\phi_{F\theta}$ are computed by the following algorithm, which finds all derived error states:

> `[Step 1]` For all states $q \in Q_{F\theta}$, initialize $\phi_{F\theta}(q)$ to the predicate $(\forall O_{F\theta})(\psi_{F\theta}(q) \Rightarrow \phi_F(q))$.
>
> `[Step 2]` For all states $q, r \in Q_F$, if $\phi_{F\theta}(r)$ is unsatisfiable and $\psi_{F\theta}(r)$ is satisfiable, then replace $\phi_{F\theta}(q)$ by the conjunction of $\phi_{F\theta}(q)$ and $(\forall O_{F\theta})(\psi_{F\theta}(q) \Rightarrow \neg\rho_{F\theta}(q,r))$. In other words, if r is an error

[2] In a more general interface algebra, the input and output ports may depend on the state of the interface. This is necessary for modeling bidirectional ports (cf. [7]).

state, then a helpful environment chooses, in every state q, an input valuation that prevents r from being entered.

Repeat `[Step 2]` until all input assumptions are replaced by equivalent predicates.

The second step must be iterated, because the strengthening of an input assumption $\phi_{F\theta}(q)$ in `[Step 2]` may cause it to change from satisfiable to unsatisfiable, possibly exposing q as an error state and thus triggering additional changes.

`Hierarchy` $F' \preceq F$ iff (1) $\Pi_{F'} \preceq \Pi_F$ and (2) there is an alternating A/G simulation $\sqsubseteq$ of F' by F with $\hat{q}_{F'} \sqsubseteq \hat{q}_F$. An *alternating A/G simulation* of F' by F is a binary relation $\sqsubseteq$ between the states $Q_{F'}$ and the states Q_F such that $q' \sqsubseteq q$ implies that (1) the input assumption $\phi_{F'}(q')$ is implied by $\phi_F(q)$, (2) the output assumption $\psi_{F'}(q')$ implies $\psi_F(q)$, and (3) for all input valuations $i \in [I_F]$ and all output valuations $o \in [O_{F'}]$, if $\phi_F(q)@i$ and $\psi_{F'}(q')@o$, then $\delta_{F'}(q', i \uplus o) \sqsubseteq \delta_F(q, i \uplus o)$.

Proposition 8. *The deterministic A/G interfaces are an interface algebra.*

5.2 The deterministic game interfaces

A *deterministic game interface* F consists of the following:

- A set P_F of *ports*.
- A finite set Q_F of *states*, including an *initial state* $\hat{q}_F \in Q_F$.
- For every state $q \in Q_F$, a finite set $\mathcal{I}_F(q)$ of *input moves*, a finite set $\mathcal{O}_F(q)$ of *output moves*, a function $\gamma_F(q)$ from $\mathcal{I}_F(q) \times \mathcal{O}_F(q)$ to $[P_F]$ called *outcome function*, and a function $\delta_F(q)$ from $[P_F]$ to Q_F called *successor function*. Determinism means that the successor state depends only on the port valuation, and not on the choice of moves. A state $q \in Q_F$ is a *termination state* if the set $\mathcal{O}_F(q)$ of output moves is empty. A state $q \in Q_F$ is an *error state* if the set $\mathcal{I}_F(q)$ of input moves is empty and the set $\mathcal{O}_F(q)$ of output moves is nonempty. We require that the initial state $\hat{q}_F$ is not an error state.

For the deterministic game interfaces, composition, connection, and hierarchy are defined as follows:

`Composition` $F\|G$ of two deterministic game interfaces F and G is defined iff $P_F \cap P_G = \emptyset$. Then, $P_{F\|G} = P_F \cup P_G$ and $Q_{F\|G} = Q_F \times Q_G$ and $\hat{q}_{F\|G} = (\hat{q}_F, \hat{q}_G)$ and for all states $q_F, r_F \in Q_F$ and $q_G, r_G \in Q_G$, let $\mathcal{I}_{F\|G}(q_F, q_G) = \mathcal{I}_F(q_F) \times \mathcal{I}_G(q_G)$ and $\mathcal{O}_{F\|G}(q_F, q_G) = \mathcal{O}_F(q_F) \times \mathcal{O}_G(q_G)$ and for all moves $m_F \in \mathcal{I}_F$, $m_G \in \mathcal{I}_G$, $n_F \in \mathcal{O}_F$, and $n_G \in \mathcal{O}_G$, let $\gamma_{F\|G}(q_F, q_G)((m_F, m_G), (n_F, n_G)) = \gamma_F(q_F)(m_F, n_F) \uplus \gamma_G(q_G)(m_G, n_G)$, and for all valuations $p_F \in [P_F]$ and $p_G \in [P_G]$, let $\delta_{F\|G}(q_F, q_G)(p_F \uplus p_G) = (\delta_F(q_F)(p_F), \delta_G(q_G)(p_G))$.

`Connection` $F\theta$ for a deterministic game interface F is defined iff the initial state of $F\theta$, as defined next, is not an error state. Let $Q_{F\theta} = Q_F$ and $\hat{q}_{F\theta} = \hat{q}_F$ and for all states $q \in Q_{F\theta}$, let $\mathcal{O}_{F\theta}(q) = \mathcal{O}_F(q)$ and for all valuations $p \in [P_{F\theta}]$, let $\delta_{F\theta}(q)(p) = \delta_F(q)(p)$. The outcome functions $\gamma_{F\theta}$ are the uniquely determined functions such that for all states $q \in Q_{F\theta}$, all input moves $m \in \mathcal{I}_F(q)$, and all output moves $n \in \mathcal{O}_{F\theta}(q)$, we have (1) $\gamma_{F\theta}(q)(m,n) \cong \gamma_F(q)(m,n)$ and (2) for all channels $(x,y) \in \theta$, we have $\gamma_{F\theta}(q)(m,n)(x) = \gamma_F(q)(m,n)(y)$. The input moves $\mathcal{I}_{F\theta}$ are computed by the following algorithm, which finds all derived error states:

`[Step 1]` For all states $q \in Q_{F\theta}$, initialize $\mathcal{I}_{F\theta}(q)$ to the set of input moves $m \in \mathcal{I}_F(q)$ such that for all output moves $n \in \mathcal{O}_{F\theta}(q)$ and all channels $(x,y) \in \theta$, we have $\gamma_{F\theta}(q)(m,n)(x) = \gamma_{F\theta}(q)(m,n)(y)$.

`[Step 2]` For all states $q, r \in Q_F$, if $\mathcal{I}_{F\theta}(r)$ is empty and $\mathcal{O}_{F\theta}(r)$ is nonempty and there is an output move $n \in \mathcal{O}_{F\theta}(q)$ and a valuation $p \in [P_{F\theta}]$ such that $\gamma_{F\theta}(q)(m,n) = p$ and $\delta_{F\theta}(q)(p) = r$, then remove m from the set $\mathcal{I}_{F\theta}(q)$ of input moves. In other words, if r is an error state, then a helpful environment chooses, in every state q, an input move that prevents r from being entered.

Repeat `[Step 2]` until no input moves can be removed.

The second step must be iterated, because the removal of input moves in `[Step 2]` may cause a set $\mathcal{I}_{F\theta}(q)$ of input moves to change from nonempty to empty, possibly exposing q as an error state and thus triggering additional changes.

`Hierarchy` $F' \preceq F$ iff there is an alternating game simulation $\sqsubseteq$ of F' by F with $\hat{q}_{F'} \sqsubseteq \hat{q}_F$. An *alternating game simulation* of F' by F is a binary relation $\sqsubseteq$ between the states $Q_{F'}$ and the states Q_F such that $q' \sqsubseteq q$ implies that for all input moves $m \in \mathcal{I}_F(q)$, there is an input move $m' \in \mathcal{I}_{F'}(q')$ such that for all output moves $n' \in \mathcal{O}_{F'}(q')$, there is an output move $n \in \mathcal{O}_F(q)$ such that for all valuations $p' \in [P_{F'}]$ and $p \in [P_F]$, if $p' = \gamma_{F'}(q')(m',n')$ and $p = \gamma_F(q)(m,n)$, then $p' \cong p$ and $\delta_{F'}(q')(p') \sqsubseteq \delta_F(q)(p)$.

Proposition 9. *The deterministic game interfaces are an interface algebra.*

Acknowledgment. We thank Edward Lee for inspiring much of this research with his ideas on type theories for component interaction [11], and his project for developing an abstract semantics for Ptolemy II [12].

References

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

[2] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *38th Symp. Foundations of Computer Science*, pp. 100–109. IEEE Computer Society Press, 1997.

[3] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *Concurrency Theory*, LNCS 1466, pp. 163–178. Springer-Verlag, 1998.

[4] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[5] L. de Alfaro and T.A. Henzinger. Interface automata. In *9th Symp. Foundations of Software Engineering.* ACM Press, 2001.

[6] L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. The control of synchronous systems. In *Concurrency Theory*, LNCS 1877, pp. 458–473. Springer-Verlag, 2000.

[7] L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. The control of synchronous systems, part II. In *Concurrency Theory*, LNCS, Springer-Verlag, 2001.

[8] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* MIT Press, 1989.

[9] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Int. Conf. Computer-aided Design*, pp. 245–252. IEEE Computer Society Press, 2000.

[10] L. Lamport. The temporal logic of actions. *ACM Trans. Programming Languages and Systems*, 16:872–923, 1994.

[11] E.A. Lee. What's ahead for embedded software? *IEEE Computer Magazine*, 33:18–26, 2000.

[12] E.A. Lee. *Overview of the Ptolemy Project.* Technical Memorandum UCB/ERL-M01/11, University of California, Berkeley, 2001.

[13] N.A. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, 1996.

[14] J. Rumbaugh, G. Booch, and I. Jacobson. *The UML Reference Guide.* Addison-Wesley, 1999.

[15] N. Shankar. Lazy compositional verification. In *Compositionality: The Significant Difference*, LNCS 1536, pp. 541–564. Springer-Verlag, 1999.