Synthesis of Synchronous Interfaces

Purandar Bhaduri
Dept. of Computer Science & Engineering
Indian Institute of Technology Guwahati
Guwahati 781039, India
pbhaduri@iitg.ernet.in

S. Ramesh
India Science Lab, GM R&D Centre
Creator, International Tech Park
Whitefield Road, Bangalore 560066, India
ramesh.s@gm.com

Abstract

Reuse of IP blocks has been advocated as a means to conquer the complexity of today's system-on-chip (SoC) designs. Component integration and verification in such systems is a cumbersome and time consuming process. We present synchronous interface automata (SIA) as a framework for modelling communication aspects of IP blocks, to serve as a unifying model in the top-down refinement, synthesis and verification stages of the design process. We show how to formally specify component composition and protocol compatibility in our model, and how we can apply the model to the problem of synthesising converters for incompatible protocols of interaction between IP blocks. Our model is based on the game theoretic framework of interface automata, suitably adapted for practical modelling of IP blocks.

1. Introduction

The "design productivity gap", which refers to the apparent mismatch between our ability to design complex integrated circuits and what is technologically possible, is a serious impediment in handling the complexity of today's system-on-chips (SoC's) and embedded systems. To manage design complexity and shorten design cycles, design methodologies that raise the level of abstraction from the RTL level to the system level have been proposed. These high level modelling and validation methods have advocated greater reuse of existing intellectual property (IP) blocks. However, this raises the fundamental problem of design correctness. In view of the complexity of system-level designs, it is important to guarantee that composition of IP blocks is correct by design. In practice, this is a difficult task, because components often come from different manufacturers and are designed at different levels of abstraction using different protocols of interactions.

Automated design reuse by composing IP blocks has to address the problem of their incompatibility due to mismatches in protocols of interaction. To meet the challenge of incompatibility of protocols when composing IP blocks, protocol converters have to be built to ensure correctness. Various approaches have been proposed for constructing a converter that resolves protocol mismatches (see [4, 15, 17–20]). Most of these models are quite simplistic and informal, and no clear formalisation of the problem of protocol mismatch exists.

We propose the *synchronous interface automata* (SIA) model as a formalism for specifying the protocol behaviour of IP blocks. The SIA model is suitable for deriving detailed implementation models from high level abstract specifications, for verifying compatibility of IP blocks, and for automatically synthesising converters when there is a mismatch in protocols. We show how the SIA model gives a formal foundation to the problem of converter synthesis, by showing how this problem can be placed in the wider framework of the following *interface synthesis* problem. We are given an interface P for a known component of the system, and the interface Q for the system as a whole. We have to find the most general interface R, which combined with P is a refinement of Q, symbolically $P \parallel R \preceq Q$. This is a central problem in component based top down design of a system, and has been investigated previously in other contexts [3, 5, 22, 23]. We show that in our SIA framework the solution is given by $R = (P \parallel Q^{\perp})^{\perp}$, where $P^{\perp}$ is the interface identical to P, except the roles of input and output are interchanged. From this general framework we are able to derive a solution to the specific problem of converter synthesis for mismatched protocols. The formalism for converter synthesis closest to ours is the game theoretic approach proposed by Passerone *et al.* in [17]. We show that our framework is more general than that of [17] – indeed the latter formulation is just a special case of the former.

Interface automata are a game based formulation of interfaces – see [9] and [8] for the details. The original interface automata formalism was proposed to model the behavioural interface of asynchronously interacting software modules. A synchronous version, referred to as Moore interface, was proposed in [7] to model interactions between components typical in hardware. In this paper we give a new definition of synchronous interfaces that is suitable for modelling the protocol behaviour of IP blocks. The main differences between our SIA model and the Moore interfaces of [7], are that, 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 [2]). The price to pay is the difficulty in composition due to the possibility of causality cycles. We come back to this point later.

The SIA model essentially defines Mealy automata with explicit input assumptions and output guarantees. Since an IP block interacts with other blocks to realise a given functionality, reasoning about its correctness requires assumptions about its environment. A block behaves correctly, i.e., meets its output guarantees, only when its input sequence satisfies its input constraints. The interplay of input assumptions and output guarantees gives rise to a game view of SIA composition. The game is between players Input and Output, where the role of Input is to provide the right inputs in a given state so that no incompatibility can arise. Two SIA are composable only if there is a winning input strategy, which amounts to the *existence* of an environment which can make both of them work together. This is in contrast to *input enabled models*, such as the I/O automata framework [14], where it is required that no input action can ever be refused in any state. Effectively, the composition operation on SIA composes input assumptions and output guarantees of two SIA, and any unmatched assumptions are propagated to the environment – see [8, 9] for the detailed motivation on the use of games for composition and refinement for components.

The main technical contribution of this paper is in proposing an algebraic framework for composition and refinement of SIA, and its use in solving the synthesis problem described above. We demonstrate the significance of the problem by showing how it can be used to solve the converter synthesis problem between mismatched protocols of interaction of IP blocks.

2. Synchronous Interface Automata

We fix some notation and conventions first. An *I/O-signature* is a pair $[\vec{\mathbf{I}}\ ;\ \vec{\mathbf{O}}]$, where $\vec{\mathbf{I}} = \mathbf{I}_1,\ldots,\mathbf{I}_n$ and $\vec{\mathbf{O}} = \mathbf{O}_1,\ldots,\mathbf{O}_m$ 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 Figure 1. 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 $\vec{I} = I_1 \times \ldots \times I_n$ interpreting all the input variables as the *input space* or *input alphabet*, and $\vec{O} = O_1 \times \ldots \times O_m$ as the *output space* or *output alphabet*.

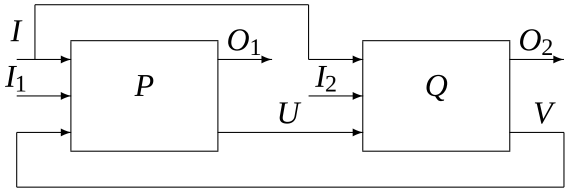


Figure 1. Block diagram for $P \otimes Q$

Definition 1 A *synchronous interface automaton* (SIA) P with I/O signature $[\vec{\mathbf{I}}\ ;\ \vec{\mathbf{O}}]$ is a tuple $(S_P, S_P^0, A_P^I, A_P^O, \delta_P)$ 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.
- $A_P^I = I_1 \times I_2 \times \ldots I_n$ and $A_P^O = O_1 \times O_2 \times \ldots O_m$ are the *input* and *output alphabets*.
- $\delta_P : S_P \times A_P^I \times A_P^O \to S_P$ 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.

The SIA 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.

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 $\delta(p,i,o) = p'$ 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 $A_P^I(p) = \{i \mid p \xrightarrow[o]{i} p' \text{ for some } o \text{ and } p'\}$ and $A_P^O(p) = \{o \mid p \xrightarrow[o]{i} p' \text{ for some } i \text{ and } p'\}$ respectively.

Definition 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.

When composing two SIA, the available inputs may need to be cut down, in order to avoid reaching incompatible states. This leads to the definition of an input strategy, a predetermined way of choosing the input at every state. Note that our strategies are *deterministic* (at every state there is exactly one choice of input action) and *memoryless* (the choice of input action depends only on the current state and not on the past history).

Definition 3 An *input strategy* for P is a map $\pi^I : S_P \to A_P^I$.

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 $o \in A_P^O$, $\delta_P(s, \pi^I(s), o) \in \text{Reach}(P, \pi^I)$.

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 $[\vec{\mathbf{I}}\ ;\ \vec{\mathbf{O}}]$ and $[\vec{\mathbf{I}}'\ ;\ \vec{\mathbf{O}}']$ are *composable* if they don't share an output variable, i.e., $\vec{\mathbf{O}} \cap \vec{\mathbf{O}}' = \emptyset$.

To define composition, we first define the product of two synchronous interfaces, just as in the asynchronous case in [8, 9]. In the definition below, we assume that $\vec{\mathbf{I}}_1 \cap \vec{\mathbf{O}}_2 = \emptyset$ and $\vec{\mathbf{O}}_1 \cap \vec{\mathbf{I}}_2 = \emptyset$. In the signatures of the composable SIA P and Q, $\vec{\mathbf{I}}$ is the list of shared input variables. The output variables $\vec{\mathbf{U}}$ of P and $\vec{\mathbf{V}}$ of Q appear as input variables of the other automaton, as in Figure 1.

Definition 4 Let P and Q be two composable SIA with I/O signatures $[\vec{\mathbf{I}}, \vec{\mathbf{I}}_1, \vec{\mathbf{V}}\ ;\ \vec{\mathbf{U}}, \vec{\mathbf{O}}_1]$ and $[\vec{\mathbf{I}}, \vec{\mathbf{I}}_2, \vec{\mathbf{U}}\ ;\ \vec{\mathbf{V}}, \vec{\mathbf{O}}_2]$. Thus $A_P^I = I \times I_1 \times V$ and $A_P^O = U \times O_1$ are the input and output alphabets of P, and $A_Q^I = I \times I_2 \times U$ and $A_Q^O = V \times O_2$ the respective alphabets of Q. Then the product $P \otimes Q$ of SIA P and Q with I/O signature $[\vec{\mathbf{I}}, \vec{\mathbf{I}}_1, \vec{\mathbf{I}}_2\ ;\ \vec{\mathbf{U}}, \vec{\mathbf{V}}, \vec{\mathbf{O}}_1, \vec{\mathbf{O}}_2]$ is defined by the tuple $(S_{P\otimes Q}, S_{P\otimes Q}^0, A_{P\otimes Q}^I, A_{P\otimes Q}^O, \delta_{P\otimes Q})$ where

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

The block diagram for the product $P \otimes Q$ is illustrated in Figure 1. Note that the above definition and Figure 1 describe the most general situation. For instance, if P and Q do not share any input signal then $\vec{\mathbf{I}}$ is the empty tuple, so P has signature $[\vec{\mathbf{I}}_1, \vec{\mathbf{V}}\ ;\ \vec{\mathbf{U}}, \vec{\mathbf{O}}_1]$ and Q has signature $[\vec{\mathbf{I}}_2, \vec{\mathbf{U}}\ ;\ \vec{\mathbf{V}}, \vec{\mathbf{O}}_2]$.

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

The set $\text{Incomp}(P, Q)$ of *locally incompatible states* of P and Q is the set of states in $S_P \times S_Q$ which are not locally compatible.

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 a state. The states from which this is possible are called compatible.

Definition 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 $\mathrm{Reach}(P \otimes Q, (p,q), \pi^I)$ does not contain a locally incompatible state of $P \otimes Q$. We write $\mathrm{Comp}(P,Q)$ 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 7 The composition $P \parallel Q$ of two SIA is defined by restricting the product $P \otimes Q$ to the set of compatible states:

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

Thus 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 $P \otimes Q$: an input strategy which avoids all locally incompatible states. The calculation of winning strategy in such safety games, if one exists, by using the *controllable predecessors* of a set of states U and iterative refinement is standard [21].

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 [10] for details.

The game view of interfaces leads to a new notion of refinement called *alternating refinement* [1]. 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 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. $A^I_Q(t) \subseteq A^I_P(s)$;
2. $A^O_P(s) \subseteq A^O_Q(t)$;
3. $(\delta_P(s,a), \delta_Q(t,a)) \in \rho$ for all $a \in A^I_Q(t) \times A^O_P(s)$,

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:

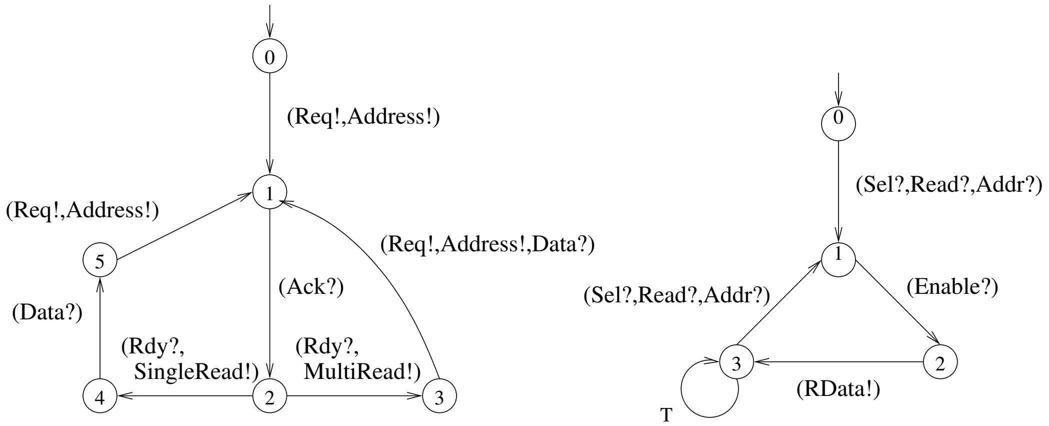
1. $A^I_Q \subseteq A^I_P$;
2. $A^O_P \subseteq A^O_Q$;
3. there is an alternating simulation ρ from P to Q, such that $(s^0,t^0) \in \rho$ for some $s^0 \in S^0_P$ and $t^0 \in S^0_Q$.

3. Synthesis of Synchronous Interfaces

For top-down design of SoC's, we would like to synthesise a component R that combined with a known component P realises the specification Q. In other words, we are interested in 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 general solution we mean, a solution U, such that for any solution V, it is the case that $V \preceq U$, which means V can be substituted for U in any context without leading to protocol mismatches. In this section we prove that the most general solution to $P \parallel R \preceq Q$ is given by $R = (P \parallel Q^\perp)^\perp$ and a solution exists iff P and $Q^\perp$ are compatible. 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 $P^\perp$.

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: $A^I_P \subseteq A^I_Q \cup A^O_Q$ and $A^O_P \subseteq A^O_Q$. 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:

$$\begin{aligned}
P &: [\vec{\mathbf{I}}_1, \vec{\mathbf{V}} \,;\, \vec{\mathbf{U}}, \vec{\mathbf{O}}_1] \\
Q &: [\vec{\mathbf{I}}_2, \vec{\mathbf{V}} \,;\, \vec{\mathbf{U}}, \vec{\mathbf{I}}_1, \vec{\mathbf{O}}_1, \vec{\mathbf{O}}_2] \\
Q^\perp &: [\vec{\mathbf{U}}, \vec{\mathbf{I}}_1, \vec{\mathbf{O}}_1, \vec{\mathbf{O}}_2 \,;\, \vec{\mathbf{I}}_2, \vec{\mathbf{V}}] \\
P \otimes Q^\perp &: [\vec{\mathbf{I}}_1, \vec{\mathbf{O}}_2 \,;\, \vec{\mathbf{U}}, \vec{\mathbf{V}}, \vec{\mathbf{I}}_2, \vec{\mathbf{O}}_1] \\
(P \parallel Q^\perp)^\perp &: [\vec{\mathbf{U}}, \vec{\mathbf{V}}, \vec{\mathbf{I}}_2, \vec{\mathbf{O}}_1 \,;\, \vec{\mathbf{I}}_1, \vec{\mathbf{O}}_2] \\
P \otimes (P \parallel Q^\perp)^\perp &: [\vec{\mathbf{I}}_2, \vec{\mathbf{V}} \,;\, \vec{\mathbf{U}}, \vec{\mathbf{I}}_1, \vec{\mathbf{O}}_1, \vec{\mathbf{O}}_2]
\end{aligned}$$



(a) Pipeline

(b) Handshake

Figure 2. Two Mismatched Protocols

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 1 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 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 $\mathrm{Reach}(P \otimes (P \parallel Q^{\perp})^{\perp}, \pi^I)$ such that $(p,(p,q)) \in \mathrm{Incomp}(P,(P \parallel Q^{\perp})^{\perp})$. It follows from Definition 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 $(p,q) \xrightarrow[i_1,o_2]{u,v,i_2,o_1} (p',q')$ 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 1 *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 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, $(u,i_1,o_1,o_2) \in A^O_{P\parallel R}((p,(p,q)))$, 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 $((p,r),q) \in \rho$, $(i_2,v) \in A_Q^I(q)$, $(u,i_1,o_1,o_2) \in A_{P\parallel R}^O(p,r)$, $(p,r) \xrightarrow[u,i_1,o_1,o_2]{i_2,v} (p',r')$ 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 $\mathrm{Reach}(P \otimes Q^{\perp}, \pi^I)$ that $\mathrm{Reach}(P \otimes Q^{\perp}, \pi^I) \cap \mathrm{Incomp}(P, Q^{\perp}) = \emptyset$. □

Theorem 2 *When the condition stated in Theorem 1 is satisfied, the most general solution to $P \parallel R \preceq Q$ is $R = (P \parallel Q^{\perp})^{\perp}$.*

Proof In the proof of Theorem 1 (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., $((p,t),q) \in \rho$, $(u,v,i_2,o_1) \in A^I_{(P\parallel Q^{\perp})^{\perp}}((p,q))$ and $(i_1,o_2) \in A_T^O(t)$. 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., $(t',(p',q')) \in \nu$. □

4. Converter Synthesis

In this section we show how the SIA framework and the interface synthesis procedure described in Section 3 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 [17], 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 [17]. We assume that P_1 and P_2 have disjoint alphabets. It's the responsibility of the designer to specify the exact relationship between the two, through another SIA S, the *specification*. In more detail, the specification S expresses the causal relationships between the actions of P_1 and P_2, such as between sending and receiving of a data packet. In addition, the specification S captures the capabilities the designer wishes to endow the converter with – how much memory it can use to store packets before transmitting them, whether it can lose or duplicate data packets, and so on. See the example below and [17] for more details.

We think of the specification as accepting inputs from the two protocols as well as the converter, as shown in Figure 4. 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 be obtained by using our interface synthesis procedure be as follows. 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. Let S be a specification expressing the causal relationship between the two alphabets and what the converter is allowed and not allowed to do. Then a converter C, if it exists, is the (most general) solution for C to the interface synthesis problem instance $P \parallel C \preceq S^{\perp}$. 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.

The solution we obtain is an extension to the more restricted game theoretic solution for C in [17]. In [17] the converter synthesis problem investigated is between a sender and a receiver protocol. The problem is solved as a game between two players, the protocols and the specification on one side, and the converter on the other. The objective of the converter is to read outputs from the sender and provide inputs to the receiver in such a way that the protocols and the specification are satisfied. A winning strategy for the game leads to a correct converter. If we unfold the definition of parallel composition and alternating refinement, we have an identical winning condition to our synthesis problem. Our solution is more general because, we do not place the restriction that one protocol is the sender and the other the receiver. Moreover, we don't have to set up the game manually as in [17], which seems to involve considerable ingenuity. Instead, the game formulation follows directly from the general framework presented in this paper.

We illustrate the converter synthesis problem for IP blocks via an example adapted from [12]. 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

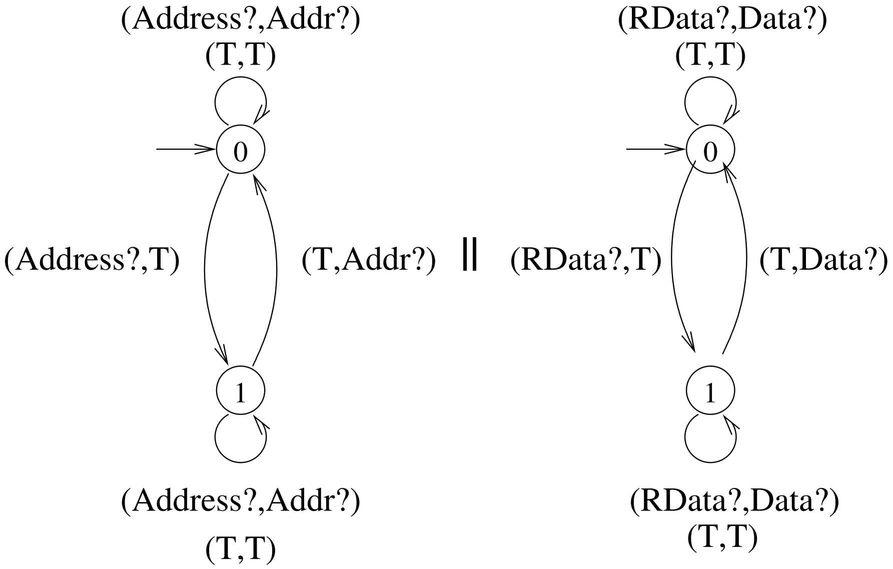


Figure 3. Specification of Converter

a transition are don't cares (either low or high). Sometimes we indicate a don't care explicitly by a signal T. Figure 2 illustrates the synchronous interfaces for two mismatched protocols. Figure 2(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 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.

Figure 2(b) is an interface, with input variables $[\mathtt{Sel},\mathtt{Read},\mathtt{Addr},\mathtt{Enable}]$ and output variable $[\mathtt{RData}]$, 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 Figure 2 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 Figure 3. 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 in 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 inFigure 3 is of type input.

The correct converter for the two protocols, as synthesised by our method, is shown is Figure 5.

5. Conclusion and Related Work

We have presented synchronous interface automata, a game based formalism for reasoning about composition and refinement of synchronous hardware components, such as IP blocks in SoC designs. The asynchronous version of the synthesis problem considered here was solved in [3], using the interface automata model of [9]. What is remarkable is, the solutions for the asynchronous and synchronous versions have the same form, namely the most general solution to $P \parallel R \preceq Q$ is $R = (P \parallel Q^{\perp})^{\perp}$.

Our result has a formal resemblance to the language

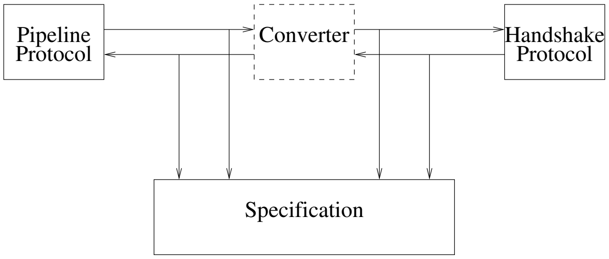


Figure 4. Block diagram of protocols, specification and converter

equation posed in [22, 23]. In their framework, 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.

There is also a striking similarity between our work and the the solution to the *rectification problem* given in [5], using Dill's trace theory [11]. The results in [5] are applicable to combinational circuits i.e., interfaces without any state. In [5] P, Q, R are combinational circuits modelled as I/O relations and $||$ a composition relation similar to SIA composition; $(\_)^{\perp}$ is the *mirror* function that swaps inputs and outputs and $\preceq$ is the *conformance relation*. In contrast to this work, our results are applicable to stateful interfaces in the form of Mealy machines. Further, the timing model used in trace theory is asynchronous whereas we use a synchronous timing model. The notion of alternating simulation relation is also different from the conformance relation. Recent work on *agent algebras* [6, 16] formalises the notions of composition and conformance in an abstract algebraic framework, and makes use of the mirror relation 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 synchronous synthesis framework.

As we have mentioned in the previous section, our work is inspired by Passerone *et al.* in [17]. Our solution to the synthesis problem is identical to the one in [17] 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 general, 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. Our work can be seen as sharing the same algebraic flavour as the the work on agent algebras.

As future work, we would like to relax some restrictions we have put on the interface model, such as the requirement of observable nondeterminism. Weaker notions of determinism, such as *weak determinism* [13] could be investigated. Other possibilities would be to include the effect of hiding internal signals and including fairness specifications. 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 [16]) has a similar goal, and it will be interesting to investigate the connections between the two.

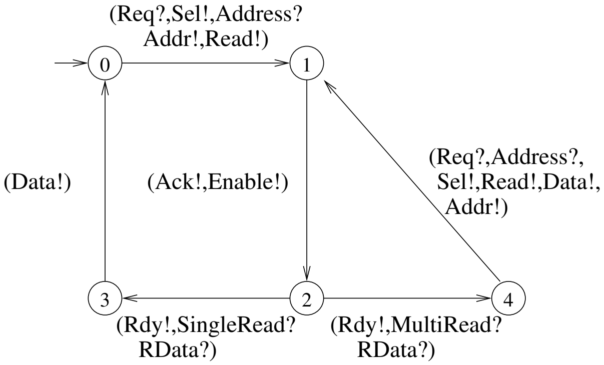

(Req?,Sel!,Address? Addr!,Read!)
0
1
(Data!)
(Ack!,Enable!)
(Req?,Address?, Sel!,Read!,Data!, Addr!)
3
2
4
(Rdy!,SingleRead? RData?)
(Rdy!,MultiRead? RData?)

Figure 5. Converter for the two protocols

Acknowledgements We thank David Benson for his comments on an earlier draft and anonymous referees for their suggestions on improving the presentation of the paper.

References

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

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

[3] P. 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.

[4] G. Borriello, L. Lavagno, and R. B. Ortega. Interface synthesis: A vertical slice from digital logic to software components. In *International Conference on Computer Aided Design (ICCAD-98)*, pages 693–695. ACM Press, 1998.

[5] J. R. Burch, D. Dill, E. Wolf, and G. D. Micheli. Modeling hierarchical combinational circuits. In M. Lightner, editor, *Proceedings of the IEEE/ACM International Conference on*

Computer-Aided Design, pages 612–617. IEEE Computer Society Press, 1993.

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

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

[8] L. 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.

[9] L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

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

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

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

[13] V. D'Silva, S. Ramesh, and A. 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.

[14] N. A. Lynch and M. 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 Aug. 1987.

[15] S. Narayan and D. D. Gajski. Interfacing incompatible protocols using interface process generation. In *Proc. of the 32nd Design Automation Conference*, June 1995.

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

[17] R. Passerone, L. de Alfaro, T. 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.

[18] R. Passerone, J. A. Rowson, and A. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proc. of the 35th Design Automation Conference*, June 1998.

[19] D. Shin and D. D. Gajski. Interface synthesis from protocol specification. Technical report, CECS Technical Report 02-13, April 2002.

[20] J. Smith and G. D. Micheli. Automated composition of hardware components. In *Proc. of the 35th Design Automation Conference*, June 1998.

[21] W. 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.

[22] N. Yevtushenko, T. Villa, R. K. Brayton, A. Petrenko, and A. 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.

[23] N. Yevtushenko, T. Villa, R. K. Brayton, A. Petrenko, and A. 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.