To appear in Formal Aspects of Computing, 2007

Interface Synthesis and Protocol Conversion

Purandar Bhaduri[1] and S. Ramesh[2]

[1]Department of Computer Science and Engineering
Indian Institute of Technology Guwahati, India
[2]India Science Lab, GM R&D Centre, Bangalore, India

Abstract. Given deterministic interfaces P and Q, we investigate the problem of synthesising an interface R such that P composed with R refines Q. We show that a solution exists iff P and $Q^{\perp}$ are compatible, and the most general solution is given by $(P \parallel Q^{\perp})^{\perp}$, where $P^{\perp}$ is the interface P with inputs and outputs interchanged. Remarkably, the result holds both for asynchronous and synchronous interfaces. We model interfaces using the interface automata formalism of de Alfaro and Henzinger. For the synchronous case, we give a new definition of synchronous interface automata based on Mealy machines and show that the result holds for a weak form of nondeterminism, called observable nondeterminism. We also characterise solutions to the synthesis problem in terms of winning input strategies in the automaton $(P \otimes Q^{\perp})^{\perp}$, and the most general solution in terms of the most permissive winning strategy. We apply the solution to the synthesis of converters for mismatched protocols in both the asynchronous and synchronous domains. For the asynchronous case, this leads to automatic synthesis of converters for incompatible network protocols. In the synchronous case, we obtain automatic converters for mismatched intellectual property blocks in system-on-chip designs. The work reported here is based on earlier work on interface synthesis in [Bha05] for the asynchronous case, and [BR06] for the synchronous one.

Keywords: Interface automata; component compatibility; interface synthesis; protocol conversion.

1. Introduction

Interfaces play a central role in component based design and verification of systems. In this paper we study the problem of synthesising an interface R, which composed with a known interface P is a refinement of an interface Q. This is a central problem in component based top-down design of a system. The interface Q is an abstract interface, a high level specification of the component under development. The interface P is a known part of the implementation and we are required to find the most general (i.e., abstract) solution R satisfying the relation $P \parallel R \preceq Q$, when it exists. Here $P \parallel Q$ is the composition of P and Q, and $P \preceq Q$ denotes 'P is a refinement of Q'. This problem has wide ranging applications such as logic synthesis, design of discrete controllers, communication gateway design and protocol conversion, and has been studied previously in different contexts – see [MvB83, Shi89, Par89, LX90, HU99, YVB$^+$01, YVB$^+$02]. We study two versions of the problem, one for the asynchronous and the other for the

Correspondence and offprint requests to: Purandar Bhaduri, Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, Guwahati 781039, India. Tel: +91 316 2582360. Fax: +91 361 269 2787. e-mail: pbhaduri@iitg.ernet.in

synchronous case, in the setting of interface automata [dAH01], where composition and refinement of interfaces are respectively the composition of interface automata and alternating refinement relations[AHKV98].

Interface automata are a formalism for reasoning about composition and refinement of component interfaces in terms of the protocol aspects of component behaviour. They are like ordinary automata, except for the distinction between input and output actions. The input actions of an interface automaton P are controlled by its environment. Therefore an input action labelling a transition is an *input assumption* (or constraint on P's environment). Dually, an output action of P is under P's control, and represents an an *output guarantee* of P. Unlike I/O automata [LT87], interface automata are *not* required to be input enabled. If an input action a is not enabled at a state s, it is an assumption on the automaton's environment that it will not provide a as an input in state s.

When two interfaces P and Q are composed, the combined interface may contain incompatible states: states where one interface can generate an output that is not a legal input for the other. In the combined interface it is the environment's responsibility to ensure that such a state is unreachable [dAH01]. This can be formalised as a two person game [dAH01] which has the same flavour as the controller synthesis problem of Ramadge and Wonham [RW89]; in our setting the role of the controller is played by the environment. More formally, we follow de Alfaro [dA03] in modelling an interface as a game between two players, Output and Input. Player Output represents the system and its moves represent the outputs generated by the system. Player Input represents the environment; its moves represent the inputs the system receives from its environment. In general, the set of available moves of each player depends on the current state of the combined system. The interface is well-formed if the Input player has a winning strategy in the game, where the winning condition is to avoid all incompatible states. Clearly, the game aspect is relevant only when defining the composition of two interfaces.

Refinement of interfaces corresponds to weakening assumptions and strengthening guarantees. An interface P refines Q only if P can be used in any environment where Q can be. The usual notion of refinement is *simulation* or *trace containment* [LT87]. For interface automata, a more appropriate notion is that of *alternating simulation* [AHKV98], which is contravariant on inputs and covariant on outputs: if $P \preceq Q$ (P *refines* Q), P accepts more inputs (weaker input assumptions) and provides fewer outputs (stronger output guarantees). Thus alternating refinement preserves compatibility: if P and Q are compatible (i.e., $P \parallel Q$ is well-formed) and $P' \preceq P$, then so are P' and Q. The basic notions of interface automata are summarised in Section 2.

In Section 3 we show that a solution to $P \parallel R \preceq Q$ for R exists for deterministic interface automata iff P and $Q^{\perp}$ are compatible, and the most abstract (under alternating refinement) solution is given by $(P \parallel Q^{\perp})^{\perp}$. Further, in Section 4 we show that such an R can be constructed from the most permissive winning strategy for player Input in the combined game $(P \otimes Q^{\perp})^{\perp}$. Here $P^{\perp}$ is the game P with the moves of the players Input and and Output interchanged, and $P \otimes Q$ is the combined game obtained from P and Q by synchronising on shared actions and interleaving the rest. We say a strategy π is more permissive than π' when, at every position in the game, the set of moves allowed by π includes those allowed by π'. The most permissive winning strategy is one that is least restrictive. This result ties up the relation between composition, refinement, synthesis and winning strategies, and should be seen as one more step towards a "uniform framework for the study of control, verification, component-based design, and implementation of open systems", based on games [dA03].

Note that the notation $P^{\perp}$ is borrowed from linear logic [Gir87], where games play an important semantic role [Bla92]. Using the notation of linear logic, the solution R to the synthesis problem can be written as $(P \otimes Q^{\perp})^{\perp} = P^{\perp} \invamp Q = P \multimap Q$, where $\otimes$, $\invamp$ and $\multimap$ are respectively, the linear logic connectives 'With', 'Par' and linear implication. In our setting, the $\otimes$ connective of linear logic is parallel composition $\parallel$. The striking similarity of this solution with submodule construction by von Bochmann in [vB02] and the language equation posed by Yevtushenko *et al.*in [YVB$^+$01, YVB$^+$02] is intriguing. In these works, the largest solution of the language equation $P \bullet R \subseteq Q$ for R is the language $\overline{P \bullet \overline{Q}}$ where $P \bullet Q$ is the synchronous (or parallel) composition of languages P and Q, and $\overline{P}$ is the complement of P. Clearly, there is a formal correspondence between $P \bullet Q$ and our $P \parallel Q$, between $\overline{P}$ and our $P^{\perp}$, and between language inclusion and alternating simulation.

We should also mention the formal resemblance of our work with Abramsky's Semantics of Interaction [Abr97], based on the game semantics of linear logic. In particular, the strategy called *Application* (or *Modus Ponens*) in [Abr97] is the solution to our synthesis problem in a different setting. The solution $R = P \multimap Q$ suggests that the problem of synthesis can be seen as the construction of a suitable morphism in an appropriate category of interface automata, along the lines of [MT98, Tab04]. However, we do not pursue this thread in this paper.

There is also a similarity between our work and the solution to the *rectification problem* given in [BDWM93], using Dill's trace theory [Dil89]. The results in [BDWM93] are applicable to combinational circuits i.e., interfaces without any state. In [BDWM93] P, Q, R are combinational circuits modelled as input-output relations and $||$ the circuit composition operator; *mir* is the *mirror* function that swaps inputs and outputs and complements the input-output relation; $\preceq$ is the *conformance relation* which captures the implementation relation between circuits and specifications. The rectification problem solved in [BDWM93] is the following: when can a small subnetwork of a combinational circuit be replaced by another less expensive one? The answer to the rectification problem is provided by a general theorem that states that if Q is a specification of the entire circuit then a replacement subnetwork P combined with the rest of the circuit R will satisfy the specification Q (i.e., $P \parallel R \subseteq Q$) iff $R \subseteq \text{mir}(P \parallel \text{mir}(Q))$ (modulo a projection operator used to ignore the internal signals of a composed circuit). Clearly, this has the same form as our solution to the synthesis problem.

As a practical application, in the asynchronous case, in Section 5 we show how to apply interface synthesis to the protocol conversion problem for mismatched network protocols. The heterogeneity of existing networks often results in incompatible protocols trying to communicate with each other. The protocol conversion problem is, given two network protocols P_1 and P_2 which are mismatched, to come up with a converter C which mediates between the two protocols, such that the combined system conforms to an overall specification S. We show that a converter C, if it exists, can be obtained as the solution to $P \parallel C \preceq S$, where $P = P_1 \parallel P_2$ is the composition of the two protocols.

For the synchronous case, in Section 6 we present synchronous interface automata for reasoning about composition and refinement of synchronous hardware components, such as intellectual property (IP) blocks in system-on-chip (SoC) designs. In Section 7 we pose and solve the same synthesis problem as in the asynchronous case. What is remarkable is, the solutions for the asynchronous and synchronous versions have the same form, namely the most general solution to $P \parallel R \preceq Q$ exists iff P and $Q^{\perp}$ are compatible, and is given by $R = (P \parallel Q^{\perp})^{\perp}$. This points to a unified theory of interface synthesis underlying both the asynchronous and synchronous cases. The work on *agent algebras* by Passerone*et al.* [BPSV03, Pas04] seems to bear this out. In Section 8 we show how to apply the synthesis method to automatically synthesise protocol converters for mismatched IP blocks.

To summarise the main contributions of this paper, we have an algebraic characterisation of solutions to the problem of synthesis of interface automata, where the operators are parallel composition $P \parallel Q$ and 'mirror' $P^{\perp}$. The synthesis algorithm relies on the computation of winning strategies in games implicit in the definition of parallel composition for interface automata. The algorithm is the standard iterative fixed point computation using controllable predecessors for solving safety games (see [Tho95]). The usefulness of the solution to the synthesis problem is demonstrated both in the asynchronous and synchronous cases by providing automated methods of constructing protocol converters. This provides a unified theory and an algorithm for solving protocol conversion problems which have appeared in the literature in many guises. The work reported here is based on earlier work on interface synthesis in [Bha05] for the asynchronous case, and [BR06] for the synchronous one.

Related Work Merlin and von Bochmann [MvB83] proposed the *submodule construction problem* (also called equation solving or factorisation) for communication protocol specification and synthesis. Given a module specification S and a submodule specification M_1 in terms of labelled transition systems, they proposed a method to construct a submodule M_2 such that M_1 and M_2 together realise the specification S by synchronous interaction. Haghverdi and Ural [HU99] gave a more formal presentation of the problem and an algorithm for its solution using prefix-closed finite state machines as descriptions of submodules. One limitation of this solution is that the notion of correct realisation is trace equivalence which is well known not to preserve certain behavioural properties such as the presence of deadlock. This problem was addressed in the context of equation solving for CCS processes using observational equivalence or strong bisimulation (see [Mil89]) by various researchers. A representative sample can be found in Shields [Shi89], Parrow [Par89] and Larsen and Xinxin [LX90]. Yevtushenko *et al.* [YVB$^+$01, YVB$^+$02] study the related problem of language equation solving for both synchronous and parallel (interleaving) composition in the context of synchronous and asynchronous circuits. The main difference between the works cited above and ours is that we consider models of *open systems* with an asymmetry between inputs and outputs. This naturally leads to the game formulation and alternating simulation as the behavioural preorder captured by the interface automata formalism.

The controller synthesis problem and its solution as a winning strategy in a game has a long history, going back to Büchi and Landwebers' solution of Church's problem [BL69]. More recent applications of the idea in the synthesis of open systems occur in [PR89, MPS95, MT98]. The control of discrete event systems was introduced by Ramdage and Wonham [RW89] using a language theoretic approach, where the 'controllable' and 'uncontrollable' actions clearly correspond to the moves of the player and the opponent in a corresponding game. The use of games for the synthesis of converters for mismatched protocols by Passerone *et al.* [PdAHSV02] can be seen as applications of the same general principle. The present paper uses games to solve the interface synthesis problem, but the solution has a closed form reminiscent of the submodule construction problem referred to above and the rectification problem of Dill [BDWM93]. The combinatorial details of the game structure is hidden behind the algebraic form of the solution.

Recent work on *agent algebras* [BPSV03, Pas04] formalises the notions of composition and conformance in an abstract algebraic framework, and makes use of the mirror function in an essential way. The work provides sufficient conditions for characterising all controllers that satisfy a specification when composed with a plant. One possible future work is to investigate the relationship between agent algebras and our interface synthesis framework.

2. Asynchronous Interfaces: Interface Automata

In this section we define interface automata and their composition and refinement. We follow the game formulation presented in [dA03]. Throughout this work we consider only *deterministic* interface automata.

Definition 2.1. An *interface automaton* P is a tuple $(S_P, S_P^0, \mathcal{A}_P^I, \mathcal{A}_P^O, \Gamma_P^I, \Gamma_P^O, \delta_P)$ where:

- S_P is a finite set of *states*.
- $S_P^0 \subseteq S_P$ is the set of *initial states*, which has at most one element, denoted s_P^0.
- $\mathcal{A}_P^I$ and $\mathcal{A}_P^O$ are disjoint sets of *input* and *output actions*. The set $\mathcal{A}_P = \mathcal{A}_P^I \cup \mathcal{A}_P^O$ is the set of all *actions*.
- $\Gamma_P^I : S_P \rightarrow 2^{\mathcal{A}_P^I}$ is a map assigning to each state $s \in S_P$ a set (possibly empty) of *input moves*. Similarly, $\Gamma_P^O : S_P \rightarrow 2^{\mathcal{A}_P^O}$ assigns to each state $s \in S_P$ a set (again, possibly empty) of *output moves*. The input and output moves at a state s correspond to actions that can be accepted and generated at s respectively. Denote by $\Gamma_P(s) = \Gamma_P^I(s) \cup \Gamma_P^O(s)$ the set of all actions at s.
- $\delta_P : S_P \times \mathcal{A}_P \rightarrow S_P$ is a *transition function* associating a target state $\delta_P(s, a)$ with each state $s \in S_P$ and action $a \in \mathcal{A}_P$. Note that the value $\delta_P(s, a)$ makes sense only when $a \in \Gamma_P(s)$. When $a \notin \Gamma_P(s)$, the value can be arbitrary.

The interface automaton P is said to be *empty* when its set of initial states S_P^0 is empty. Empty interface automata arise when incompatible automata are composed.

Definition 2.2. An *input strategy* for P is a map $\pi^I : S_P^+ \rightarrow 2^{\mathcal{A}_P^I}$ satisfying $\pi^I(\sigma s) \subseteq \Gamma_P^I(s)$ for all $s \in S_P$ and $\sigma \in S_P^*$. An *output strategy* $\pi^O : S_P^+ \rightarrow 2^{\mathcal{A}_P^O}$ is defined similarly. The set of input and output strategies of P are denoted by Π_P^I and Π_P^O respectively.

Note In keeping with [dA03], we have given the general definition of strategy, where the moves can depend on the history of the game. It turns out that for the case of safety games we consider in this paper, it is enough to consider only *memoryless* strategies, where the current state suffices to define the set of moves. This simplification will be made in Section 6 while discussing the synchronous interface automata.

An input and output strategy jointly determine a set of traces in S_P^+ as follows. At each step, if the input strategy proposes a set $\mathcal{B}^I$ of actions, and the output strategy proposes a set $\mathcal{B}^O$ of actions, an action from $\mathcal{B}^I \cup \mathcal{B}^O$ is chosen nondeterministically.

Definition 2.3. Given a state $s \in S_P$, and input strategy π^I and an output strategy π^O, the set Outcomes$(s, \pi^I, \pi^O) \subseteq S_P^+$ of resulting plays is defined inductively as follows:

- $s \in \text{Outcomes}_P(s, \pi^I, \pi^O)$;
- if $\sigma t \in \text{Outcomes}(s, \pi^I, \pi^O)$ for $\sigma \in S_P^+$ and $t \in S_P$, then for all $a \in \pi^I(\sigma t) \cup \pi^O(\sigma t)$ the sequence $\sigma t \delta_P(s, a) \in \text{Outcomes}_P(s, \pi^I, \pi^O)$.

A state $s \in S_P$ is said to be *reachable* in P, if there is a sequence of states $s_0, s_1, \ldots, s_n$ with $s_0 \in S_P^0$, $s_n = s$, and for all $0 \leq k < n$ there is $a_k \in \Gamma_P(s_k)$ such that $\delta_P(s_k, a_k) = s_{k+1}$. Reach$(P)$ denotes the set of reachable states of P

The refinement of interface automata is known as *alternating simulation*, the right notion of simulation between games [AHKV98]. Intuitively, an alternating simulation $\rho \subseteq S_P \times S_Q$ from P to Q is a relation for which $(s, t) \in \rho$ implies all input moves from t can be simulated by s and all output moves from s can be simulated by t.

Definition 2.4. An *alternating simulation* ρ from P to Q is a relation $\rho \subseteq S_P \times S_Q$ such that, for all $(s, t) \in \rho$ and all $a \in \Gamma_Q^I(t) \cup \Gamma_P^O(s)$, the following conditions are satisfied:

1. $\Gamma_Q^I(t) \subseteq \Gamma_P^I(s)$;
2. $\Gamma_P^O(s) \subseteq \Gamma_Q^O(t)$;

3. $(\delta_P(s,a), \delta_Q(t,a)) \in \rho$.

Refinement between interface automata is defined as the existence of an alternating simulation between the initial states.

Definition 2.5. An interface automaton P *refines* an interface automaton Q, written $P \preceq Q$, if the following conditions are satisfied:

1. $\mathcal{A}^I_Q \subseteq \mathcal{A}^I_P$;
2. $\mathcal{A}^O_P \subseteq \mathcal{A}^O_Q$;
3. there is an alternating simulation ρ from P to Q, such that $(s^0, t^0) \in \rho$ for some $s^0 \in S^0_P$ and $t^0 \in S^0_Q$.

We now define the parallel composition $P \parallel Q$ of interface automata P and Q in a series of steps.

Definition 2.6. P and Q are *composable* if $\mathcal{A}^O_P \cap \mathcal{A}^O_Q = \emptyset$.

We first define the *product automaton* $P \otimes Q$ of two composable interface automata P and Q, by synchronising their shared actions and interleaving all others. The set of shared actions of P and Q is defined by $\text{Shared}(P,Q) = \mathcal{A}_P \cap \mathcal{A}_Q$.

Definition 2.7. The *product* $P \otimes Q$ of two composable interface automata P and Q is defined by

- $S_{P\otimes Q} = S_P \times S_Q$;
- $S^0_{P\otimes Q} = S^0_P \times S^0_Q$;
- $\mathcal{A}^I_{P\otimes Q} = (\mathcal{A}^I_P \cup \mathcal{A}^I_Q) \backslash \text{Comm}(P,Q)$ where $\text{Comm}(P,Q) = (\mathcal{A}^O_P \cap \mathcal{A}^I_Q) \cup (\mathcal{A}^I_P \cap \mathcal{A}^O_Q)$ is the set of *communication actions*, a subset of $\text{Shared}(P,Q)$;
- $\mathcal{A}^O_{P\otimes Q} = \mathcal{A}^O_P \cup \mathcal{A}^O_Q$;
- $\Gamma^I_{P\otimes Q}((s,t)) = (\Gamma^I_P(s) \backslash (\mathcal{A}^O_Q \cup \mathcal{A}^I_Q)) \cup (\Gamma^I_Q(t) \backslash (\mathcal{A}^O_P \cup \mathcal{A}^I_P)) \cup (\Gamma^I_P(s) \cap \Gamma^I_Q(t))$ for all $(s,t) \in S_P \times S_Q$;
- $\Gamma^O_{P\otimes Q}((s,t)) = \Gamma^O_P(s) \cup \Gamma^O_Q(t)$, for all $(s,t) \in S_P \times S_Q$;
- for all $a \in \mathcal{A}_{P\otimes Q}$,

$$\delta_{P\otimes Q}((s,t),a) = \begin{cases} (\delta_P(s,a), \delta_Q(t,a)) & \text{if } a \in \mathcal{A}_P \cap \mathcal{A}_Q \\ (\delta_P(s,a), t) & \text{if } a \in \mathcal{A}_P \backslash \mathcal{A}_Q \\ (s, \delta_Q(t,a)) & \text{if } a \in \mathcal{A}_Q \backslash \mathcal{A}_P \end{cases}$$

Note There is an asymmetry between input and output actions in the above definition. An input and output action with the same label combine to form an *output* action. The intent is to model multi-way broadcast communication, as in the version of interface automata defined in [dA03] and in I/O automata [LT87]. This is in contrast to the original version defined in [dAH01], where an input and an output action with the same name combine to give an *internal* action.

Since interface automata need not be input enabled, there may be reachable states in $P \otimes Q$ where a communication action can be output by one of the automaton but cannot be accepted as input by the other. These states are called *locally incompatible*.

Definition 2.8. The set $\text{Incomp}(P,Q)$ of *locally incompatible states* of P and Q consists of all pairs $(s,t) \in S_P \times S_Q$ for which one of the following two conditions hold:

1. $\exists a \in \text{Comm}(P,Q)$ such that $a \in \Gamma^O_P(s)$ but $a \notin \Gamma^I_Q(t)$,
2. $\exists a \in \text{Comm}(P,Q)$ such that $a \in \Gamma^O_Q(t)$ but $a \notin \Gamma^I_P(s)$.

A local incompatibility can be avoided if there is a helpful environment, which by providing the right sequence of inputs can steer the automaton away from such an undesirable state. The states from which Input can prevent the product $P \otimes Q$ from reaching a state in $\text{Incomp}(P,Q)$ are called *compatible*. In other words, the compatible states are those from which Input has a winning strategy.

Definition 2.9. A state $s \in S_{P\otimes Q}$ is *compatible* if there is an input strategy $\pi^I \in \Pi^I_{P\otimes Q}$ such that, for all output strategies $\pi^O \in \Pi^O_{P\otimes Q}$, all $\sigma \in \text{Outcomes}_{P\otimes Q}(s, \pi^I, \pi^O)$ and all incompatible states $w \in \text{Incomp}(P,Q)$, the state w does not appear in the sequence σ.

The composition $P \parallel Q$ is obtained by restricting $P \otimes Q$ to the states that can be reached from the initial state under an input strategy that avoids all locally incompatible states. We call these states *backward compatible*. These are the states that are reachable from the initial state of $P \otimes Q$ by visiting only compatible states. Note that in [dA03] backward compatible states are called *usably reachable states*.

Definition 2.10. A state $s \in S_{P\otimes Q}$ is *backward compatible* in $P \otimes Q$ if there is an input strategy $\pi^I \in \Pi^I_{P\otimes Q}$ such that:

- for all initial states $s_0 \in S^0_{P\otimes Q}$, all output strategies $\pi^O \in \Pi^O_{P\otimes Q}$, all outcomes $\sigma \in \text{Outcomes}_{P\otimes Q}(s_0, \pi^I, \pi^O)$ and all $w \in \text{Incomp}(P,Q)$, w does not occur in σ;
- there is an initial state $s_0 \in S^0_{P\otimes Q}$, an output strategy $\pi^O \in \Pi^O_{P\otimes Q}$, and an outcome $\sigma \in \text{Outcomes}_{P\otimes Q}(s_0, \pi^I, \pi^O)$ such that $s \in \sigma$.

Definition 2.11. The *composition* $P \parallel Q$ of two interface automata P and Q, with T the set of backward compatible states of the product $P \otimes Q$, is an interface automaton defined by:

- $S_{P\|Q} = T$
- $S^0_{P\|Q} = S^0_{P\otimes Q} \cap T$
- $\mathcal{A}^I_{P\|Q} = \mathcal{A}^I_{P\otimes Q}$
- $\mathcal{A}^O_{P\|Q} = \mathcal{A}^O_{P\otimes Q}$
- $\Gamma^I_{P\|Q}(s) = \{a \in \Gamma^I_{P\otimes Q}(s) \mid \delta_{P\otimes Q}(s,a) \in T\}$ for all $s \in T$
- $\Gamma^O_{P\|Q}(s) = \Gamma^O_{P\otimes Q}(s)$ for all $s \in T$
- for all $s \in T$, $a \in \Gamma_{P\|Q}(s)$,

$$\delta_{P\|Q}(s,a) = \begin{cases} \delta_{P\otimes Q}(s,a) & \text{if } \delta_{P\otimes Q}(s,a) \in T \\ \text{arbitrary} & \text{otherwise} \end{cases}$$

Definition 2.12. P and Q are said to be *compatible* if their composition is non-empty i.e., $S^0_{P\|Q} \neq \emptyset$. This is equivalent to $s^0_{P\otimes Q} \in T$, where T is the set of backward compatible states of $P \otimes Q$.

Notation We write $\text{Reach}^O(P)$ to denote the set of states of P that are reachable from the initial state s^0_P by following only output actions.

We use the following lemma in our proof of Theorems 3.3 and 3.4 in Section 3. Since the best input strategy to avoid locally incompatible states is simply to generate no inputs to $P \otimes Q$ at any state, the set of compatible states in $P \otimes Q$ is simply the set of states from which $P \otimes Q$ cannot reach a state in $\text{Incomp}(P,Q)$ by a sequence of output actions.

Lemma 2.13. P and Q are compatible iff the states in $\text{Reach}^O(P \otimes Q)$ are locally compatible, i.e., $\text{Reach}^O(P \otimes Q) \cap \text{Incomp}(P,Q) = \emptyset$.

Proof. Suppose P and Q are compatible. Then $s^0_{P\otimes Q}$ is a backward compatible state in $P \otimes Q$. This implies there is an input strategy π^I for $P \otimes Q$ which avoids all locally incompatible states starting from $s^0_{P\otimes Q}$, no matter what the output strategy is. Now Output can always force $P \otimes Q$ to enter any state in $\text{Reach}^O(P \otimes Q)$. In other words, an output strategy π^O exists for which every state s in $\text{Reach}^O(P \otimes Q)$ appears in some sequence in $\text{Outcomes}_{P\otimes Q}(s^0_{P\otimes Q}, \pi^I, \pi^O)$. Since $s^0_{P\otimes Q}$ is a backward compatible in $P \otimes Q$, it follows that $\text{Reach}^O(P \otimes Q) \cap \text{Incomp}(P,Q) = \emptyset$. Conversely, suppose the states in $\text{Reach}^O(P \otimes Q)$ are locally compatible. This implies that any state in $\text{Incomp}(P,Q)$ can be reached, if at all, by following a sequence of actions which includes at least one input action. Then the input strategy which disables all such input actions avoids all locally incompatible states and so $s^0_{P\otimes Q}$ is backward compatible.
□

3. Synthesis of Interface Automata

The synthesis problem for interface automata is as follows. Given interface automata P and Q, we want to find the most general solution R to $P \parallel R \preceq Q$ when it exists, and characterise the conditions under which it exists. By a most

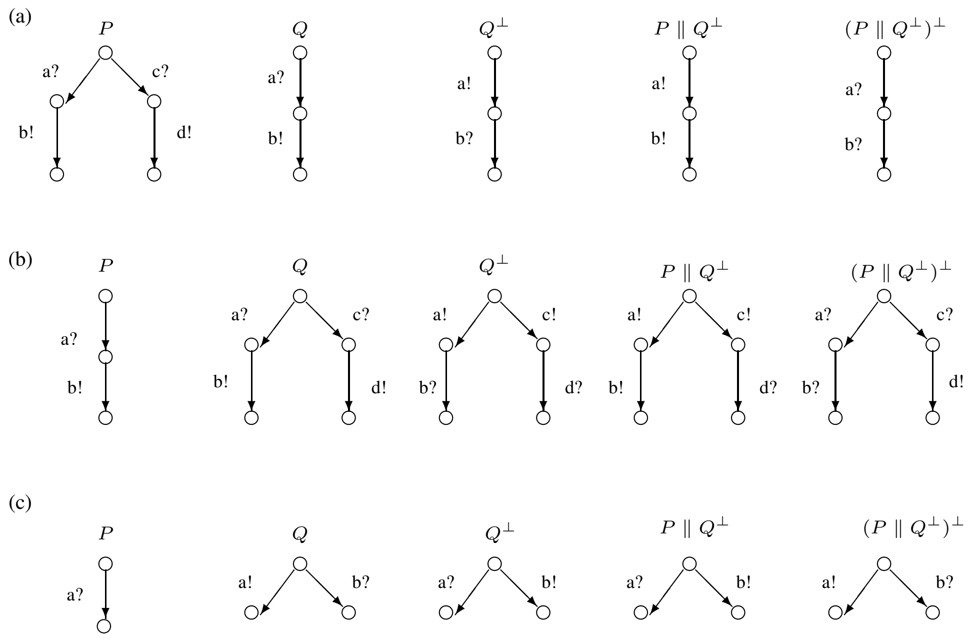


Fig. 1. Interface Automata Synthesis Examples

general solution we mean, a solution U, such that for any solution V, it is the case that $V \preceq U$. In this section we prove our main result for asynchronous interfaces, viz., the most general solution to $P \parallel R \preceq Q$ exists iff P and $Q^\perp$ are compatible and is given by $R = (P \parallel Q^\perp)^\perp$. Here $P^\perp$ is the same as P, except all the input actions in P become output actions in $P^\perp$ and similarly the output actions of P are the input actions of $P^\perp$.

Example 3.1. Fig. 1 presents three examples to illustrate the synthesis idea with given interface automata P and Q. The construction of $Q^\perp$, $P \parallel Q^\perp$ and $R = (P \parallel Q^\perp)^\perp$ are shown in each case.

1. In Fig. 1(a), the input actions are $\mathcal{A}_P^I = \mathcal{A}_Q^I = \{a, c\}$, and the output actions are $\mathcal{A}_P^O = \mathcal{A}_Q^O = \{b, d\}$. Note that in $P \parallel Q^\perp$, the transition labelled $c?$ does not appear, as it is a shared action, and has to be present in both P and $Q^\perp$ to appear in their product. Note also, how b appears as an input action in the result $(P \parallel Q^\perp)^\perp$.
2. In Fig. 1(b), the action sets are $\mathcal{A}_P^I = \{a\}$, $\mathcal{A}_P^O = \{b\}$, $\mathcal{A}_Q^I = \{a, c\}$ and $\mathcal{A}_Q^O = \{b, d\}$. In this case, the solution is essentially identical with Q, except for the polarity of action b. Note that there is already an alternating simulation between P and Q. The input transition labelled $b?$ appears in R because we assume $\mathcal{A}_P^O \subseteq \mathcal{A}_R^I$: in some sense, R can be thought of as a controller for P, and hence should be allowed to use all the output actions of P as input, in addition to driving the input actions of P. Note that if we changed the the input action set of P to be $\mathcal{A}_P^I = \mathcal{A}_Q^I = \{a, c\}$, then there would be no solution R, because P and $Q^\perp$ would not be compatible: in the initial state, $Q^\perp$ is ready to output a c, but P is not ready to accept it as input, even though c is a communication action between the two.
3. In Fig. 1(c), the action sets are $\mathcal{A}_P^I = \{a\}$, $\mathcal{A}_P^O = \emptyset$, $\mathcal{A}_Q^I = \{b\}$ and $\mathcal{A}_Q^O = \{a\}$. In this example, an input of P appears as an output of Q. The result $(P \parallel Q^\perp)^\perp$ adds the input b and also converts a from an input to an output. In this case, R is identical to Q.

Note Throughout this section we make the weak assumption that $\mathcal{A}_P^I \subseteq \mathcal{A}_Q^I \cup \mathcal{A}_Q^O$. This is to ensure that an environment E for which $Q \parallel E$ is a closed system (i.e., has no inputs) will also make $(P \parallel R) \parallel E$ a closed system. So any inputs to P will be provided by an output from the environment of Q or from R. In the latter case, such an input of P will be an output of Q. Further, we assume that the solution R satisfies $\mathcal{A}_P^O \subseteq \mathcal{A}_R^I$. This is to allow R to use the output actions of P as inputs in carrying out its control objectives. It is clear that any solution R will satisfy $\mathcal{A}_R^O \subseteq \mathcal{A}_Q^O \backslash \mathcal{A}_P^O$, and for the most general solution the two sets will be equal.

Notation We write $p \xrightarrow{a} p'$ if $a \in \Gamma_P(p)$ and $\delta(p, a) = p'$ for states p, p' and action a in an interface automaton P. We call $p \xrightarrow{a} p'$ an *input transition* if a is an input action of P. An output transition is defined similarly.

First we prove a result about compatibility that is used in Theorem 3.3 below.

Lemma 3.2. If P and $Q^{\perp}$ are compatible, then P and $(P \parallel Q^{\perp})^{\perp}$ are compatible.

Proof. Suppose P and $Q^{\perp}$ are compatible, but P and $(P \parallel Q^{\perp})^{\perp}$ are not. By Lemma 2.13, this means there exists a state $(p, (p', q)) \in \text{Reach}^O(P \otimes (P \parallel Q^{\perp})^{\perp})$ which is in Incomp$(P, (P \parallel Q^{\perp})^{\perp})$. Since the interface automata we consider are deterministic, it must be the case that $p = p'$. This implies that there exists an $a \in \text{Comm}(P, (P \parallel Q^{\perp})^{\perp})$ such that either (a) $a \in \Gamma^O_P(p)$ and $a \notin \Gamma^I_{(P\parallel Q^{\perp})^{\perp}}(p, q) = \Gamma^O_{(P\parallel Q^{\perp})}(p, q) = \Gamma^O_P(p) \cup \Gamma^I_Q(q)$, which is impossible, or (b) $a \notin \Gamma^I_P(p)$ and $a \in \Gamma^O_{(P\parallel Q^{\perp})^{\perp}}(p, q)$ which implies $(p, q) \xrightarrow{a} (p', q')$ is an input transition in $P \parallel Q^{\perp}$ and $p \xrightarrow{a} p'$ is not an input transition in P. This is possible only if $a \in \mathcal{A}^O_Q$ but $a \notin \mathcal{A}^I_P$, which contradicts our assumption that $a \in \text{Comm}(P, (P \parallel Q^{\perp})^{\perp})$.
$\square$

Theorem 3.3. A solution R to $P \parallel R \preceq Q$ exists iff P and $Q^{\perp}$ are compatible.

Proof. ($\Leftarrow$) Suppose P and $Q^{\perp}$ are compatible. By Lemma 3.2 so are P and $(P \parallel Q^{\perp})^{\perp}$. Take $R = (P \parallel Q^{\perp})^{\perp}$. We show that there exists an alternating simulation ρ between $P \parallel R$ and Q. Define the relation $\rho = \{((p, (p, q)), q) \mid (p, (p, q))$ is a state in $P \parallel R\}$. Since (s^0_P, s^0_Q) is the initial state of R, $(s^0_P, (s^0_P, s^0_Q))$ is the initial state of $P \parallel R$, and hence $((s^0_P, (s^0_P, s^0_Q)), s^0_Q)$ is in ρ. Now suppose $((p, (p, q)), q) \in \rho$ and $q \xrightarrow{a} q$ is an input transition in Q. It follows that $q \xrightarrow{a} q'$ is an output transition in $Q^{\perp}$. Therefore, $p \xrightarrow{a} p'$ is an input transition in P for some p', since (p, q), being in $P \parallel Q^{\perp}$, is backward compatible in $P \otimes Q^{\perp}$. Hence $(p, q) \xrightarrow{a} (p', q')$ is an output transition in $P \parallel Q^{\perp}$, and so an input transition in $(P \parallel Q^{\perp})^{\perp}$, whence $(p, (p, q)) \xrightarrow{a} (p', (p', q')$ is an input transition in $P \parallel (P \parallel Q^{\perp})^{\perp}$ and by definition of ρ, $((p', (p', q')), q')$ is again in ρ. Similarly for the output side, suppose $((p, (p, q)), q) \in \rho$ and $(p, (p, q)) \xrightarrow{a} (p', (p'', q'))$ is an output transition in $P \parallel (P \parallel Q^{\perp})^{\perp}$. Since we consider only deterministic automata, $p' = p''$. Also, it must be the case that $a \in \text{Comm}(P, (P \parallel Q^{\perp})^{\perp})$, because an output action of P is an output action of $P \parallel Q^{\perp}$, and therefore an input action of $(P \parallel Q^{\perp})^{\perp}$. Suppose $p \xrightarrow{a} p'$ is an output transition in P, and because P and $Q^{\perp}$ are compatible, and (p, q) is backward compatible in $P \otimes Q^{\perp}$, $q \xrightarrow{a} q'$ is an input transition in $Q^{\perp}$, and hence an output transition in Q. On the other hand, if $p \xrightarrow{a} p'$ is an input transition in P, then since $(p, (p, q)) \xrightarrow{a} (p', (p', q'))$ is an output transition in $P \parallel (P \parallel Q^{\perp})^{\perp}$, $(p, q) \xrightarrow{a} (p', q')$ is an output transition in $(P \parallel Q^{\perp})^{\perp}$, and therefore an input transition in $(P \parallel Q^{\perp})$. From the assumption that $\mathcal{A}^I_P \subseteq \mathcal{A}^I_Q$ and by the definition of the product $P \otimes Q^{\perp}$ it follows that $q \xrightarrow{a} q'$ is an input transition of $Q^{\perp}$, and hence an output transition of Q. By the definition of ρ, $((p', (p', q')), q') \in \rho$, hence ρ is an alternating simulation as required.

($\Rightarrow$) We show the contrapositive. Suppose P and $Q^{\perp}$ are not compatible. Then, by Lemma 2.13, there exists a state $(p, q) \in \text{Reach}^O(P, Q)$ which is incompatible, i.e., there is an a such that either (a) $a \in \Gamma^O_P(p)$ and $a \notin \Gamma^O_Q(q)$ or (b) $a \notin \Gamma^I_P(p)$ and $a \in \Gamma^I_Q(q)$. Both possibilities rule out the existence of an alternating simulation between $P \parallel R$ and Q for any R.
$\square$

Theorem 3.4.
When the condition stated in Theorem 3.3 is satisfied, the most general solution to $P \parallel R \preceq Q$ exists and is given by $R = (P \parallel Q^{\perp})^{\perp}$.

Proof. In the proof of Theorem 3.3 (If part) we have already shown that $R = (P \parallel Q^{\perp})^{\perp}$ is a solution. Suppose U is any solution to $P \parallel R \preceq Q$. We construct an alternating simulation ν from U to $(P \parallel Q^{\perp})^{\perp}$ as follows. By assumption, there exists an alternating simulation ρ from $P \parallel U$ and Q. Define $\nu = \{(u, (p, q)) \mid ((p, u), q) \in \rho\}$. Clearly $(s^0_U, (s^0_P, s^0_Q)) \in \nu$, since $((s^0_P, s^0_U), s^0_Q) \in \rho$. Now suppose $(u, (p, q)) \in \nu$ and $u \xrightarrow{a} u'$ is an output transition in U. This implies $p \xrightarrow{a} p'$ is an input transition in P for some p', since by assumption $((p, u), q) \in \rho$ and therefore (p, u) is backward compatible in $P \otimes U$. Hence, $(p, u) \xrightarrow{a} (p', u')$ is an output transition in $P \parallel U$. It follows that $q \xrightarrow{a} q'$ is an output transition in Q for some q', with $((p', u'), q') \in \rho$, which is equivalent to $q \xrightarrow{a} q'$ is an input transition in $Q^{\perp}$. Therefore, $(p, q) \xrightarrow{a} (p', q')$ is an input transition in $P \parallel Q^{\perp}$, since (p, q) is backward

compatible in $P \otimes Q^\perp$ by assumption. It follows that $(p,q) \xrightarrow{a} (p',q')$ is an output transition in $(P \parallel Q^\perp)^\perp$ and $(u',(p',q')) \in \nu$ as required. Next suppose $(u,(p,q)) \in \nu$ and $(p,q) \xrightarrow{a} (p',q')$ is an input transition in $(P \parallel Q^\perp)^\perp$, which is the same as $(p,q) \xrightarrow{a} (p',q')$ is an output transition in $P \parallel Q^\perp$. This implies that either (a) $p \xrightarrow{a} p'$ is an input transition in P and $q \xrightarrow{a} q'$ is an input transition in Q or (b) $p \xrightarrow{a} p'$ is an output transition in P and $q \xrightarrow{a} q'$ is an output transition in Q. For the first case, by the existence of the alternating simulation ρ, $(p,u) \xrightarrow{a} (p',u')$ is an input transition in $P \parallel U$ for some state u' in U with $((p',u'),q') \in \rho$ and hence $(u',(p',q')) \in \nu$. For the second case, $u \xrightarrow{a} u'$ is an input transition in U for some u', since (p,u) is backward compatible in $P \otimes U$. Further $(u',(p',q')) \in \nu$, since $((p',u'),q') \in \rho$, and the conclusion follows.
□

4. Winning Strategies and Synthesis

We now characterise the most general solution to $P \parallel R \preceq Q$ in terms of winning strategies. Specifically, we show that the most general solution corresponds to the *most permissive* winning strategy for Input in $P \otimes (P \parallel Q^\perp)^\perp$.

First we define winning strategies for Input and Output in games corresponding to the product $P \otimes Q$ of two interface automata P and Q. We also define a natural partial order $\sqsubseteq^I$ on input strategies, such that $\sigma_P^I \sqsubseteq^I \tau_P^I$ if the strategy τ_P^I generates more inputs than σ_P^I at every state of P. A similar order $\sqsubseteq^O$ is defined on output strategies. Since the orders are lattices, the *most permissive strategy* exists, as is given by the lattice join. We then show that the parallel composition $P \parallel Q$ can be extracted from the most permissive winning strategy for Input.

Definition 4.1. Let P and Q be composable interface automata. A *winning input strategy* for $P \otimes Q$ is an input strategy π^I such that for all output strategies π^O, all initial states $s_0 \in S^0_{P\otimes Q}$, all $\sigma \in \text{Outcomes}_{P\otimes Q}(s_0,\pi^I,\pi^O)$, and all incompatible states $w \in \text{Incomp}(P,Q)$, the state w does not appear in the sequence σ. The definition of a winning output strategy is symmetric, where the winning condition is that a state in Incomp(P,Q) must be reached in every run $\sigma \in \text{Outcomes}_{P\otimes Q}(s_0,\pi^I,\pi^O)$.

We now define the order $\sqsubseteq$ on strategies. The idea is that an input strategy is higher in the order if it accepts more inputs. Dually an output strategy is higher in the order if it generates more outputs.

Definition 4.2. The binary relation $\sqsubseteq^I$ on input strategies for P is defined by $\pi_0^I \sqsubseteq \pi_1^I$ iff $\pi_0^I(\sigma) \subseteq \pi_1^I(\sigma)$ for all $\sigma \in S_P^+$. When $\pi_0^I \sqsubseteq \pi_1^I$, we say π_1^I is *more permissive than* π_0^I. Similarly, for output strategies, $\pi_0^O \sqsubseteq^O \pi_1^O$ iff $\pi_0^O(\sigma) \subseteq \pi_1^O(\sigma)$ for all $\sigma \in S_P^+$.

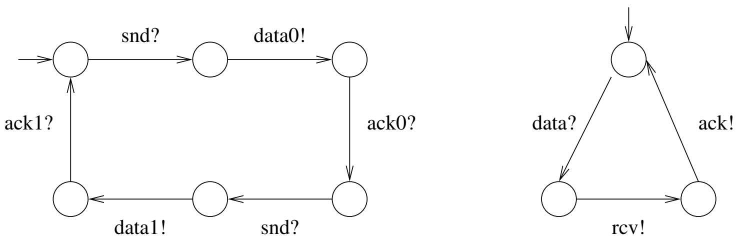
Clearly, the relations $\sqsubseteq^I$ and $\sqsubseteq^O$ are lattices, with top elements $\pi_T^I(\sigma s) = \Gamma_P^I(s)$ and $\pi_T^O(\sigma s) = \Gamma_P^O(s)$, and join and meet given by pointwise union and intersection. Note that the bottom elements are the empty strategies, which are allowed by the definition of strategies.

Corollary 4.3. If there is a winning strategy for either player in a game then there is a most permissive winning strategy for that player.

Proof. Simply take the join of the set of all winning strategies for the player.
□

Next we show how to extract an interface automaton $\pi^I(P \otimes Q)$ from an input strategy π^I for the game $P \otimes Q$, by cutting down some of its states and transitions.

Definition 4.4. The interface automaton $\pi^I(P \otimes Q)$ *defined by input strategy* π^I for the game $P \otimes Q$ is defined as follows. Its set of input and output actions are the same as those of $P \otimes Q$. The set $S_{\pi^I(P\otimes Q)}$ contains those states of $P \otimes Q$ that are reached in some sequence in $\text{Outcomes}_{P\otimes Q}(s^0_{P\otimes Q},\pi^I,\pi_T^O)$, where π_T^O is the top output strategy in the lattice of strategies (the one that produces the most output). The input moves of $\pi^I(P \otimes Q)$ are defined by $\Gamma^I(s) = \{a \mid a \in \Gamma^I_{P\otimes Q}$ such that $a \in \pi^I(\sigma s)$ for some $\sigma \in S^+_{\pi^I(P\otimes Q)}\}$. The input transitions of $\pi^I(P \otimes Q)$ are defined by $\delta(s,a) = \delta_{P\otimes Q}(s,a)$ when $a \in \Gamma^I(s)$ and an arbitrary element of $S_{\pi^I(P\otimes Q)}$ otherwise. The output moves and transitions are the straightforward restrictions of the output moves and transitions of $P \otimes Q$ to the set of states $S_{\pi^I(P\otimes Q)}$.



(a) Alternating Bit Sender (b) Nonsequenced Receiver

Fig. 2. Two mismatched protocols

The following proposition states that the parallel composition $P \parallel Q$ of interface automata P and Q is the interface automaton $\pi^I_w(P \otimes Q)$ defined by input strategy π^I_w for the game $P \otimes Q$, where π^I_w is the most permissive winning input strategy, if one exists.

Proposition 4.5. For composable interface automata P and Q, $P \parallel Q$ can be obtained as $\pi^I_w(P \otimes Q)$ where π^I_w is the most permissive winning input strategy for $P \otimes Q$. If no winning input strategy exists then P and Q are incompatible.

Proof. By Definition 4.1, if no winning input strategy exists, there exists an output strategy π^O such that an incompatible state appears in some sequence $\sigma \in \text{Outcomes}_{P\otimes Q}(s^0_{P\otimes Q}, \pi^I, \pi^O)$, for all input strategies π^I. From Definition 2.10, this implies that the set T of backward compatible states is empty, and hence by Definition 2.11 the composition $P \parallel Q$ is empty. Suppose there is a winning input strategy for $P \otimes Q$. We show that the set of states $S_{\pi^I_w(P\otimes Q)}$ is identical with the backward compatible states T of $P \otimes Q$, where π^I_w is the most permissive winning input strategy for $P \otimes Q$. Suppose $s \in S_{\pi^I_w(P\otimes Q)}$. Since π^I_w is a winning strategy, s satisfies the first clause in Definition 2.10 of backward compatibility. By Definition 4.4, s is reached in some play in $\text{Outcomes}_{P\otimes Q}(s^0_{P\otimes Q}, \pi^I_w, \pi^O)$ and therefore s satisfies the second clause as well. Now suppose s is a backward compatible state of $P \otimes Q$. By Definition 2.10 there exists a winning input strategy π^I and some output strategy π^O for $P \otimes Q$, for which s appears in some play $\sigma \in \text{Outcomes}_{P\otimes Q}(s^0_{P\otimes Q}, \pi^I, \pi^O)$. It follows that s appears in some play in $\text{Outcomes}_{P\otimes Q}(s^0_{P\otimes Q}, \pi^I_w, \pi^O)$, and by Definition 4.4, s is in $S_{\pi^I_w(P\otimes Q)}$.
□

Next we characterise solutions to $P \parallel R \preceq Q$ in terms of winning strategies for Input in $(P \otimes Q^\perp)^\perp$, and show that the most general solution arises from the most permissive strategy.

Theorem 4.6. A solution to $P \parallel R \preceq Q$ exists iff a winning input strategy π exists for $(P \otimes Q^\perp)^\perp$. The most general solution to $P \parallel R \preceq Q$ is given by $\pi^I_w((P \otimes Q^\perp)^\perp)$, where π^I_w is the most permissive winning input strategy.

Proof. From Theorems 3.3 and 3.4 it follows that a solution exists iff P and $Q^\perp$ are compatible, and in such a case $R = (P \parallel Q^\perp)^\perp$ is the most general solution. By Proposition 4.5, $(P \parallel Q^\perp)^\perp = \pi^I_w((P \otimes Q^\perp)^\perp)$ where π^I_w is the most permissive winning strategy for $(P \otimes Q^\perp)^\perp$.
□

Computing Winning Strategies The calculation of winning strategy in such safety games, if one exists, is by standard iterative refinement using the *controllable predecessors* operator [Tho95]. The complexity of the algorithm is linear in the size of the game graph. Since the game graph in computing the composition $P \parallel Q$ of two interface automata P and Q is given by the product $P \otimes Q$, computing $P \parallel Q$ can be performed in time $O(|P||Q|)$. Here $|P|$ is the size of P, given by the sum of the states and transitions in P. It follows that $(P \parallel Q^\perp)^\perp$ is also computable in $O(|P||Q|)$, since $P^\perp$ can be obtained from P in linear time.

5. Application: Network Protocol Conversion

In this section we describe an application of interface synthesis to the protocol conversion problem. In today's world, global communication over heterogeneous networks of computers can often lead to protocol mismatches between

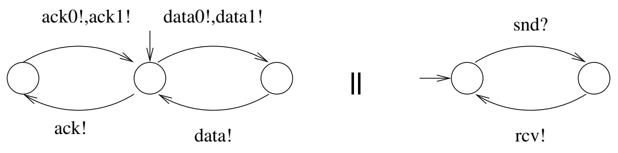


Fig. 3. Specification of Converter

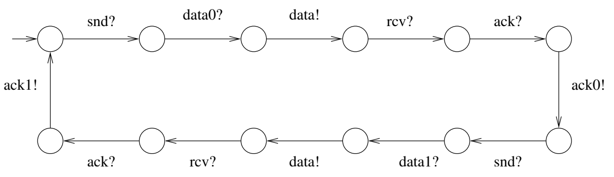


Fig. 4. Converter for the two protocols

communicating entities. The lack of a uniform global standard for communication protocols entails that protocol converters have to be built for mediating between incompatible protocols [Lam88, CL90]. We illustrate the use of interface synthesis to the protocol conversion problem through an example adapted from [KNM97].

Consider the two interface shown in Fig. 2 representing two incompatible protocols. Fig. 2(a) is a simplified version of a sender using the Alternating Bit Protocol (ABP), while the one in Fig. 2(b) is a receiver using the Nonsequenced Protocol (NS). The ABP sender accepts data from the user (a higher level protocol) using the input action `snd?` and transmits it with label 0 using output action `data0!`. After receiving an acknowledgement with the correct label 0 via the input action `ack0?`, the sender is ready to accept the next piece of data from the user and transmit it with label 1. The protocol performs in a loop, alternating labels between 0 and 1. In this simplified version we ignore retransmissions due to timeouts and receipt of acknowledgements with wrong labels.

The NS receiver in Fig. 2(b) is much simpler, which on receiving a data packet via input action `data?`, delivers it to the user via the output action `rcv!`, and sends an acknowledgement to the sender via `ack!`. Since the NS receiver does not use any labels for the data and acknowledgement packets there is a protocol mismatch between ABP and NS.

When we want the two protocols above to work together without causing any inconsistency by using a converter, we need to specify what the converter is allowed and not allowed to do. This idea was proposed in [PdAHSV02] in the setting of synchronous hardware-like protocols. We require that the system as a whole (the two protocols along with the converter) satisfies the interface described by Fig. 3. This specification interface is obtained as the parallel composition of two interfaces. The one on the left specifies that the converter can send data packets and acknowledgements to the NS receiver and ABP sender, only after receiving a data packet or acknowledgement from the other protocol. No data or acknowledgement can be sent speculatively, nor can packets be lost or duplicated. The interface on the right specifies the overall behaviour that the user expects from the system: the `snd` and `rcv` events will alternate strictly in any system run. Note that every action in Fig. 3 is of type output, except for `snd?`.

The correct converter for the two protocols is shown is Fig. 4. The converter can be obtained be as follows. Let P be the parallel composition of the two protocols which need conversion. Since we assume the two sets of actions to be disjoint, the composition is always well defined. The specification S for the converter relates the two actions sets by specifying temporal ordering of actions. For instance, in our example, the specification dictates that a `data` action can only follow a corresponding `data0` or `data1` action. The converter C is then the (most general) solution to $P \parallel C \preceq S$. Intuitively, the goal of the converter is to meet the specification, while satisfying the input assumptions of the two protocols. Moreover, the converter can control only the inputs to the protocols and not their outputs.

6. Synchronous Interface Automata

In earlier sections we have presented the synthesis problem and its solution for interface automata, a formalism intended for component-based modelling and development of asynchronous systems. Typical examples include software modules interacting through method invocations, distributed systems and network protocols. In this section we want to extend the synthesis problem to the synchronous setting, where all actions are triggered by clock ticks, as in sequential circuits and synchronous reactive programs. One of the motivations for studying the synthesis problem in the synchronous setting is to enable better reuse of intellectual property (IP) blocks in system-on-chip (SoC) designs.

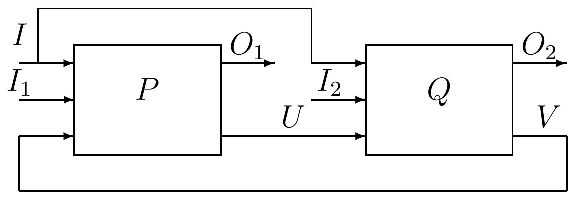


Fig. 5. Block diagram for $P \otimes Q$

Automated design reuse by composing IP blocks is inherently difficult because components often come from different manufacturers and are designed using different protocols of interactions.

We propose the *synchronous interface automata* (SIA) model as the synchronous counterpart of interface automata. The interface synthesis problem considered in Section 3 – find the most general interface R, which combined with P is a refinement of Q – is reconsidered in this framework with appropriate modifications in the definitions. We show that in our SIA framework the solution is again given by $R = (P \parallel Q^{\perp})^{\perp}$. From this general framework we are able to derive a solution to the specific problem of converter synthesis for mismatched protocols in SoC designs.

Our SIA model should be contrasted with the synchronous Moore interfaces proposed in [CdAHM02] to model interactions between components typical in hardware. The main differences are, we use Mealy rather than Moore machines, and instead of specifying initial states and transitions in terms of predicates on state variables, we take the state transition framework, where transitions are triggered by input signals and emit output signals. The advantage of the Mealy framework is that our systems satisfy the *synchrony hypothesis*, i.e., have zero response time, which is an useful abstraction at the specification level (see [BB91]). The price to pay is the difficulty in composition due to the possibility of causality cycles. We come back to this point later.

Our SIA model essentially defines Mealy automata with explicit input assumptions and output guarantees. The game view of interface composition and refinement then applies *mutatis mutandis* to the synchronous setting. Of course, the specific details of the SIA formalism, as described in the following paragraphs, are quite different from the (asynchronous) interface automata formalism.

We start by fixing some notation and conventions. An *I/O-signature* is a pair $[\overrightarrow{\mathbf{I}}\ ;\ \overrightarrow{\mathbf{O}}]$, where $\overrightarrow{\mathbf{I}} = \mathbf{I}_1, \ldots, \mathbf{I}_n$ and $\overrightarrow{\mathbf{O}} = \mathbf{O}_1, \ldots, \mathbf{O}_m$ are disjoint lists of *input* and *output variables*. I/O-signatures are used to identify input and output lines when composing synchronous interfaces, as in Fig. 5. We use the vector notation to denote lists and suppress their lengths. Each input variable $\mathbf{I}_k$ is interpreted over a finite set I_k called the domain of $\mathbf{I}_k$, and likewise each output variable $\mathbf{O}_j$ is interpreted over a finite set O_j. Input values in the sets I_k are denoted $i_1, i_2, \ldots$, while output values in O_j are denoted $o_1, o_2, \ldots$. We refer to the set $\overrightarrow{I} = I_1 \times \ldots \times I_n$ interpreting all the input variables as the *input space* or *input alphabet*, and $\overrightarrow{O} = O_1 \times \ldots \times O_m$ as the *output space* or *output alphabet*.

Definition 6.1. A *synchronous interface automaton* (SIA) P with I/O signature $[\overrightarrow{\mathbf{I}}\ ;\ \overrightarrow{\mathbf{O}}]$ is a tuple $(S_P, S_P^0, \mathcal{A}_P^I, \mathcal{A}_P^O, \delta_P)$ where:

- S_P is a finite set of *states*.
- $S_P^0 \subseteq S_P$ is the set of *initial states*, which has at most one element.
- $\mathcal{A}_P^I = I_1 \times I_2 \times \ldots I_n$ and $\mathcal{A}_P^O = O_1 \times O_2 \times \ldots O_m$ are the *input* and *output alphabets*.
- $\delta_P : S_P \times \mathcal{A}_P^I \times \mathcal{A}_P^O \rightarrow S_P$ is a (partial) *transition function* associating a target state $\delta_P(s, i, o)$ with each state $s \in S_P$ and input and output values i and o, when it is defined.

As in the asynchronous case, the SIA P is said to be *empty* when its set of initial states S_P^0 is empty. Again, such automata arise when incompatible automata are composed.

The meaning of $\delta_P(s, i, o) = s'$ is that the SIA P can transit from state s to s' on input i, and perform the output o. Although a given pair (s, i) of current state and input value does not uniquely determine the next state, the triple (s, i, o) of current state with input and output values certainly does, when it is defined. This is the property of *observable nondeterminism*. It allows us to treat SIA as deterministic automata when we forget the distinction between inputs and outputs by clubbing them together.

Notation We write $p \xrightarrow[o]{i} p'$ if $\delta_P(p,i,o) = p'$ for states p, p' and input-output action (i,o) in an SIA P. Also, the set of input and output actions possible at a state p in P are denoted $\mathcal{A}_P^I(p) = \{i \mid p \xrightarrow[o]{i} p' \text{ for some } o \text{ and } p'\}$ and $\mathcal{A}_P^O(p) = \{o \mid p \xrightarrow[o]{i} p' \text{ for some } i \text{ and } p'\}$ respectively.

Definition 6.2. An *execution fragment* of an SIA P is a finite alternating sequence of states and input-output values $s_0, (i_0, o_0), s_1, (i_1, o_1), \ldots, s_n$ such that $\delta_P(s_k, i_k, o_k) = s_{k+1}$ for all $0 \leq k < n$. Given two states $s, s' \in S_P$, we say that s' is *reachable from* s if there is an execution fragment whose first state is s, and whose last state is s'. A state s' is *reachable* in P if there exists an initial state $s \in S_P^0$ such that s' is reachable from s. Let $\text{Reach}(P)$ denote the set of reachable states of P.

As in the asynchronous case we define an input strategy as a predetermined way of choosing the input at any stage of the game.

Definition 6.3. An *input strategy* for P is a map $\pi^I : S_P \to \mathcal{A}_P^I$.

Note We consider only memoryless strategies here, since they suffice for safety games. Also, we restrict to *deterministic* strategies – at every state there is exactly one choice of input action. This is in keeping with the deterministic nature of synchronous hardware.

Given an input strategy π^I, only a subset of states in $\text{Reach}(P)$ can be reached. Let $\text{Reach}(P, \pi^I) \subseteq S_P$, the *states reached under input strategy* π^I, be defined inductively as follows:

- $S_P^0 \subseteq \text{Reach}(P, \pi^I)$, and
- for all $s \in \text{Reach}(P, \pi^I)$ and $o \in \mathcal{A}_P^O$, $\delta_P(s, \pi^I(s), o) \in \text{Reach}(P, \pi^I)$.

The set $\text{Reach}(P, p, \pi^I)$ of states reached under π^I starting from state p in P is defined in the obvious way.

The composition of two SIA is a partial operation, as two synchronous interfaces may not be compatible. We give the precise definitions below. Two SIA P and Q with I/O signatures $[\overrightarrow{\mathbf{I}} \; ; \; \overrightarrow{\mathbf{O}}]$ and $[\overrightarrow{\mathbf{I}'} \; ; \; \overrightarrow{\mathbf{O}'}]$ are *composable* if they don't share an output variable, i.e., $\overrightarrow{\mathbf{O}} \cap \overrightarrow{\mathbf{O}'} = \emptyset$.

To define composition, we first define the product of two synchronous interfaces, just as in the asynchronous case. In the definition below, we require that $\overrightarrow{\mathbf{I}_1} \cap \overrightarrow{\mathbf{O}_2} = \emptyset$ and $\overrightarrow{\mathbf{O}_1} \cap \overrightarrow{\mathbf{I}_2} = \emptyset$. In the signatures of the composable SIA P and Q, $\overrightarrow{\mathbf{I}}$ is the list of shared input variables. The output variables $\overrightarrow{\mathbf{U}}$ of P and $\overrightarrow{\mathbf{V}}$ of Q appear as input variables of the other automaton, as in Fig. 5.

Definition 6.4. Let P and Q be two composable SIA with I/O signatures $[\overrightarrow{\mathbf{I}}, \overrightarrow{\mathbf{I}_1}, \overrightarrow{\mathbf{V}} \; ; \; \overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{O}_1}]$ and $[\overrightarrow{\mathbf{I}}, \overrightarrow{\mathbf{I}_2}, \overrightarrow{\mathbf{U}} \; ; \; \overrightarrow{\mathbf{V}}, \overrightarrow{\mathbf{O}_2}]$. Thus $\mathcal{A}_P^I = I \times I_1 \times V$ and $\mathcal{A}_P^O = U \times O_1$ are the input and output alphabets of P, and $\mathcal{A}_Q^I = I \times I_2 \times U$ and $\mathcal{A}_Q^O = V \times O_2$ the respective alphabets of Q. Then the product $P \otimes Q$ of SIA P and Q with I/O signature $[\overrightarrow{\mathbf{I}}, \overrightarrow{\mathbf{I}_1}, \overrightarrow{\mathbf{I}_2} \; ; \; \overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{V}}, \overrightarrow{\mathbf{O}_1}, \overrightarrow{\mathbf{O}_2}]$ is defined by the tuple $(S_{P\otimes Q}, S_{P\otimes Q}^0, \mathcal{A}_{P\otimes Q}^I, \mathcal{A}_{P\otimes Q}^O, \delta_{P\otimes Q})$ where

- $S_{P\otimes Q} = S_P \times S_Q$
- $S_{P\otimes Q}^0 = S_P^0 \times S_Q^O$
- $\mathcal{A}_{P\otimes Q}^I = I \times I_1 \times I_2$
- $\mathcal{A}_{P\otimes Q}^O = U \times V \times O_1 \times O_2$
- $\delta_{P\otimes Q}((p,q), (i, i_1, i_2), (u, v, o_1, o_2))$ is defined to be the pair (p', q') where $p' = \delta_P(p, (i, i_1, v), (u, o_1))$ and $q' = \delta_Q(q, (i, i_2, u), (v, o_2))$, when both are defined.

The block diagram for the product $P \otimes Q$ is illustrated in Fig. 5. Note that the above definition and Fig. 5 describe the most general situation. For instance, if P and Q do not share any input signal then $\overrightarrow{\mathbf{I}}$ is the empty tuple, so P has signature $[\overrightarrow{\mathbf{I}_1}, \overrightarrow{\mathbf{V}} \; ; \; \overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{O}_1}]$ and Q has signature $[\overrightarrow{\mathbf{I}_2}, \overrightarrow{\mathbf{U}} \; ; \; \overrightarrow{\mathbf{V}}, \overrightarrow{\mathbf{O}_2}]$.

The transition function $\delta_{P\otimes Q}$ has the following interpretation: when $P \otimes Q$ is in state (p,q) and there is a P-transition from p that accepts (i, i_1, v) as input and generates (u, o_1) as output and a Q-transition from q that accepts (i, i_2, u) as input and generates (v, o_2) as output, then there is a transition from (p,q) that accepts (i, i_1, i_2) as input and generates (u, v, o_1, o_2) as output.

Intuitively, a state (p,q) in the product $P \otimes Q$ is locally compatible if the environment can provide a suitable input such that both P and Q can separately satisfy the input assumption of the other SIA in states p and q respectively. Otherwise the state is locally incompatible. We say that two SIA P and Q are compatible, if there is a way to provide inputs to $P \otimes Q$ so that locally incompatible states are not reached.

Definition 6.5. Let P and Q be two SIA with I/O signatures as above. The set of *locally compatible states* of P and Q consist of all pairs $(p, q) \in S_P \times S_Q$ such that the following two conditions are satisfied:

1. there exist i, i_1, i_2, v for which there is a transition $p \overset{i, i_1, v}{\underset{u, o_1}{\longrightarrow}} p'$ in P, and for all such o_1, u, p', there exist o_2, q' with $q \overset{i, i_2, u}{\underset{o_2, v}{\longrightarrow}} q'$ in Q;
2. there exist i, i_1, i_2, u for which there is a transition $q \overset{i, i_2, u}{\underset{o_2, v}{\longrightarrow}} q'$ in Q, and for all such o_2, v, q', there exist o_1, p' with $p \overset{i, i_1, v}{\underset{u, o_1}{\longrightarrow}} p'$ in P.

The set Incomp(P, Q) of *locally incompatible states* of P and Q is the set of states in $P \otimes Q$ which are not locally compatible.

Just as in the asynchronous case, we would like a benign environment to provide the right inputs to avoid hitting the locally incompatible states. The states from which this is possible are called compatible.

Definition 6.6. Let P and Q be two composable SIA. A state (p, q) in $P \otimes Q$ is *compatible* if there is an input strategy π^I for $P \otimes Q$, such that Reach$(P \otimes Q, (p, q), \pi^I)$ does not contain a locally incompatible state of $P \otimes Q$. We write Comp(P, Q) for the set of compatible states of $P \otimes Q$. Two SIA P and Q are *compatible* if the sole initial state of $P \otimes Q$ is compatible.

Definition 6.7. The composition $P \parallel Q$ of two SIA is defined by restricting the product $P \otimes Q$ to the set of compatible states:

- $S_{P\parallel Q} =$ Comp(P,Q);
- $S^0_{P\parallel Q} = S^0_{P\otimes Q} \cap$ Comp(P,Q);
- $\mathcal{A}^I_{P\parallel Q} = \mathcal{A}^I_{P\otimes Q}$;
- $\mathcal{A}^O_{P\parallel Q} = \mathcal{A}^O_{P\otimes Q}$;
- for all $s \in$ Comp(P,Q), $i \in \mathcal{A}^I_{P\parallel Q}(s)$, $o \in \mathcal{A}^O_{P\parallel Q}(s)$, $\delta_{P\parallel Q}(s, i, o) = \delta_{P\otimes Q}(s, i, o)$ if $\delta_{P\otimes Q}(s, i, o) \in$ Comp(P,Q), and it is undefined otherwise.

The motivation for the above definitions are exactly the same as in the asynchronous case, and should not come as a surprise. To summarise, P and Q are considered compatible if there is some environment in which they can be used together without violating each other's input assumption. This is equivalent to saying that there is a winning input strategy in the product $P \otimes Q$. As pointed out earlier, the solution of such safety games is entirely classical.

Note The SIA model essentially defines Mealy automata, the novelty being in the definition of composition using the game interpretation. It is well known that the synchronous composition of non-blocking Mealy automata may have *causality cycles* – circular dependencies between input and output signals in the composed Mealy automaton. We assume that all our SIA are *statically typed*, i.e., the dependencies between input and output signals are fixed. When composing two SIA we require that the the combined dependency relation is acyclic. This condition can be enforced syntactically and checked in linear time – see [dAHM00] for details.

As in the asynchronous case, the game view of interfaces leads to *alternating refinement* [AHKV98] as the correct notion of refinement. Informally, $P \preceq Q$ (P refines Q) if all legal inputs of Q are also legal for P, and when P and Q are fed the same legal input, Q generates more output than P does. This definition ensures that whenever $P \preceq Q$, P can safely be substituted for Q in any design without creating any incompatibility.

Definition 6.8. Let P and Q be two SIA with identical I/O signatures. An *alternating simulation* ρ from P to Q is a relation $\rho \subseteq S_P \times S_Q$ such that, for all $(s, t) \in \rho$ the following conditions are satisfied:

1. $\mathcal{A}^I_Q(t) \subseteq \mathcal{A}^I_P(s)$;

2. $\mathcal{A}^O_P(s) \subseteq \mathcal{A}^O_Q(t)$;
3. $(\delta_P(s,a), \delta_Q(t,a)) \in \rho$ for all $a \in \mathcal{A}^I_Q(t) \times \mathcal{A}^O_P(s)$,

Given two SIA P and Q with identical I/O signatures, we say P *refines* Q, written $P \preceq Q$, if the following conditions are satisfied:

1. $\mathcal{A}^I_Q \subseteq \mathcal{A}^I_P$;
2. $\mathcal{A}^O_P \subseteq \mathcal{A}^O_Q$;
3. there is an alternating simulation ρ from P to Q, such that $(s^0, t^0) \in \rho$ for some $s^0 \in S^0_P$ and $t^0 \in S^0_Q$.

7. Synthesis of Synchronous Interfaces

In this section we revisit the synthesis problem, in the context of SIA. We prove the synchronous analogue of Theorems 3.3 and 3.4: the most general solution to $P \parallel R \preceq Q$ exists iff P and $Q^\perp$ are compatible and is given by $R = (P \parallel Q^\perp)^\perp$. Here $P^\perp$ has the same interpretation as in the asynchronous case.

Note Throughout the section we assume that the list of output variables of Q includes all the output variables of P and the input variables of P that are not input variables of Q: $\mathcal{A}^I_P \subseteq \mathcal{A}^I_Q \cup \mathcal{A}^O_Q$ and $\mathcal{A}^O_P \subseteq \mathcal{A}^O_Q$. So any inputs to P will be provided by an output from the environment of Q or from R. In the latter case, such an input of P will be an output of Q. We fix the I/O signatures of the various interfaces involved, once and for all:

$P \,:\, [\overrightarrow{\mathbf{I_1}}, \overrightarrow{\mathbf{V}} \,;\, \overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{O_1}}]$
$Q \,:\, [\overrightarrow{\mathbf{I_2}}, \overrightarrow{\mathbf{V}} \,;\, \overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{I_1}}, \overrightarrow{\mathbf{O_1}}, \overrightarrow{\mathbf{O_2}}]$
$Q^\perp \,:\, [\overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{I_1}}, \overrightarrow{\mathbf{O_1}}, \overrightarrow{\mathbf{O_2}} \,;\, \overrightarrow{\mathbf{I_2}}, \overrightarrow{\mathbf{V}}]$
$P \otimes Q^\perp \,:\, [\overrightarrow{\mathbf{I_1}}, \overrightarrow{\mathbf{O_2}} \,;\, \overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{V}}, \overrightarrow{\mathbf{I_2}}, \overrightarrow{\mathbf{O_1}}]$
$(P \parallel Q^\perp)^\perp \,:\, [\overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{V}}, \overrightarrow{\mathbf{I_2}}, \overrightarrow{\mathbf{O_1}} \,;\, \overrightarrow{\mathbf{I_1}}, \overrightarrow{\mathbf{O_2}}]$
$P \otimes (P \parallel Q^\perp)^\perp \,:\, [\overrightarrow{\mathbf{I_2}}, \overrightarrow{\mathbf{V}} \,;\, \overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{I_1}}, \overrightarrow{\mathbf{O_1}}, \overrightarrow{\mathbf{O_2}}]$

Notice that Q and $P \otimes (P \parallel Q^\perp)^\perp$ have the same I/O signature.

First we prove a result about compatibility that is used in Theorem 7.2 below. Here we make use of the fact that if $(p, (p', q))$ is a reachable state in $P \otimes (P \parallel Q^\perp)^\perp$, then it follows from the property of observable nondeterminism that $p = p'$.

Lemma 7.1. If P and $Q^\perp$ are compatible, then P and $(P \parallel Q^\perp)^\perp$ are compatible.

Proof. Suppose P and $Q^\perp$ are compatible, but P and $(P \parallel Q^\perp)^\perp$ are not compatible. This means that for all input strategies π^I in $P \otimes (P \parallel Q^\perp)^\perp$, there exists a state $(p, (p, q))$ in Reach$(P \otimes (P \parallel Q^\perp)^\perp, \pi^I)$ such that $(p, (p, q)) \in$ Incomp$(P, (P \parallel Q^\perp)^\perp)$. It follows from Definition 6.5 that at least one of the following cases must hold:

1. For all i_1, i_2, v there exist u, o_1, p' such that $p \xrightarrow[u, o_1]{i_1, v} p'$ is in P, but there do not exist o_2, q' for which there is a transition $(p, q) \xrightarrow[i_1, o_2]{u, v, i_2, o_1} (p', q')$ is in $(P \parallel Q^\perp)^\perp$. Now, since P and $Q^\perp$ are compatible, and (p, q) is a state in $P \parallel Q^\perp$ by assumption, there exist i_2, v, o_2, q' such that $q \xrightarrow[i_2, v]{u, i_1, o_1, o_2} q'$ is in $Q^\perp$.But this implies $(p, q) \xrightarrow[u, v, i_2, o_1]{i_1, o_2} (p', q')$ is in $P \parallel Q^\perp$, and hence $(p, q) \xrightarrow[i_1, o_2]{u, v, i_2, o_1} (p', q')$ is in $(P \parallel Q^\perp)^\perp$, which is a contradiction.
2. For all u, v, i_2, o_1 there exist i_1, o_2, p', q' such that $(p, q) \xrightarrow[i_1, o_2]{u, v, i_2, o_1} (p', q')$ is in $(P \parallel Q^\perp)^\perp$, but $p \xrightarrow[u, o_1]{i_1, v} p'$ is not in P. This is clearly not possible by the definition of product of SIA.

□

Theorem 7.2. A solution R to $P \parallel R \preceq Q$ exists iff P and $Q^\perp$ are compatible.

Proof. ($\Leftarrow$) Suppose P and $Q^\perp$ are compatible. By Lemma 7.1 so are P and $(P \parallel Q^\perp)^\perp$. Take $R = (P \parallel Q^\perp)^\perp$. We show that there exists an alternating simulation ρ between $P \parallel R$ and Q that relates their initial states. Define the relation $\rho = \{((p,(p,q)),q) \mid (p,(p,q)) \text{ is a state in } P \parallel R\}$. Since (s_P^0, s_Q^0) is the initial state of R, $(s_P^0,(s_P^0,s_Q^0))$ is the initial state of $P \parallel R$, and hence $((s_P^0,(s_P^0,s_Q^0)),s_Q^0)$ is in ρ. Now suppose, for the output side, $(u,i_1,o_1,o_2) \in \mathcal{A}^O_{P\parallel R}((p,(p,q)))$, and there is a transition $(p,(p,q)) \xrightarrow[u,i_1,o_1,o_2]{i_2,v} (p',(p',q'))$ in $P \parallel R$. It follows that there exist transitions $p \xrightarrow[u,o_1]{i_1,v} p'$ in P and $(p,q) \xrightarrow[i_1,o_2]{u,v,i_2,o_1} (p',q')$ in $(P \parallel Q^\perp)^\perp$. So $(p,q) \xrightarrow[u,v,i_2,o_1]{i_1,o_2} (p',q')$ is a transition in $P \parallel Q^\perp$. Hence $q \xrightarrow[i_2,v]{u,i_1,o_1,o_2} q'$ is a transition in $Q^\perp$, and thus $q \xrightarrow[u,i_1,o_1,o_2]{i_2,v} q'$ is a transition in Q and $((p',(p',q'),q') \in \rho$ by the assumption that $(p',(p',q'))$ is a state in $P \parallel R$. Likewise, for the input side, suppose $q \xrightarrow[u,i_1,o_1,o_2]{i_2,v} q'$ is a transition in Q. It follows that $q \xrightarrow[i_2,v]{u,i_1,o_1,o_2} q'$ is a transition in $Q^\perp$. Since P and $Q^\perp$ are compatible by assumption and (p,q) is a state in $P \parallel R$, there must be a transition $p \xrightarrow[u,o_1]{i_1,v} p'$ in P, and therefore there must be a transition $(p,(p,q)) \xrightarrow[u,i_1,o_1,o_2]{i_2,v} (p',(p',q'))$ in $P \parallel R$, and $((p',(p',q')),q') \in \rho$ by the definition of ρ.

($\Rightarrow$) Suppose a solution to $P \parallel R \preceq Q$ exists. Let ρ be an alternating simulation from $P \parallel R$ to Q such that $((s_P^0,s_R^0),s_Q^0) \in \rho$. We use R and ρ to construct a winning input strategy in $P \otimes Q^\perp$. It is easy to see that for states (p,r) in $P \parallel R$ and q in Q, if $((p,r),q) \in \rho$ then (p,q) is locally compatible in $P \otimes Q^\perp$. The winning input strategy $\pi^I(p,q)$ in $P \otimes Q^\perp$ is given by an input move (i_1,o_2) such that there exist a state r and values v,u,o_1,o_2 satisfying $((p,r),q) \in \rho$, $(i_2,v) \in \mathcal{A}^I_Q(q)$, $(u,i_1,o_1,o_2) \in \mathcal{A}^O_{P\parallel R}(p,r)$, $(p,r) \xrightarrow[u,i_1,o_1,o_2]{i_2,v} (p',r')$ and $q \xrightarrow[u,i_1,o_1,o_2]{i_2,v} (p',r')$; otherwise, $\pi^I(p,q)$ is arbitrary. To show that π^I is winning, we prove by induction on the definition of Reach$(P \otimes Q^\perp, \pi^I)$ that Reach$(P \otimes Q^\perp, \pi^I) \cap \text{Incomp}(P,Q^\perp) = \emptyset$. $\square$

Theorem 7.3. When the condition stated in Theorem 7.2 is satisfied, the most general solution to $P \parallel R \preceq Q$ exists and is given by $R = (P \parallel Q^\perp)^\perp$.

Proof. In the proof of Theorem 7.2 (If part) we have already shown that $R = (P \parallel Q^\perp)^\perp$ is a solution. Suppose T is any solution to $P \parallel R \preceq Q$. We construct an alternating simulation ν from T to $(P \parallel Q)^\perp$ as follows. By assumption, there exists an alternating simulation ρ from $P \parallel T$ to Q. Define $\nu = \{(t,(p,q)) \mid ((p,t),q) \in \rho\}$. Clearly $(s_T^0,(s_P^0,s_Q^0)) \in \nu$, since $((s_P^0,s_T^0),s_Q^0) \in \rho$. Now suppose $(t,(p,q)) \in \nu$ i.e., $((p,t),q) \in \rho$, $(u,v,i_2,o_1) \in \mathcal{A}^I_{(P\parallel Q^\perp)^\perp}((p,q))$ and $(i_1,o_2) \in \mathcal{A}^O_T(t)$. Let $t \xrightarrow[i_1,o_2]{u,v,i_2,o_1} t'$ be a transition in T and $(p,q) \xrightarrow[i_1,o_2]{u,v,i_2,o_1} (p',q')$ a transition in $(P \parallel Q^\perp)^\perp$. This implies that $p \xrightarrow[u,o_1]{i_1,v} p'$ is in P and $q \xrightarrow[u.i_1,o_1,o_2]{i_2,v} q'$ is in Q. Hence $(p,t) \xrightarrow[i_1,u,o_1,o_2]{i_2,v} (p',t')$ is in $P \parallel T$. Since $((p,t),q) \in \rho$ by assumption, it follows that $((p',t'),q') \in \rho$, i.e., $(t',(p',q')) \in \nu$. $\square$

The complexity of computing $(P \parallel Q^\perp)^\perp$ is again $O(|P||Q|)$ using the standard iterative refinement technique as in the asynchronous case.

8. Converter Synthesis

In this section we show how the SIA framework and the interface synthesis procedure described in Section 7 can be used to synthesise a protocol converter for two IP blocks that have incompatible protocols of interaction. Our work is inspired by Passerone *et al.* in [PdAHSV02], and should be seen as both a generalisation and a simplification of that work.

Let P_1 and P_2 be the SIA describing two mismatched protocols, such as a sender using a handshake and a receiver using a serial protocol, as in [PdAHSV02]. We assume that P_1 and P_2 have disjoint alphabets. Just as in network

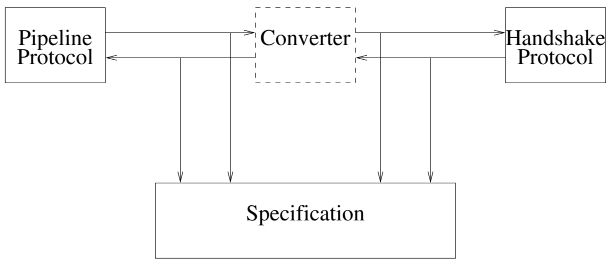


Fig. 6. Block diagram of protocols, specification and converter

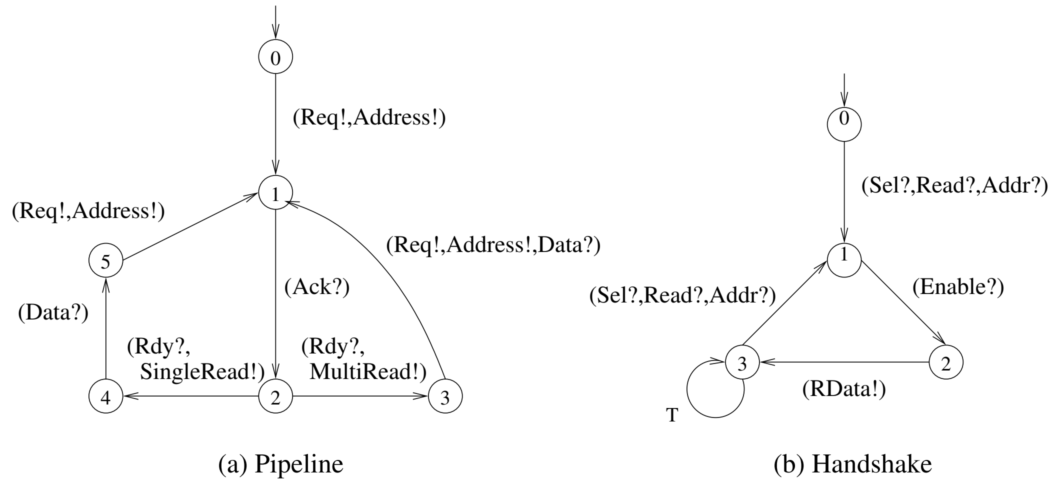


Fig. 7. Two Mismatched Protocols

protocol conversion in Section 5, it is the responsibility of the designer to specify the exact relationship between the two, through another SIA S, the *specification*. To summarise, the specification S expresses the causal relationships between the actions of P_1 and P_2. In addition, the specification S captures the capabilities of the converter in terms of storage and retransmission capabilities. See the example below and [PdAHSV02] for more details.

We think of the specification as accepting inputs from the two protocols as well as the converter, as shown in Fig. 6. This is in contrast to the network protocol conversion discussed in Section 5, where we took the dual 'output' view. Intuitively, the goal of the converter is to meet the specification, while satisfying the input assumptions of the two protocols. Moreover, the converter can control only the inputs to the protocols and not their outputs. The converter can then be obtained by using our interface synthesis procedure as in Section 5. Let $P = P_1 \parallel P_2$ be the parallel composition of the two mismatched protocols, which is well formed, since we assume that the input and output actions of P_1 and P_2 are disjoint. Then a converter C, if it exists, is the (most general) solution for C to the interface synthesis problem instance $P \parallel C \preceq S^{\perp}$, where S is the specification. The meaning of this relation is that $P \parallel C$ is a safe environment for S, that is, the composite $P \parallel C$ of protocols plus converter will not give rise to an incompatibility when combined with S.

We illustrate the converter synthesis problem for IP blocks via an example adapted from [DRS04a]. We adopt the following convention in drawing synchronous interfaces for IP blocks. When only boolean valued signals are involved, as is the case in this example, values not mentioned in a transition are don't cares (either low or high). Sometimes we indicate a don't care explicitly by a signal `T`. Fig. 7 illustrates the synchronous interfaces for two mismatched protocols. Fig. 7(a) is a protocol called Pipeline, with input variables $[\mathtt{Ack}, \mathtt{Rdy}, \mathtt{Data}]$ and output variables $[\mathtt{Req}, \mathtt{Address}, \mathtt{SingleRead}, \mathtt{MultiRead}]$, that requests data from specified addresses in memory. When the protocol wants to read some data it raises the line `Req!` to high, and places the value of the memory address on `Address!`, and waits for an acknowledgement `Ack?` in the following clock cycle. In this simplified example, we ignore all data values such as addresses, and consider only boolean control values. In the next clock cycle the protocol checks that the signal `Rdy?` is high. If a single read is desired the protocol reads the input `Data?` and completes the transaction. If a sequence of reads is to be performed, the protocol pipelines the address phase of the next transfer with the current

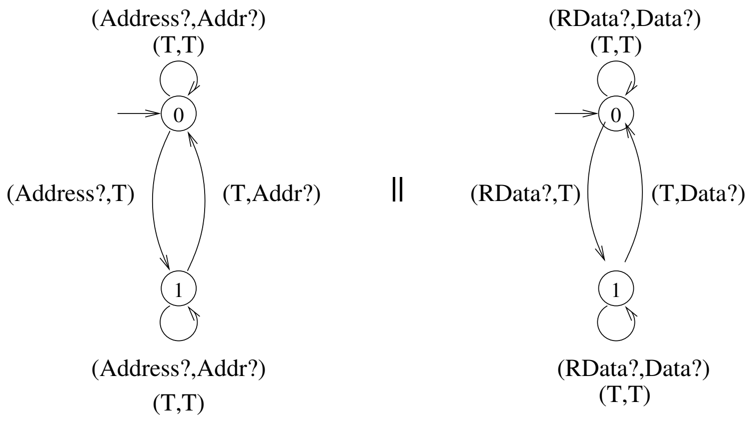


Fig. 8. Specification of Converter

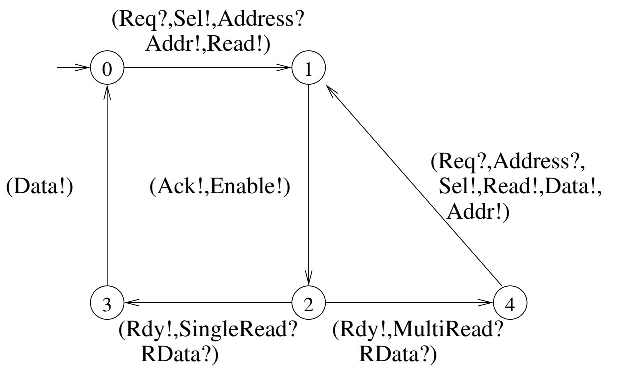


Fig. 9. Converter for the two protocols

data phase. The protocol stops in state 5 after completing a finite sequence of transfers until it is ready to begin a fresh read request. Note that in state 2, the same input `Rdy?` can lead to two distinct states, but the output values associated with the transitions are distinct – `SingleRead!` in one, and `MultiRead!` in the other, so the observable nondeterminism property is satisfied.

Fig. 7(b) is an interface, with input variables [`Sel`, `Read`, `Addr`, `Enable`] and output variable [`RData`], that performs reads from memory addresses, but it uses a handshake protocol. When it is selected for a read transfer by raising its input lines `Sel?` and `Read?`, it reads the address from `Addr?`. If the signal `Enable?` is high in the next clock cycle, it writes the data on the output line `RData!`, and is ready to handle a new read request, while waiting in state 3. The protocols in Fig. 7 are mismatched and will not work properly unless there is a converter which mediates between the two.

Now, we need to specify what the converter is allowed and not allowed to do. We require that the system as a whole (the two protocols along with the converter) satisfies the interface described by Fig. 8. This specification interface is obtained as the parallel composition of two interfaces. The one on the left specifies that the converter can send a `Addr!` signal to Handshake only after receiving a corresponding `Address?` signal from Pipeline. The signal cannot be sent speculatively, but can be stored in memory and sent at a later instant. Similarly the interface on the right specifies that the converter can send a `Data!` to Pipeline, only after a corresponding `RData?` signal has been received from Handshake. Note that every action in Fig. 8 is of type input, in conformance with Fig. 6.

The correct converter for the two protocols, as synthesised by our method, is shown is Fig. 9. Our solution to the synthesis problem is identical to the one in [PdAHSV02] for the special cases considered there for a pair of protocols, one of which is the sender, and the other the receiver. Our solution is more generals, as it applies to Mealy machines with both inputs and outputs. In addition, our closed form solution is more algebraic and hides the details of the game solution involved in the composition of synchronous interfaces.

9. Conclusion and Future Work

We have presented the synthesis problem and its solution for both asynchronous and synchronous interface automata, a game based formalism for reasoning about composition and refinement of components. It is aesthetically pleasing that the solutions for the asynchronous and synchronous versions have the same form $R = (P \parallel Q^{\perp})^{\perp}$. We have already noted the formal resemblance of our solution to solutions to the submodule construction problem in [vB02], the language equation problem in [YVB^{+}01, YVB^{+}02] , and the rectification problem in [BDWM93].

Recently, agent algebras [BPSV03, Pas04] have been proposed as a general framework for modelling a wide variety of concurrent systems. They are intended as a model for reasoning about compositionality and refinement of concurrent systems, and are an extension of Dill's trace theory [Dil89]. An agent algebra is an abstract algebraic structure with three operations – parallel composition, projection and renaming – which must satisfy certain axioms. The domain of the algebra is intended to represent a set of processes or agents. New agent algebras can be built from old ones by using the familiar constructions of direct product, direct sum and subalgebra. Agent algebras may be equipped with a preorder on the agents that represents the refinement relation. To support compositional proofs of refinement, agent algebras must satisfy a monotonicity of each operation with respect to the refinement order. The refinement preorder can often be characterised as a *conformance relation*, a relation that holds between two agents when one can be substituted for another in all possible contexts. Our synthesis problem appears in the context of agent algebras as the "problem of synthesising a local specification subject to a context". The works cited above provides sufficient conditions for characterising all controllers that satisfy a specification when composed with a plant. These conditions resemble our Theorems 3.3, 3.4, 7.2 and 7.3 Based on this, it appears that both our asynchronous and synchronous interface automata are interesting instances of agent algebras.

As future work, we would like to relax some restrictions we have put on the synchronous interface model, such as the requirement of observable nondeterminism. Weaker notions of determinism, such as *weak determinism* [DRS04b] could be investigated. Other possibilities would be to include the effect of hiding internal signals and including fairness specifications to both asynchronous and synchronous interfaces. An important advance would be to include asynchrony and synchrony within the same framework for modelling SoC designs, as the complexity of today's circuits requires locally clocked components that communicate via asynchronous signals. The work on agent algebras related to semantic foundations for heterogeneous systems (see [Pas04]) has a similar goal, and it will be interesting to investigate the connections between the two.

Acknowledgement We are grateful to the anonymous referees for suggesting many improvements in the presentation. Thanks are due to David Benson for his discussions and feedback on the work. We thank Ashish Kumar Sinha and Nikhil Gupta for implementing the asynchronous and synchronous versions of our synthesis procedure.

References

[Abr97] S. Abramsky. Semantics of interaction: an introduction to game semantics. In *Proceedings of the 1996 CLiCS Summer School*, pages 1–31. Cambridge University Press, 1997.

[AHKV98] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *CONCUR 98: Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 163–178. Springer-Verlag, 1998.

[BB91] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Procedings of the IEEE*, 79(9), 1991.

[BDWM93] J. R. Burch, D. Dill, E. Wolf, and G. De Micheli. Modeling hierarchical combinational circuits. In Michael Lightner, editor, *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 612–617. IEEE Computer Society Press, 1993.

[Bha05] Purandar Bhaduri. Synthesis of interface automata. In *Third International Symposium on Automated Technology for Verification and Analysis (ATVA 2005)*, volume 3707 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2005.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[Bla92] Andreas Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56:183–220, 1992. Special Volume dedicated to the memory of John Myhill.

[BPSV03] J. R. Burch, R. Passerone, and Alberto L. Sangiovanni-Vincentelli. Notes on agent algebras. Technical Report UCB/ERL M03/38, EECS Department, University of California, Berkeley, 2003.

[BR06] Purandar Bhaduri and S. Ramesh. Synthesis of synchronous interfaces. In *Sixth International Conference on Application of Concurrency to System Design, ACSD 2006*, pages 208–216. IEEE Computer Society, 2006.

[CdAHM02] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *CAV 02: Computer-Aided Verification*, Lecture Notes in Computer Science 2404, pages 414–427. Springer-Verlag, 2002.

[CL90] K. L. Calvert and S. S. Lam. Formal methods for protocol conversion. *IEEE Journal Selected Areas in Communications*, 8(1):127–142, January 1990.

[dA03] Luca de Alfaro. Game models for open systems. In *Proceedings of the International Symposium on Verification (Theory in Practice)*, volume 2772 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[dAH01] L. de Alfaro and T.A. Henzinger. Interface automata. In *Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[dAHM00] L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. The control of synchronous systems. In *CONCUR 00: Concurrency Theory*, Lecture Notes in Computer Science 1877, pages 458–473. Springer-Verlag, 2000.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[DRS04a] Vijay D'Silva, S. Ramesh, and Arcot Sowmya. Bridge over troubled wrappers: Automated interface synthesis. In *VLSI Design*, pages 189–194. IEEE Computer Society, 2004.

[DRS04b] Vijay D'Silva, S. Ramesh, and Arcot Sowmya. Synchronous protocol automata: A framework for modelling and verification of SoC communication architectures. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004)*, pages 390–395, 2004.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[HU99] E. Haghverdi and H. Ural. Submodule construction from concurrent system specifications. *Information and Software Technology*, 41(8):499–506, June 1999.

[KNM97] Ratnesh Kumar, Sudhir Nelvagal, and Steven I. Marcus. A discrete event systems approach for protocol conversion. *Discrete Event Dynamic Systems*, 7(3):295–315, June 1997.

[Lam88] S. S. Lam. Protocol conversion. *IEEE Transactions on Software Engineering*, 14(3):353–362, March 1988.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, 10–12 August 1987.

[LX90] Kim Guldstrand Larsen and Liu Xinxin. Equation solving using modal transition systems. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 108–117. IEEE Computer Society Press, June 1990.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. In *Theoretical Aspects of Computer Science*, volume 900 of *LNCS*, pages 229–242. Springer-Verlag, 1995.

[MT98] P. Madhusudan and P. S. Thiagarajan. Controllers for discrete event systems via morphisms. In *CONCUR '98: Concurrency Theory, 9th International Conference*, volume 1466 of *Lecture Notes in Computer Science*, pages 18–33. Springer-Verlag, 1998.

[MvB83] Philip Merlin and Gregor von Bochmann. On the construction of submodule specifications and communication protocols. *j-toplas*, 5(1):1–25, January 1983.

[Par89] J. Parrow. Submodule construction as equation solving in CCS. *Theoretical Computer Science*, 68:175–202, 1989.

[Pas04] R. Passerone. *Semantic Foundations for Heterogeneous Systems*. PhD thesis, EECS Department, University of California, Berkeley, 2004.

[PdAHSV02] R. Passerone, L. de Alfaro, T.A. Henzinger, and A. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the International Conference on Computer-Aided Design*, pages 132–139. IEEE Computer Society Press, 2002.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89. Proceedings of the sixteenth annual ACM symposium on Principles of programming languages, January 11–13, 1989, Austin, TX*, pages 179–190, New York, NY, USA, 1989. ACM Press.

[RW89] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77, 1:81–98, 1989.

[Shi89] M. W. Shields. Implicit system specification and the interface equation. *Computer Journal*, 32(5):399–412, 1989.

[Tab04] Paulo Tabuada. Open maps, alternating simulations and control synthesis. In *CONCUR '04*, number 3170 in Lecture Notes in Computer Science, pages 466–480. Springer-Verlag, 2004.

[Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *12th Annual Symposium on Theoretical Aspects of Computer Science*, volume 900 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1995.

[vB02] Gregor von Bochmann. Submodule construction for specifications with input assumptions and output guarantees. In Doron Peled and Moshe Y. Vardi, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2002*, volume 2529 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2002.

[YVB$^+$01] Nina Yevtushenko, Tiziano Villa, Robert K. Brayton, Alex Petrenko, and Alberto Sangiovanni-Vincentelli. Solution of parallel language equations for logic synthesis. In *Proceedings of the 2001 International Conference on Computer-Aided Design (ICCAD-01)*, pages 103–111. IEEE Computer Society, 2001.

[YVB$^+$02] Nina Yevtushenko, Tiziano Villa, Robert K. Brayton, Alex Petrenko, and Alberto Sangiovanni-Vincentelli. Solution of synchronous language equations for logic synthesis. In *Proceedings of the 4th Conference on Computer-Aided Technologies in Applied Mathematics*, pages 132–137, 2002.