Synchronous and Bidirectional Component Interfaces*

Arindam Chakrabarti[1], Luca de Alfaro[2], Thomas A. Henzinger[1], and Freddy Y.C. Mang[3]

[1] Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720-1770, USA. {arindam,tah}@eecs.berkeley.edu
[2] Department of Computer Engineering, University of California, Santa Cruz, CA 95064, USA. luca@soe.ucsc.edu
[3] Advanced Technology Group, Synopsys Inc. fmang@synopsys.com

Abstract. We present *interface models* that describe both the input assumptions of a component, and its output behavior. By enabling us to check that the input assumptions of a component are met in a design, interface models provide a *compatibility* check for component-based design. When refining a design into an implementation, interface models require that the output behavior of a component satisfies the design specification only when the input assumptions of the specification are satisfied, yielding greater flexibility in the choice of implementations. Technically, our interface models are games between two players, Input and Output; the duality of the players accounts for the dual roles of inputs and outputs in composition and refinement. We present two interface models in detail, one for a simple synchronous form of interaction between components typical in hardware, and the other for more complex synchronous interactions on bidirectional connections. As an example, we specify the interface of a bidirectional bus, with the input assumption that at any time at most one component has write access to the bus. For these interface models, we present algorithms for compatibility and refinement checking, and we describe efficient symbolic implementations.

1 Introduction

One of the main applications of modeling formalisms is to capture designs. We present *interface models* that are specifically geared to support the component-based approach to design. Interface models describe both the inputs that can be accepted by a component, and the outputs it can generate. As an interface constrains the acceptable inputs, the underlying component fits into some design contexts (which meet the constraints), but not into others. Interface models provide a means for answering four questions that arise in component-based design:

* This research was supported in part by the AFOSR grant F49620-00-1-0327, the DARPA grant F33615-00-C-1693, the MARCO grant 98-DT-660, the NSF grant CCR-9988172, the SRC grant 99-TJ-683.003, and the NSF CAREER award CCR-0132780.

the *well-formedness question* (can a component be used in some design, i.e., are the input constraints satisfiable?), the *verification question* (does a component satisfy a given property in all designs?), the *compatibility question* (do two components interact in compatible ways in a design?), and the *refinement question* (can a component be substituted for another one in every design context without violating compatibility?).

For each of the questions of well-formedness, verification, compatibility, and refinement, there are two basic choices for treating inputs and outputs. The *graph* view quantifies inputs and outputs with the same polarity; the *game* view quantifies inputs and outputs with opposite polarities. In the graph view, both inputs and outputs can be seen as labels in a nondeterministic state transition graph; in the game view, inputs and outputs are chosen by different players and the result of each combination of choices determines the state transition. For example, the graph view is appropriate for the verification question: does a component satisfy a given property for all acceptable inputs and all possible outputs? On the other hand, the game view is necessary for the well-formedness question [1, 11]: are there acceptable inputs for all possible choices of outputs? We argue that also for compatibility and refinement, the game view is the appropriate one.

1.1 The graph view

The graph view is taken by many process algebras (e.g., [12, 17]) and state-based models (e.g., [15, 13, 7, 4]). These frameworks are aimed at verification. Indeed, also refinement is typically viewed as a verification question: does a more detailed description of a component generate only behaviors that are permitted by a more abstract description? Refinement is usually defined as a form of trace containment or simulation: when quantifying universally over both inputs and outputs, we say that a component N refines a component M (written $N \preceq M$) if, for all input and output choices, the behaviors of N are a subset of those of M. In particular, N can only produce outputs that are also produced by M, and N can only accept inputs that are also accepted by M. This ensures that every language-theoretic property (such as safety) that holds for M also holds for N. The graph view of refinement, however, becomes problematic when we interpret refinement as substitutivity. The output clause is still appropriate: by requiring that the output behavior of N is a subset of that of M, it ensures that if the outputs of M can be accepted by the other components of the design, so can those of N. The input clause instead is questionable: it states that the implementation N should be able to accept a *subset* of the inputs accepted by the specification M. This raises the possibility that, when N is substituted for M in a design, N cannot accept some inputs from other components that could be accepted by M. Hence, *substitutivity of refinement* does not hold in the graph view. Indeed, in process algebras and the modeling language SMV [7], if $N \preceq M$ and $M \| P$ is deadlock-free, it is possible that $N \| P$ deadlocks [3]. To remedy this situation, some models, such as *I/O automata* [14] and *reactive modules* [4], require that components be able to accept *all* possible inputs; this condition is known as *input-enabledness* or *receptivity*. This requirement forces models to

specify the outputs generated in response to *all* possible inputs, including inputs that the designers know cannot occur in the actual design.

The graph view is also limited in its capability to analyze component compatibility. If models specify explicitly which inputs can be accepted, and which ones are *illegal*, then it is possible to ask the compatibility question generically: do illegal inputs occur? If we quantify universally over both inputs and outputs, we obtain a verification question: two components M and N are compatible if, once composed, they accept all inputs. This is not a natural phrasing of the compatibility question: it requires $M \| N$ to accept all inputs, even though M and N could have illegal inputs. A more compositional definition is to call M and N compatible if there are *some* input sequences that ensure that all illegal inputs of M and N are avoided, and to label all other sequences as illegal for $M \| N$. This definition of compatibility leads to a dual treatment of inputs (quantified existentially) and outputs (quantified universally), and to the game view.

1.2 The game view

According to the game view, inputs and outputs play dual roles. In trace theory [11], a trace model consists in two sets, of accepted and rejected traces, and games are used to solve the realizability and compatibility questions. In the game semantics of [2, 3] and the interface models of [9, 10], components are explicitly modeled as games between two players, Input and Output. The moves of Input represent the inputs that can be accepted, and the moves of Output the outputs that can be generated. To model the fact that these sets can change in time, after the input and output moves are chosen, the game moves to a new state, with possibly different sets of accepted inputs and possible outputs.

In the study of compatibility, game-based approaches quantify inputs existentially, and outputs universally. When two components M and N are composed, their composition may have illegal states, where one component emits outputs that are illegal inputs for the other one. Yet, M and N are considered *compatible* as long as there is *some* input behavior that ensures that, for all output behaviors, the illegal states are avoided: in other words, M and N are compatible if there is some environment in which they can be used correctly together. In turn, the input behaviors that ensure compatibility constitute the legal behaviors for the composition $M \| N$: when composing component models, both the possible output behaviors, and the legal input behaviors, are composed.

The game view leads to an *alternating* view of refinement [5]: a more detailed component N refines an abstract component M if all legal inputs of M are also legal for N and if, when M and N are subject to the same (legal) inputs, N generates output behaviors that are a subset of those of M. This definition ensures that, whenever $N \preceq M$, we can substitute N for M in every design without creating any incompatibility: in the game view, substitutivity of refinement holds. The alternating definition of refinement also mirrors the contravariant definition of subtyping in programming languages, which also supports substitutivity [18]. Indeed, the game framework can be viewed as a generalization of type theory to behaviors.

1.3 Synchronous interface models

In this paper, we adopt the game view to modeling, and we introduce two interface models for *synchronous* components. We begin with the simple model of *Moore interfaces*: in addition to the usual transition relation of a synchronous system, which describes the update rules for the outputs, a Moore interface has a symmetrically-defined transition relation for the inputs, which specifies which input transitions are acceptable. Our second model, *bidirectional interfaces*, illustrate how game-based models can be richer than their graph-based counterparts. Bidirectional connections cannot be modeled in the input-enabled setting: there are always environments that use such connections as input, and environments that use them as output, so that no component can work in all environments. Bidirectional connections, however, can be naturally modeled as a game between Input and Output players. As an example, we encode the access protocol to the PCI bus, in which several components share access to a multi-directional bus. By checking the compatibility of the component models, we can ensure that no conflicts for bus access arise. We have implemented tools for symbolic compatibility and refinement checking for both Moore and bidirectional interfaces, and we discuss how the game-based algorithms can be implemented with minor modifications to the usual symbolic machinery for graph-based algorithms, and yield a similar efficiency.

2 Compatibility and Composition

2.1 Moore interfaces

Moore interfaces model both the behavior of a system component, and the interface between the component and its environment. The state of a module is described by a set of *state variables,* partitioned into sets of *input* and *output* variables. The possible changes of output variables are described by an *output transition relation,* while the legal changes of input variables are described by an *input transition relation.* Hence, the output transition relation describes the module's behavior, and the input transition relation describes the input assumptions of the interface.

Example 1 We illustrate the features of Moore interfaces by modeling a ± 1 adder driven by a binary counter. The adder *Adder* has two control inputs q_0 and q_1, data inputs $i_7 \cdots i_0$, and data outputs $o_7 \cdots o_0$. When $q_0 = q_1 = 1$, the adder leaves the input unchanged: the next value of $o_7 \cdots o_0$ is equal to $i_7 \cdots i_0$. When $q_0 = 0$ and $q_1 = 1$, the next outputs are given by $[o'_7 \cdots o'_0] = [i_7 \cdots i_0]+1 \bmod 2^8$, where primed variables denote the values at the next clock cycle, and $[o'_7 \cdots o'_0]$ is the integer encoded in binary by $o'_7 \cdots o'_0$. Similarly, when $q_1 = 0$ and $q_0 = 1$, we have $[o'_7 \cdots o'_0] = [i_7 \cdots i_0]-1 \bmod 2^8$. The adder is designed with the assumption that q_1 and q_0 are not both 0: hence, the input transition relation of *Adder* states that $q'_0 q'_1 \neq 00$. In order to cycle between adding $0, +1, -1$, the control inputs q_0 and q_1 are connected to the outputs q_1 and q_0 of a two-bit count-to-zero counter

Counter. The counter has only one input, cl: when $cl = 0$, then $q_1'q_0' = 11$; otherwise, $[q_1'q_0'] = [q_1q_0] - 1 \bmod 4$.

When we compose *Counter* and *Adder*, we synthesize for their composition $Counter \| Adder$ a new input assumption, that ensures that the input assumptions of both *Counter* and *Adder* are satisfied. To determine the new input assumption, we solve a game between Input, which chooses the next values of cl and $i_7 \cdots i_0$, and Output, which chooses the next values of q_0, q_1, and $o_7 \cdots o_0$. The goal of Input is to avoid a transition to $q_1q_0 = 00$. At the states where $q_1q_0 = 01$, Input can win if $cl = 0$, since we will have $q_1'q_0' = 11$; but Input cannot win if $cl = 1$. By choosing $cl' = 0$, Input can also win from the states where $q_1q_0 = 10$. Finally, Input can always win from $q_1q_0 = 11$, for all cl'. Thus, we associate with $Counter \| Adder$ a new input assumption encoded by the transition relation requiring that whenever $q_1q_0 = 10$, then $cl' = 0$. The input requirement $q_1q_0 \neq 00$ of the adder gives rise, in the composite system, to the requirement that the reset-to-1 occurs early in the count-to-zero cycle of the counter. ∎

Given a set $\mathcal{W}$ of typed variables with finite domain, a state s over $\mathcal{W}$ is a function that assigns to each $x \in \mathcal{W}$ a value $s[\![x]\!]$ of the appropriate type; we write $\mathcal{S}[\mathcal{W}]$ for the set of all states over $\mathcal{W}$. We denote by $\mathcal{W}' = \{x' \mid x \in \mathcal{W}\}$ the set obtained by priming each variable in $\mathcal{W}$; given a predicate φ on $\mathcal{W}$, we denote by φ' the predicate on $\mathcal{W}'$ obtained by replacing in φ every $x \in \mathcal{W}$ with $x' \in \mathcal{W}'$. Given a state $s \in \mathcal{S}[\mathcal{W}]$ and a predicate φ on $\mathcal{W}$, we write $s \models \varphi$ if φ is satisfied under the variable interpretation specified by s. Given two states $s, s' \in \mathcal{S}[\mathcal{W}]$ and a predicate φ on $\mathcal{W} \cup \mathcal{W}'$, we write $(s, s') \models \varphi$ if φ is satisfied by the interpretation that assigns to $x \in \mathcal{W}$ the value $s[\![x]\!]$, and to $x' \in \mathcal{W}'$ the value $s'[\![x]\!]$. Moore interfaces are defined as follows.

Definition 1 (Moore interface) A *Moore interface* $M = \langle \mathcal{V}_M^i, \mathcal{V}_M^o, \theta_M^i, \theta_M^o, \tau_M^i, \tau_M^o \rangle$ consists of the following components:

- A finite set $\mathcal{V}_M^i$ of *input variables,* and a finite set $\mathcal{V}_M^o$ of *output variables.* The two sets must be disjoint; we define $\mathcal{V}_M = \mathcal{V}_M^i \cup \mathcal{V}_M^o$.
- A satisfiable predicate θ_M^i on $\mathcal{V}_M^i$ defining the legal initial values for the input variables, and a satisfiable predicate θ_M^o on $\mathcal{V}_M^o$ defining the initial values for the output variables.
- An *input transition predicate* τ_M^i on $\mathcal{V}_M \cup (\mathcal{V}_M^i)'$, specifying the legal updates for the input variables, and an *output transition predicate* τ_M^o on $\mathcal{V}_M \cup (\mathcal{V}_M^o)'$, specifying how the module can update the values of the output variables. We require that the formulas $\forall \mathcal{V}_M . \exists (\mathcal{V}_M^i)' . \tau_M^i$ and $\forall \mathcal{V}_M . \exists (\mathcal{V}_M^o)' . \tau_M^o$ hold. ∎

The above interfaces are called *Moore* because the next value of the output variables can depend on the current state, but not on the next value of the input variables, as in Moore machines. The requirements on the input and output transition relations ensure that the interface is non-blocking: from every state there is some legal input and possible output. Given a Moore interface $M = \langle \mathcal{V}_M^i, \mathcal{V}_M^o, \theta_M^i, \theta_M^o, \tau_M^i, \tau_M^o \rangle$, we let $Traces(\mathcal{V}_M^i, \mathcal{V}_M^o, \theta_M^i, \theta_M^o, \tau_M^i, \tau_M^o)$ be the set of *traces* of M, consisting of all the infinite sequences $s_0, s_1, s_2, \ldots$ of states of $\mathcal{S}[\mathcal{V}_M]$ such that $s_0 \models \theta_M^i \wedge \theta_M^o$, and $(s_k, s_{k+1}) \models \tau_M^i \wedge \tau_M^o$ for all $k \geq 0$.

Composition of Moore interfaces. Two Moore interfaces M and N are *composable* if $\mathcal{V}^o_M \cap \mathcal{V}^o_N = \emptyset$. If M and N are composable, we merge them into a single interface P as follows. We let $\mathcal{V}^o_P = \mathcal{V}^o_M \cup \mathcal{V}^o_N$ and $\mathcal{V}^i_P = (\mathcal{V}^i_M \cup \mathcal{V}^i_N) \setminus \mathcal{V}^o_P$. The output behavior of P is simply the joint output behavior of M and N, since each interface is free to choose how to update its output variables: hence, $\theta^o_P = \theta^o_M \wedge \theta^o_N$ and $\tau^o_P = \tau^o_M \wedge \tau^o_N$. On the other hand, we cannot simply adopt the symmetrical definition for the input assumptions. A syntactic reason is that $\theta^i_M \wedge \theta^i_N$ and $\tau^i_M \wedge \tau^i_N$ may contain variables in $(\mathcal{V}^o_P)'$. But a deeper reason is that we may need to strengthen the input assumptions of P further, in order to ensure that the input assumptions of M and N hold. If we can find such a further strengthening θ^i and τ^i, then M and N are said to be *compatible,* and $P = M \| N$ with θ^i_P and τ^i_P being the weakest such strengthenings; otherwise, we say that M and N are incompatible, and $M \| N$ is undefined. Hence, informally, M and N are compatible if they can be used together under some assumptions.

Definition 2 (Compatibility and composition of Moore interfaces) For any two Moore interfaces M and N, we say that M and N are *composable* if $\mathcal{V}^o_M \cap \mathcal{V}^o_N = \emptyset$. If M and N are composable, let $\mathcal{V}^o_P = \mathcal{V}^o_M \cup \mathcal{V}^o_N$, $\mathcal{V}^i_P = (\mathcal{V}^i_M \cup \mathcal{V}^i_N) \setminus \mathcal{V}^o_P$, $\mathcal{V}_P = \mathcal{V}^o_P \cup \mathcal{V}^i_P$, $\theta^o_P = \theta^o_M \wedge \theta^o_N$, and $\tau^o_P = \tau^o_M \wedge \tau^o_N$.

The interfaces M and N are *compatible* (written $M \wr\!\wr N$) if they are composable, and if there are predicates θ^i on $\mathcal{V}^i_P$ and τ^i on $\mathcal{V}_P \cup (\mathcal{V}^i_P)'$ such that (i) θ^i is satisfiable; (ii) $\forall \mathcal{V}_P . \exists (\mathcal{V}^i_P)' . \tau^i$ holds; (iii) for all $s_0, s_1, s_2, \ldots \in \mathit{Traces}(\mathcal{V}^i_P, \mathcal{V}^o_P, \theta^i, \theta^o_P, \tau^i, \tau^o_P)$ we have $s_0 \models \theta^i_M \wedge \theta^i_N$ and, for all $k \geq 0$, $(s_k, s_{k+1}) \models \tau^i_M \wedge \tau^i_N$.

The *composition* $P = M \| N$ is defined if and only if $M \wr\!\wr N$, in which case P is obtained by taking for the input predicate θ^i_P and for the input transition relation τ^i_P the weakest predicates such that the above condition holds. ∎

To compute $M \| N$, we consider a game between Input and Output. At each round of the game, Output chooses new values for the output variables $\mathcal{V}^o_P$ according to τ^o_P; simultaneously and independently, Input chooses (unconstrained) new values for the input variables $\mathcal{V}^i_P$. The goal of Input is to ensure that the resulting behavior satisfies $\theta^i_M \wedge \theta^i_P$ at the initial state, and $\tau^i_M \wedge \tau^i_N$ at all state transitions. If Input can win the game, then M and N are compatible, and the most general strategy for Input will give rise to θ^i_P and τ^i_P; otherwise, M and N are incompatible. The algorithm for computing θ^i_P and τ^i_P proceeds by computing iterative approximations to τ^i_P, and to the set C of states from which Input can win the game. We let $C_0 = \mathrm{T}$ and, for $k \geq 0$:

$$\widetilde{\tau}_{k+1} = \forall (\mathcal{V}^o_P)' . \big(\tau^o_P \rightarrow (\tau^i_M \wedge \tau^i_N \wedge C'_k)\big) \qquad C_{k+1} = C_k \wedge \exists (\mathcal{V}^i_P)' . \widetilde{\tau}_{k+1}. \qquad (1)$$

Note that $\widetilde{\tau}_{k+1}$ is a predicate on $\mathcal{V}^o_P \cup \mathcal{V}^i_P \cup (\mathcal{V}^i_P)'$. Hence, $\widetilde{\tau}_{k+1}$ ensures that, regardless of how $\mathcal{V}^o_P$ are chosen, from C_{k+1} we have that (i) for one step, τ^i_M and τ^i_N are satisfied; and (ii) the step leads to C_k. Thus, indicating by $C_* = \lim_{k \to \infty} C_k$ and $\widetilde{\tau}_* = \lim_{k \to \infty} \widetilde{\tau}_k$ the fixpoints of (1) we have that C_* represents the set of states from which Input can win the game, and $\widetilde{\tau}_*$ represents the most liberal Input strategy for winning the game. This suggests us to take $\tau^i_P = \widetilde{\tau}_*$.

However, this is not always the weakest choice, as required by Definition 2: a weaker choice is $\tau_P^i = \neg C_* \vee \widetilde{\tau}_*$, or equivalently $\tau_P^i = C_* \rightarrow \widetilde{\tau}_*$. Contrary to $\tau_P^i = \widetilde{\tau}_*$, this weaker choice ensures that the interface P is non-blocking. We remark that the choices $\tau_P^i = \widetilde{\tau}_*$ and $\tau_P^i = C_* \rightarrow \widetilde{\tau}_*$ differ only at non-reachable states. Since the state-space of P is finite, by monotonicity of (1) we can compute the fixpoint C_* and $\widetilde{\tau}_*$ in a finite number of iterations. Finally, we define the input initial condition of P by $\theta_P^i = \forall \mathcal{V}^o.(\theta_P^o \rightarrow (\theta_M^i \wedge \theta_N^i \wedge C_*))$. The following algorithm summarizes these results.

Algorithm 1 Given two composable Moore interfaces M and N, let $C_0 = \mathrm{T}$, and for $k > 0$, let the predicates C_k and $\widetilde{\tau}_k$ be as defined by (1). Let $\widetilde{\tau}_* = \lim_{k\rightarrow\infty} \widetilde{\tau}_k$ and $C_* = \lim_{k\rightarrow\infty} C_k$; the limits can be computed with a finite number of iterations, and let $\theta_*^i = \forall \mathcal{V}^o.\big(\theta_P^o \rightarrow (\theta_M^i \wedge \theta_N^i \wedge C_*)\big)$. Then the interfaces M and N are *compatible* iff θ_*^i is satisfiable; in this case their composition $P = M \| N$ is given by

$$\begin{array}{lll} \mathcal{V}_P^o = \mathcal{V}_M^o \cup \mathcal{V}_N^o & \tau_P^o = \tau_M^o \wedge \tau_N^o & \theta_P^o = \theta_M^o \wedge \theta_N^o \\ \mathcal{V}_P^i = (\mathcal{V}_M^i \cup \mathcal{V}_N^i) \setminus \mathcal{V}^o & \tau_P^i = C_* \rightarrow \widetilde{\tau}_* & \theta_P^i = \theta_*^i. \end{array} \quad \blacksquare$$

Implementation considerations. We have implemented composition and compatibility checking for Moore interfaces by extending the Mocha model checker [6] to interfaces. To obtain an efficient implementation, we represent both the input and the output transition relations using a conjunctively decomposed representation, where a relation τ is represented by a list of BDDs $\tau_1, \tau_2, \ldots, \tau_n$ such that $\tau = \wedge_{i=1}^n \tau_i$. When computing $P = M \| N$, the list for τ_P^o can be readily obtained by concatenating the lists for τ_M^o and τ_N^o. Moreover, assume that τ_P^o is represented as $\bigwedge_{i=1}^n \tau_i^o$, and that $\tau_M^i \wedge \tau_N^i$ is represented as $\bigwedge_{j=1}^m \tau_j^o$. Given C_k, from (1) we obtain the conjunctive decomposition $\bigwedge_{j=1}^{m+1} \widetilde{\tau}_{k+1,j}$ for $\widetilde{\tau}_{k+1}$ by taking $\widetilde{\tau}_{k+1,m+1} = \neg\exists(\mathcal{V}_P^o)'.(\tau_P^o \wedge \neg C_k')$ and, for $1 \leq j \leq m$, by taking $\widetilde{\tau}_{k+1,j} = \neg\exists(\mathcal{V}_P^o)'.(\tau_P^o \wedge \neg\tau_j^i)$. We also obtain $C_{k+1} = \exists(\mathcal{V}_P^i)'.\bigwedge_{j=1}^{m+1} \widetilde{\tau}_{k+1,j}$. All these operations can be performed using image computation techniques. Once we reach k such that $C_k \equiv C_{k+1}$, the BDDs $\widetilde{\tau}_{k,1}, \ldots, \widetilde{\tau}_{k,m+1}$ form a conjunctive decomposition for $\widetilde{\tau}_*$. Since the two transition relations $\widetilde{\tau}_*$ and $C_* \rightarrow \widetilde{\tau}_*$ differ only for the behavior at non-reachable states, in our implementation we take directly $\tau_P^i = \widetilde{\tau}_*$, obtaining again a conjunctive decomposition. With these techniques, the size (number of BDD variables) of the interfaces that our tool is able to check for compatibility, and compose, is roughly equivalent to the size of the models that Mocha [6] can verify with respect to safety properties.

2.2 Bidirectional Interfaces

Bidirectional interfaces model components that have bidirectional connections. To model bidirectionality we find it convenient to add to the Moore model a set Q of *locations*. Informally, each location $q \in Q$ partitions the interface variables into inputs and outputs, and determines what values are legal for the inputs, and what values can be assigned to the outputs. At each location $q \in Q$, a

particular choice of output and input values determines the successor location q'. The precise definition is as follows.

Definition 3 (Bidirectional interfaces) A *bidirectional interface* M is a tuple $\langle \mathcal{V}_M, Q_M, \hat{q}_M, v^o_M, \phi^i_M, \phi^o_M, \rho_M \rangle$ consisting of the following components:

- A finite set $\mathcal{V}_M$ of input or output *(inout)* variables.
- A finite set Q_M of locations, including an initial location $\hat{q}_M \in Q_M$.
- A function $v^o_M : Q_M \to 2^{\mathcal{V}_M}$, that associates with all $q \in Q_M$ the set $v^o_M(q)$ of variables that are used as outputs at location q. For all $q \in Q_M$, we denote by $v^i_M(q) = \mathcal{V}_M \setminus v^o_M(q)$ the set of variables that are used as inputs.
- Two labelings ϕ^i_M and ϕ^o_M, which associate with each location $q \in Q_M$ a predicate $\phi^i_M(q)$ on $v^i_M(q)$, called the input assumption, and a predicate $\phi^o_M(q)$ on $v^o_M(q)$, called the output guarantee. For all $q \in Q_M$, both $\phi^i_M(q)$ and $\phi^o_M(q)$ should be satisfiable.
- A labeling ρ_M, which associates with each pair of locations $q, r \in Q_M$ a predicate $\rho_M(q,r)$ on $\mathcal{V}_M$, called the *transition guard.* We require that for every location $q \in Q_M$, (i) the disjunction $\bigvee_{r \in Q_M} \rho_M(q,r)$ is valid and (ii) $\forall r, r' \in Q_M$, $(r \neq r') \Rightarrow \neg(\rho_M(q,r) \wedge \rho_M(q,r'))$. Condition (i) ensures that the interface is non-blocking, and condition (ii) ensures determinism. ∎

We let $\mathcal{V}^i_M = \bigcup_{q \in Q_M} v^i_M(q)$ and $\mathcal{V}^o_M = \bigcup_{q \in Q_M} v^o_M(q)$ be the sets of all variables that are ever used as inputs or outputs (note that we do not require $\mathcal{V}^i_M \cap \mathcal{V}^o_M = \emptyset$). We define the set $\mathit{Traces}(\langle \mathcal{V}_M, Q_M, \hat{q}_M, \phi^i_M, \phi^o_M, \rho_M \rangle)$ of *bidirectional traces* to be the set of infinite sequences $q_0, s_0, q_1, s_1, \ldots$, where $q_0 = \hat{q}_M$, and for all $k \geq 0$, we have $q_k \in Q_M$, $s_k \in \mathcal{S}[\mathcal{V}_M]$, and $s_k \models (\phi^i_M(q_k) \wedge \phi^o_M(q_k) \wedge \rho_M(q_k, q_{k+1}))$. For $q_0, s_0, q_1, s_1, \ldots \in \mathit{Traces}(\langle \mathcal{V}_M, Q_M, \hat{q}_M, \phi^i_M, \phi^o_M, \rho_M \rangle)$ and $k \geq 0$, we say that q_k is *reachable* in $\langle \mathcal{V}_M, Q_M, \hat{q}_M, \phi^i_M, \phi^o_M, \rho_M \rangle$.

Composition of bidirectional interfaces is defined along the same lines as for Moore interfaces. Local incompatibilities arise not only when one interface output values do not satisfy the input assumptions of the other, but also when the same variable is used as output by both interfaces. The formal definition follows.

Definition 4 (Composition of bidirectional interfaces)
Given two bidirectional interfaces M and N, let $\mathcal{V}_\otimes = \mathcal{V}_M \cup \mathcal{V}_N$, $Q_\otimes = Q_M \times Q_N$, and $\hat{q}_\otimes = (\hat{q}_M, \hat{q}_N)$. For all $(p,q) \in Q_M \times Q_N$, let $\phi^o_\otimes(p,q) = \phi^o_M(p) \wedge \phi^o_M(q)$, and for all $(p',q') \in Q_M \times Q_N$, let $\rho_\otimes((p,q),(p',q')) = \rho_M(p,p') \wedge \rho_N(p,p')$. The interfaces M and N are *compatible* (written $M \between N$) if there is a labeling ψ associating with all $(p,q) \in Q_\otimes$ a predicate $\psi(p,q)$ on $\mathcal{V}_\otimes \setminus (\mathcal{V}^o_M(p) \cup \mathcal{V}^o_N(q))$ such that (i) $\psi(p,q)$ is satisfiable at all $(p,q) \in Q_\otimes$, and (ii) all traces $(p_0,q_0), s_0, (p_1,q_1), s_1, (p_2,q_2), s_2, \ldots \in \mathit{Traces}(\mathcal{V}_\otimes, Q_\otimes, \hat{q}_\otimes, \psi, \phi^o_\otimes, \rho_\otimes)$ satisfy, for all $k \geq 0$, the conditions (a) $\mathcal{V}^o_M(p_k) \cap \mathcal{V}^o_N(q_k) = \emptyset$ and (b) $s_k \models \phi^i_M(p_k) \wedge \phi^i_N(q_k)$. The composition $P = M \| N$ is defined if and only if M and N are compatible; if they are, then $P = M \| N$ is obtained by taking for ϕ^i_P the weakest predicate ψ such that the above conditions (a) and (b) on traces hold, by taking for Q_P the subset of locations of $Q_\otimes$ that are reachable in $\langle \mathcal{V}_\otimes, Q_\otimes, \hat{q}_\otimes, \mathcal{V}^o_\otimes, \phi^i_P, \phi^o_\otimes, \rho_\otimes \rangle$, by taking $\mathcal{V}_P = \mathcal{V}_\otimes$ and $\hat{q}_P = \hat{q}_\otimes$, and by taking for $\mathcal{V}^o_P, \phi^i_P, \phi^o_P$, and ρ_P the restrictions of $\mathcal{V}^o_\otimes, \phi^i_\otimes, \phi^o_\otimes$, and $\rho_\otimes$ to Q_P. ∎

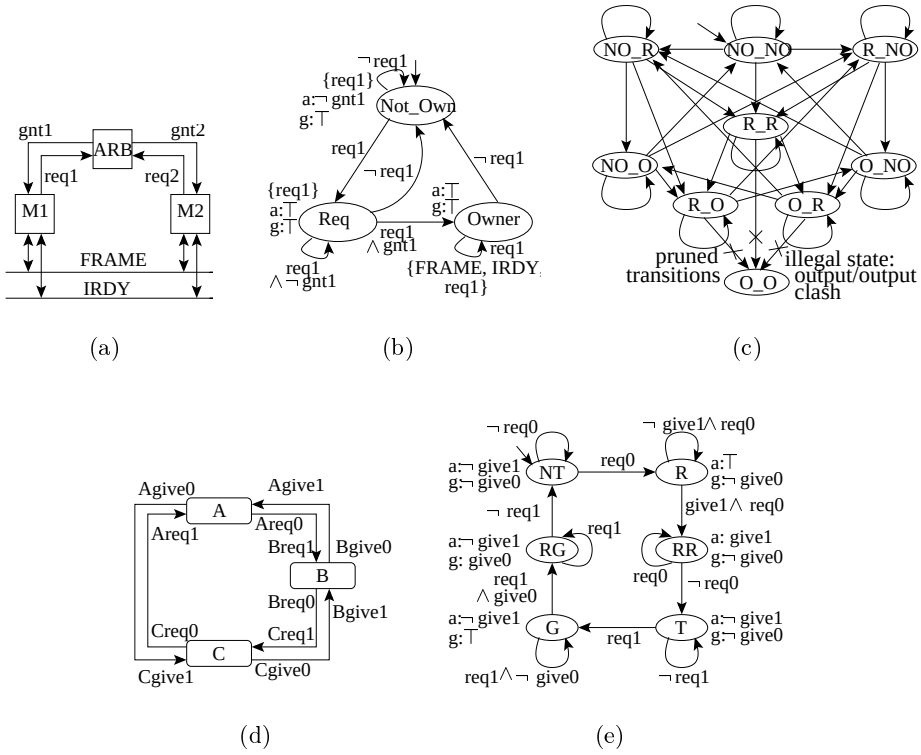
Algorithm 2 Given two bidirectional interfaces M and N, let $\mathcal{V}_\otimes = \mathcal{V}_M \cup \mathcal{V}_N$, $Q_\otimes = Q_M \times Q_N$, and $\hat{q}_\otimes = (\hat{q}_M, \hat{q}_N)$. For all $(p,q) \in Q_M \times Q_N$, let $\phi^i_\otimes(p,q) = \phi^i_M(p) \wedge \phi^i_M(q)$, and for all $(p',q') \in Q_M \times Q_N$, let $\rho_\otimes((p,q),(p',q')) = \rho_M(p,p') \wedge \rho_N(p,p')$. The input labeling $\phi^i_\otimes(p,q)$ is computed by repeating the following steps, that progressively strengthen the input assertions:

> [Step 1] For all $(p,q) \in Q_M \times Q_N$, if $v^o_M(p) \cap v^o_N(q) \neq \emptyset$, then initialize $\phi^i_\otimes(p,q)$ to F; otherwise initialize $\phi^i_\otimes(p,q)$ to the predicate $\forall v^o_\otimes(p,q).(\phi^o_\otimes(p,q) \rightarrow (\phi^i_M(p) \wedge \phi^i_N(q))$.
>
> [Step 2] For all (p,q) and (p',q') in $Q_M \times Q_N$, if $\phi^i_\otimes(p',q')$ is unsatisfiable, then replace $\phi^i_\otimes(p,q)$ with $\phi^i_\otimes(p,q) \wedge \forall v^o_\otimes(p,q).(\phi^o_\otimes(p,q) \rightarrow \neg\rho_\otimes((p,q),(p',q'))$.
>
> Repeat [Step 2] until all input assumptions are replaced by equivalent predicates, i.e., are not strengthened.

We have that $M \wr\!\wr N$ iff $\phi^i_\otimes(\hat{q}_M, \hat{q}_N)$ is satisfiable. If $M \wr\!\wr N$ then their composition P is defined by taking Q_P to be the subset of locations of $Q_\otimes$ that are reachable in $\langle \mathcal{V}_\otimes, Q_\otimes, \hat{q}_\otimes, v^o_\otimes, \phi^i_P, \phi^o_\otimes, \rho_\otimes \rangle$, by taking $\mathcal{V}_P = \mathcal{V}_\otimes$ and $\hat{q}_P = \hat{q}_\otimes$, and by taking for $v^o_P, \phi^i_P, \phi^o_P$, and ρ_P the restrictions of $v^o_\otimes, \phi^i_\otimes, \phi^o_\otimes$, and $\rho_\otimes$ to Q_P. ∎

We have developed and implemented symbolic algorithms for composition and compatibility and refinement checking of bidirectional interfaces. The tool, written in Java, is based on the CUDD Package used in JMocha [8]. In our implementation, the locations are represented explicitly, while the input assumptions and output guarantees at each location are represented and manipulated symbolically. This hybrid representation is well-suited to the modeling of bidirectional interfaces, where the set of input and output variables depends on the location.

Example 2 (PCI Bus) We consider a PCI bus configuration with two PCI-compliant master devices and a PCI arbiter as shown in Figure 1(a). Each PCI master device has an *gnt* input and a *req* output to communicate with the arbiter, and a set of shared (read-write) signals, the `IRDY` and the `FRAME`, which are used to communicate with target devices. The arbiter ensures that at most one master device can write to the shared signals. Figure 1(b) shows a graphical description of the interface representing a master device. The figure shows for each location, the assumption ("a"), the guarantee ("g"), the set of inout variables that the interface writes to, and guarded transitions between locations. Composing two such interfaces we obtain the interface shown in Figure 1(c). Location `Owner_Owner` is illegal because both components write the shared variables `FRAME` and `IRDY`. Input assumptions of locations `Req_Req`, `Owner_Req` and `Req_Owner` are strengthened to make the illegal location unreachable. Note that this propagates the PCI master's assumptions about its environment to an assumption on the behavior of the arbiter (which is the environment of the composite module): the arbiter should never assert `gnt1` (`gnt2`) during or after asserting `gnt2` (`gnt1`), until `req2` (`req1`) is de-asserted at least once. ∎



(a) (b) (c)

(d) (e)

Fig. 1. PCI and Token-ring Protocols 1(a) PCI Local Bus Structural Diagram 1(b) PCI Master Interface 1(c) Composite interface for two PCI Master Modules 1(d) Token Ring Network Configuration 1(e) Token-ring NT Interface

2.3 Properties of compatibility and composition

If M and N are composable Moore interfaces, we define their *product* $M \otimes N$ by $\mathcal{V}^o_{M\otimes N} = \mathcal{V}^o_M \cup \mathcal{V}^o_N$ and $\mathcal{V}^i_{M\otimes N} = (\mathcal{V}^i_M \cup \mathcal{V}^i_N) \setminus \mathcal{V}^o_{M\otimes N}$, and by letting $\theta^o_{M\otimes N} = \theta^o_M \wedge \theta^o_N$, $\theta^i_{M\otimes N} = \theta^i_M \wedge \theta^i_N$, $\tau^o_{M\otimes N} = \tau^o_M \wedge \tau^o_N$, and $\tau^i_{M\otimes N} = \tau^o_M \wedge \tau^i_N$. Intuitively, an *environment* for a Moore interface M is an interface that drives all free inputs of M, ensuring that all the input assumptions are met. Precisely, we say that a Moore interface N is an environment for a Moore interface M if M and N are composable and closed (i.e., $\mathcal{V}^o_M \cap \mathcal{V}^o_N = \emptyset$, and $\mathcal{V}^i_{M\otimes N} = \emptyset$), and if the following conditions hold:

- *Non-blocking:* $\theta^i_{M\otimes N}$ is satisfiable, and $\forall \mathcal{V}^o_{M\otimes N}.(\exists \mathcal{V}^i_{M\otimes N})'.\tau^i_{M\otimes N}$ holds.
- *Legal:* for all sequences $s_0, s_1, s_2, \ldots$ of states in $\mathcal{S}[\mathcal{V}_{M\otimes N}]$ with $s_0 \models \theta^o_{M\otimes N}$ and $(s_k, s_{k+1}) \models \tau^o_{M\otimes N}$ for all $k \geq 0$, we have also that $s_0 \models \theta^i_{M\otimes N}$ and $(s_k, s_{k+1}) \models \tau^i_{M\otimes N}$ for all $k \geq 0$.

Analogous definitions for product and environment can be given for bidirectional interfaces. The following theorem states the main properties of compatibility and composition of Moore interfaces; an analogous result holds for bidirectional interfaces.

Theorem 1 (properties of compatibility and composition) *The following assertions hold:*

1. *Given three Moore interfaces M, N, P, either $(M\|N)\|P$ and $(M\|N)\|P$ are both undefined (due to non-composability or incompatibility), or they are both defined, in which case they are equal.*
2. *Given two composable Moore interfaces M and N, we have that $M \wr\!\!\wr N$ iff there is an environment for $M \otimes N$.*
3. *Given two compatible Moore interfaces M and N, and P composable with $M\|N$, we have that $(M\|N) \wr\!\!\wr P$ iff there is an environment for $M \otimes N \otimes P$.*

The second assertion makes precise our statement that two interfaces are compatible iff there is some environment in which they can work correctly together. The third assertion states that composition does not unduly restrict the input assumptions: checking compatibility with the composition $M\|N$ amounts to checking compatibility with M and N.

3 Refinement

We define refinement as alternating simulation [5]: roughly, a component N refines M (written $N \preceq M$) if N can simulate all inputs of M, and if M can simulate all outputs of N. Encoding the relation between the states of two Moore interfaces M and N by a predicate R, we can state the definition of refinement as follows.

Definition 5 (Refinement of Moore interfaces) Given two Moore interfaces M and N, we have that $N \preceq M$ if $\mathcal{V}^i_N \subseteq \mathcal{V}^i_M$ and $\mathcal{V}^i_M \cap (\mathcal{V}^o_M \cup \mathcal{V}^o_N) = \emptyset$, and if there is a predicate R on $\mathcal{V}_M \cup \mathcal{V}_N$ such that the following formulas are valid:

$$\begin{aligned} \theta^i_M \wedge \theta^o_N &\rightarrow \exists(\mathcal{V}^o_M \setminus \mathcal{V}^o_N).(\theta^i_N \wedge \theta^o_M \wedge R) \\ R \wedge \tau^i_M \wedge \tau^o_N &\rightarrow \exists(\mathcal{V}^o_M \setminus \mathcal{V}^o_N)'.(\tau^i_N \wedge \tau^o_M \wedge R') \end{aligned} \qquad \blacksquare$$

As for normal simulation, there is a unique largest refinement relation between any two Moore interfaces. Hence, Definition 5 provides an iterative algorithm for deciding refinement: let $R_0 = \mathrm{T}$, and for $k \geq 0$, let

$$R_{k+1} = R_k \wedge \forall(\mathcal{V}_M \cup \mathcal{V}_N)'.\big(\tau^i_M \wedge \tau^o_N \rightarrow \exists(\mathcal{V}^o_M \setminus \mathcal{V}^o_N)'.(\tau^i_N \wedge \tau^o_M \wedge R'_k)\big). \quad (2)$$

Denoting with $R_* = \lim_{k\to\infty} R_k$ the fixpoint (that again can be computed in a finite number of iterations), we have that $N \preceq M$ if and only if (i) $\mathcal{V}^i_N \subseteq \mathcal{V}^i_M$

and $\mathcal{V}^i_M \cap (\mathcal{V}^o_M \cup \mathcal{V}^o_N) = \emptyset$, and (ii) $\theta^i_M \wedge \theta^o_N \rightarrow \exists(\mathcal{V}^o_M \setminus \mathcal{V}^o_N).(\theta^i_N \wedge \theta^o_M \wedge R)$. In order to obtain an efficient implementation, we can again take advantage for the computation of (2) of list representations for the transition relations, and apply image-computation techniques.

Refinement of bidirectional interfaces is defined similarly, except that the refinement relation relates the locations of the two interfaces, rather than the states. The definition is as follows.

Definition 6 (Refinement of bidirectional interfaces) Given two bidirectional interfaces M and N, N refines M ($N \preceq M$) iff there is a binary relation $\preceq \subseteq Q_N \times Q_M$ such that $\hat{q}_N \preceq \hat{q}_M$, and such that for all $q \preceq p$ we have (i) $v^i_N(q) \subseteq v^i_M(q)$, (ii) $v^o_N(q) \supseteq v^o_M(p)$, (iii) $\phi^i_M(p) \rightarrow \phi^i_N(q)$, (iv) $\phi^o_N(q) \rightarrow \phi^o_M(p)$, (v) for all $s \in \mathcal{S}[v^i_M(p)]$ and all $t \in \mathcal{S}[v^o_N(q)]$, if $s \models \rho_M(p, p')$ and $t \models \rho_N(q, q')$, then $q' \preceq p'$. ∎

We can check whether $N \preceq M$ by adapting the classical iterative refinement check [16]. We start with the total relation $\preceq_0 = Q_N \times Q_M$, and for $k \geq 0$, we let $\preceq_{k+1}$ be the subset of $\preceq_k$ such that conditions (i)–(v) hold, with $\preceq_k$ in place of $\preceq$ in condition (v). Once we reach $m \geq 0$ such that $\preceq_{m+1} = \preceq_m$, we have that $N \preceq M$ iff $\hat{q}_N \preceq \hat{p}_N$. Since bidirectional interfaces are deterministic we can reduce the refinement checking problem to graph reachability on the product interface and hence $N \preceq M$ can be decided in $O(|Q_N| \times |Q_M|)$ time.

Example 3 (Token Ring) The IEEE 802.5 (Token Ring) is a widely used deterministic LAN protocol. Figure 1(e) shows an interface modeling a node that initially does not have the token. The same diagram with T as initial state would represent a node that initially has the token. We call these two interfaces NT and T, respectively. The token ring components are connected in a cyclic network; each pair of adjacent nodes communicate by *req* and *gnt* signals (Figure 1(d)). The *req* signal flows clockwise, and is used to request the token; the signal *give* flows counterclockwise, and is used to grant the token. The protocol fails if more than one node has the token simultaneously: indeed, we can verify that two T interfaces are not compatible, while an NT interface is compatible with a T interface. Moreover, the protocol works for any number of participating nodes. To verify this, we check two refinements: first, an open-ring configuration consisting entirely of NT nodes is a refinement of the configuration consisting in just one NT node; second, an open-ring configuration with any number of NT nodes and one T node is a refinement of a configuration consisting in a single T node. Our implementation is able to perform the above compatibility and refinement checks in a fraction of a second. ∎

The notion of refinement, in addition to implementation, captures also substitutivity: if N refines M, and M is compatible with the remainder P of the design, then P is also compatible with N.

Theorem 2 (Substitutivity of refinement) *Consider three bidirectional Moore or bidirectional interfaces M, N, P, such that $M \between P$, and $N \preceq M$. If $(\mathcal{V}^o_N \cap \mathcal{V}^i_P) \subseteq (\mathcal{V}^o_M \cap \mathcal{V}^i_P)$, then $N \between P$ and $(N \| P) \preceq (M \| P)$.*

The result has a proviso: all the variables that are output by N and input by P should also be output by M. If this were not the case, it would be possible for the additional outputs of N to violate the input assumptions of P.

References

1. M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, volume 372 of *Lect. Notes in Comp. Sci.*, pages 1–17. Springer-Verlag, 1989.
2. S. Abramsky. Games in the semantics of programming languages. In *Proc. of the 11th Amsterdam Colloquium*, pages 1–6. ILLC, Dept. of Philosophy, University of Amsterdam, 1997.
3. S. Abramsky, S. Gay, and R. Nagarajan. A type-theoretic approach to deadlock-freedom of asynchronous systems. In *TACS'97: Theoretical Aspects of Computer Software. Third International Symposium*, 1997.
4. R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, pages 7–48, 1999.
5. R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *CONCUR 97: Concurrency Theory*, volume 1466 of *Lect. Notes in Comp. Sci.*, pages 163–178. Springer-Verlag, 1998.
6. R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: modularity in model checking. In *CAV 98: Proc. of 10th Conf. on Computer Aided Verification*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 521–525. Springer-Verlag, 1998.
7. E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *CAV 96: Proc. of 8th Conf. on Computer Aided Verification*, volume 1102 of *Lect. Notes in Comp. Sci.*, pages 419–422. Springer-Verlag, 1996.
8. L. de Alfaro, R. Alur, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Meyer-Kirsch, and B.Y. Wang. Mocha: A model checking tool that exploits design structure. In *ICSE 01: Proceedings of the 23rd International Conference on Software Engineering*, 2001.
9. L. de Alfaro and T.A. Henzinger. Interface automata. In *Proc. of 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM Press, 2001.
10. L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *EMSOFT 01: Proc. of First Int. Workshop on Embedded Software*, volume 2211 of *Lect. Notes in Comp. Sci.*, pages 148–165. Springer-Verlag, 2001.
11. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1988.
12. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
13. N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.
14. N.A. Lynch and M. Tuttle. Hierarcical correctness proofs for distributed algorithms. In *Proc. of 6th ACM Symp. Princ. of Dist. Comp.*, pages 137–151, 1987.
15. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
16. R. Milner. An algebraic definition of simulation between programs. In *Proc. of Second Int. Joint Conf. on Artificial Intelligence*, pages 481–489. The British Computer Society, 1971.
17. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
18. J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.