Convertibility Verification and Converter Synthesis: Two Faces of the Same Coin[*]

Roberto Passerone
Cadence Berkeley Laboratories
Berkeley, CA 94704

Luca de Alfaro
University of California, Santa Cruz
Santa Cruz, CA 95064

Thomas A. Henzinger, Alberto L. Sangiovanni-Vincentelli
University of California, Berkeley
Berkeley, CA 94720

Abstract

An essential problem in component-based design is how to compose components designed in isolation. Several approaches have been proposed for specifying component interfaces that capture behavioral aspects such as interaction protocols, and for verifying interface compatibility. Likewise, several approaches have been developed for synthesizing converters between incompatible protocols. In this paper, we introduce the notion of adaptability as the property that two interfaces have when they can be made compatible by communicating through a converter that meets specified requirements. We show that verifying adaptability and synthesizing an appropriate converter are two faces of the same coin: adaptability can be formalized and solved using a game-theoretic framework, and then the converter can be synthesized as a strategy that always wins the game. Finally we show that this framework can be related to the rectification problem in trace theory.

1. Introduction

Composing Intellectual Property blocks is an essential element of a design methodology based on re-use. The composition of these blocks when the IPs have been developed by different groups inside the same company, or by different companies, is notoriously difficult. Side effects often make the behavior of the resulting design unpredictable. Design rules have been proposed that try to alleviate the problem by forcing the designers to be precise about the behavior of the individual components and to verify this behavior under a number of assumptions about the environment in which they have to operate. While this is certainly a step in the right direction, it is by no means sufficient to guarantee correctness: extensive simulation and prototyping are still needed on the compositions. Several methods have been proposed for hardware and software components that encapsulate the IPs so that their behavior is protected from the interaction with other components. Interfaces are then used to ensure the compatibility between components. Roughly speaking, two interfaces are compatible if they "fit together" as they are.

Simple interfaces, typically specified in the type system of a system description language, may describe the types of values that are exchanged between the components. More expressive interfaces, typically specified informally in design documents, may describe the protocol for the component interaction [1, 2, 3, 4, 5, 6]. In [4, 5] we presented a formal methodology for specifying the protocol aspects of interfaces in a way that supports automatic compatibility checks. The key element of the approach is the interpretation of an interface as a game between a component and its environment, and the use of game-theoretic algorithms for compatibility checking. With this approach, given interfaces for different IPs, one can check whether these IPs can be composed.

When components are taken from legacy systems or from third-party vendors, it is likely that the interface protocols are not compatible. This does not mean though that we are doomed: approaches have been proposed that construct a converter among incompatible communication protocols. We refer the reader to [2] for references and a discussion of related work and for a general overview and description of the problem of protocol conversion. In [2] we proposed to define a protocol as a formal language (a set of strings from an alphabet) and to use automata to finitely represent such languages. The problem of converting one protocol into another was then addressed by considering their conjuction as the product of the corresponding automata and by removing the states and transitions that led to a violation of one of the two protocols. While the algorithm was effective in the examples that were tried, it lacked a more formal and mathematically sound interpretation. In particular this made it difficult to understand and analyse its limitations and properties.

In this paper, we combine and extend the results of both [4] and [2]. In particular, we apply the game-theoretic interface paradigm of [4, 5] not only to the checking of interface compatibility,

[*] This work was supported in part by the MARCO/DARPA Gigascale Silicon Research Center (http://www.gigascale.org), DARPA grant F33615-00-C-1693, NSF grant CCR-9988172, ONR grant N00014-02-1-0671, SRC grant 99-TJ-683.003, and NSF CAREER award CCR-0132780. Their support is gratefully acknowledged.

but also to the synthesis of *interface adaptors*, should the original interfaces be incompatible. Informally, two interfaces are adaptable if they can be made to fit together by communicating through a third component, the adaptor. If interfaces specify only value types, then adaptors are simply type converters. However, if interfaces specify interaction protocols, then adaptors are protocol converters. The converter may need state to re-arrange the communication between the original interfaces, in order to ensure compatibility[1]. A novel aspect of our approach is that the protocol converter is synthesized from a specification that says which re-arrangements are appropriate in a given communication context. For instance, it is possible to specify that the converter can change the timing of messages, but not their order, using an n-bounded buffer, or that some messages may be duplicated. Following and extending the approach of [4], we synthesize interface protocol converters using game-theoretic methods. In this way we provide a general formalization and a uniform solution, based on game theory, for the protocol conversion problem of [2].

In Section 2 we illustrate with an example the automata-based approach to the synthesis of protocol converters [2]. Unlike [2], here we derive the converter starting from a specification of what constitutes an acceptable protocol conversion. We set up and solve the conversion problem for send-receive protocols, where the *sender* and the *receiver* are specified as automata. A third automaton, the *requirement*, specifies constraints on the converter, such as buffer size and the possibility of message loss. In Section 3 we recast the problem of protocol conversion in a more general, game-theoretic framework inspired by [4]. This approach also provides us with ready-to-use solution algorithms. Here the solution consists of a winning strategy that complies with both the receive protocol and the requirement in a game against the sender (which must be allowed to follow any sequence of actions permitted by the send protocol). If a winning strategy exists, then the two protocols are convertible and the winning strategy can be used to synthesize the converter. This shows that convertibility verification and converter synthesis are indeed two faces of the same coin. In Section 4 we outline how the method can be extended to deal with fairness constraints on the converter. Finally, in Section 5 we adapt the solution to the rectification problem of [9] to produce a protocol converter, and we contrast this approach to our methodology.

2. Automata-based Solution

We illustrate our approaches to protocol conversion by way of an example, which is an extension (and in some sense, also a simplification) of the one found in [2]. A producer and a consumer component wish to communicate some complex data across a communication channel. They both partition the data into two parts. The interface of the producer is defined so that it can wait an unbounded amount of time between the two parts. On the other hand, the interface of the consumer is defined so that it requires that once the first part has been received, the second is also received during the state transition that immediately follows the first. Clearly, the two protocols are incompatible. Below, we illustrate how to synthesize a converter that enables them to communicate correctly.

The two protocols can be represented by the automata shown in Figure 1. There, the symbols a and b (and their primed counterparts) are used to denote the first and the second part of the data, respectively. The symbol $\top$ denotes instead the absence or irrelevance of the data. In other words, it acts as a don't care.

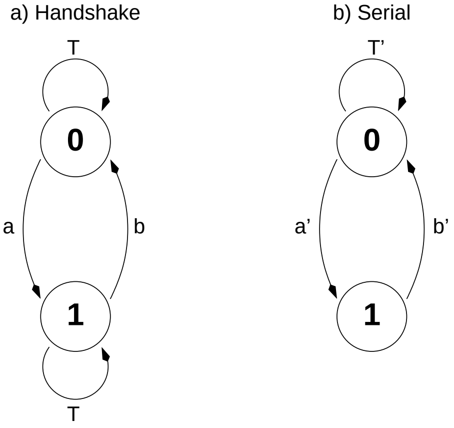

Figure 1: Handshake and serial protocols

Figure 1.a shows the producer protocol. The self loop in state 1 indicates that the transmission of a can be followed by any number of cycles before b is also transmitted. We call this protocol *handshake* because it could negotiate when to send the second part of the data. After b is transmitted, the protocol returns to its initial state, and is ready for a new transaction. The ability to handle multiple transactions is also an extension of our previous work.

Figure 1.b shows the receiver protocol. Here state 1 does not have a self loop. Hence, once a has been received, the protocol assumes that b is transmitted in the cycle that immediately follows. This protocol is called *serial* because it requires a and b to be transferred back-to-back. Similarly to the sender protocol, once b is received the automaton returns to its initial state, ready for a new transaction.

We have used non-primed and primed versions of the symbols in the alphabet of the automata to emphasize that the two sets of signals are different and should be connected through a converter. It is the specification (below) that defines the exact relationships that must hold between the elements of the two alphabets. Note that in the definition of the two protocols nothing relates the quantities of one (a and b) to those of the other (a' and b'). The symbol a could represent the toggling of a signal, or could symbolically represent the value of, for instance, an 8-bit variable. It is only in the interpretation of the designer that a and a' actually hold the same value. The specification that we are about to describe does not enforce this interpretation, but merely defines the (partial) order in which the symbols can be presented to and produced by the converter. It is possible to represent explicitly the values passed; this is necessary when the behavior of the protocols depends on the data, or when the data values provided by one protocol must be modified (translated) before being forwarded to the other protocol. The synthesis of a protocol converter would then yield a converter capable of both translating data values, and of modifying their timing and order. However, the price to pay for the ability to synthesize data translators is the state explosion in the automata to describe the interfaces and the specification. Observe also that if a and b are symbolic representation of data, some other means must be available in the implementation to distinguish when the actual data corresponds to a

[1] Hence the notion of protocol converter can be seen as a special case of the notion of behavior adaptor introduced in [7] to characterize a modeling approach for communication-based design that is the basis of the Metropolis framework [8].

or to b. At this level of the description we don't need to be specific; examples of methods include toggling bits, or using data fields to specify message types.

What constitutes a correct transaction? Or in other words, what properties do we want the communication to have? In the context of this particular example the answer seems straightforward. Nonetheless, different criteria could be enforced depending on the application. Each criterion is embodied by a different specification.

One example of a specification is shown in Figure 2. The alphabet of the automaton is derived from the Cartesian product of the alphabets of the two protocols for which we want to build a converter. This specification states that no symbols should be discarded or duplicated by the converter, and symbols must be delivered in the same order in which they were received; moreover, the converter can store at most one undelivered symbol at any time. The three states in the specification correspond to three distinct cases.

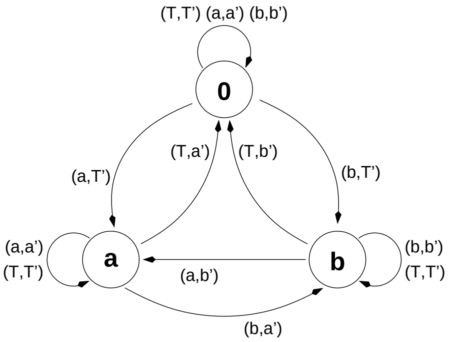

Figure 2: Specification automaton

- State **0** denotes the case in which all received symbols have been delivered (or that no symbol has been received, yet).
- State **a** denotes the case in which symbol a has been received, but it hasn't been output yet.
- Similarly, state **b** denotes the case in which symbol b has been received, but not yet output.

Note that this specification is not concerned with the particular form of the protocols being considered (or else it would itself function as the converter); for example, it does not require that the symbols a or b are received in any particular order (other than the one in which they are sent). On the other hand, the specification makes precise what the converter can, and cannot do, ruling out for instance converters that simply discard all input symbols from one protocol, never producing any output for the destination protocol. In fact, we can view the specification as an observer that specifies what *can* happen (a transition on some symbol is available) and what *should not* happen (a transition on some symbol is not available). As such, it is possible to decompose the specification into several automata, each one of which specifies a particular property that the synthesized converter should exhibit. This is similar to the monitor-based property specification proposed by Shimizu et al. [3] for the verification of communication protocols. In our work, however, we use the monitors to drive the synthesis so that the converter is guaranteed to exhibit the desired properties (correct-by-construction).

A high-level view of the relationship between the protocols and the specification is presented in Figure 3. The protocol *handshake* produces outputs a and b, the protocol *serial* accepts inputs a' and b'. The specification accepts inputs a, b, a', b', and acts as a global observer that states what properties the converter should have. Once we compose the two protocols and the specification, we obtain a system with outputs a, b, and inputs a', b' (Figure 3). The converter will have inputs and outputs exchanged: a and b are the converter inputs, and a', b' its outputs.

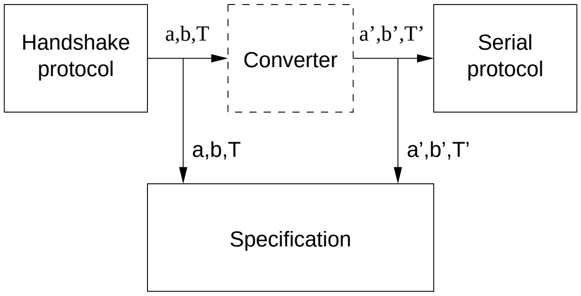

Figure 3**: Inputs and outputs of protocols, specification, and converter.**

The synthesis of the converter begins with the composition (product machine) of the two protocols, shown in Figure 4. Here the direction of the signals is reversed: the inputs to the protocols become the outputs of the converter, and vice versa. This composition is also a specification for the converter, since on both sides the converter must comply with the protocols that are being interfaced. However this specification doesn't have the notion of synchronization (partial order, or causality constraint) that the specification discussed above dictates.

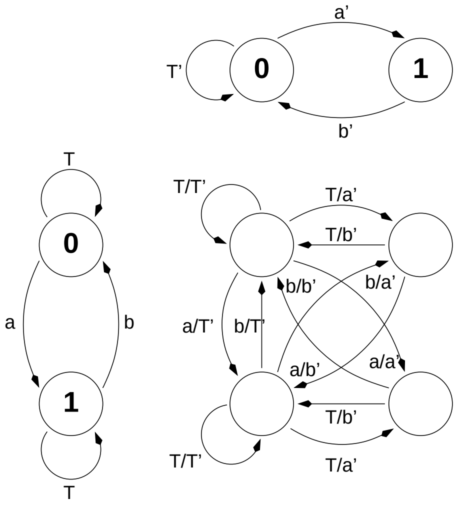

Figure 4: Composition between handshake and serial

We can ensure that the converter satisfies both specifications by taking the converter to be composition of the product machine with the specification. Figures 5 through 7 explicitly show the steps that we go through to compute this product. The position of the state reflects the position of the corresponding state in the protocol composition, while the label inside the state represents the corresponding state in the specification. Observe that the bottom-right state is reached when the specification goes back to state **0**. This procedure corresponds to the synthesis algorithm proposed in our previous work [2]. The approach here is however fundamentally different: the illegal states are defined by the specification, and not by the particular algorithm employed.

The initial step is shown in Figure 5. The composition with the specification makes the transitions depicted in dotted line illegal (if taken, the specification would be violated). However, transitions can be removed from the composition only if doing so does not result in an assumption on the behavior of the sender. In Figure 5, the transition labeled $\top/a'$ leaving state **0** can be removed because the machine can still respond to a $\top$ input by taking the self loop, which is legal. However, removing the transition labeled $\top/b'$ leaving the bottom-right state would make the machine unreceptive to input $\top$. Equivalently, the converter is imposing an assumption on the producer that $\top$ will not occur in that state. Because this assumption is not verified, and because we can't change the producer, we can only avoid the problem by making the bottom-right state unreachable, and remove it from the composition.

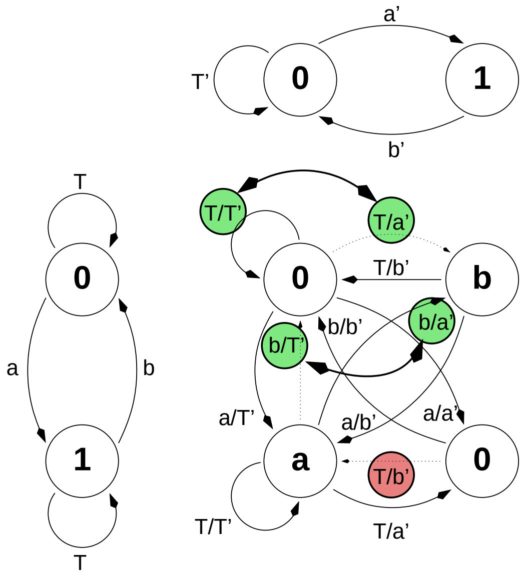

Figure 5: Interface cross Correctness, phase 1

The result is shown in Figure 6. The transitions that are left dangling because of the removal of the state should also be removed, and are now shown in dotted lines. The same reasoning as before applies, and we can only remove transitions that can be replaced by others with the same input symbol. In this case, all illegal transitions can be safely removed.

The resulting machine shown in Figure 7 has now no illegal transitions. This machine complies both with the specification and with the two protocols, and thus represents the correct conversion (correct relative to the specification). Notice how the machine at first stores the symbol a without sending it (transition $a/\top'$). Then, when b is received, the machine sends a', immediately followed in the next cycle by b', as required by the serial protocol.

3. Game-Theoretic Solution

We use game theory to reformulate the problem and the procedure discussed in the previous section more precisely.

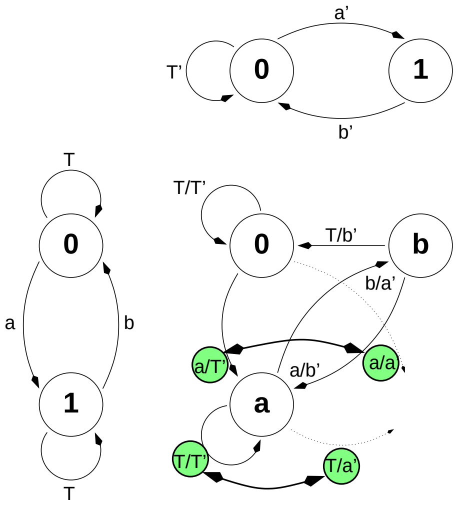

Figure 6: Interface cross Correctness, phase 2

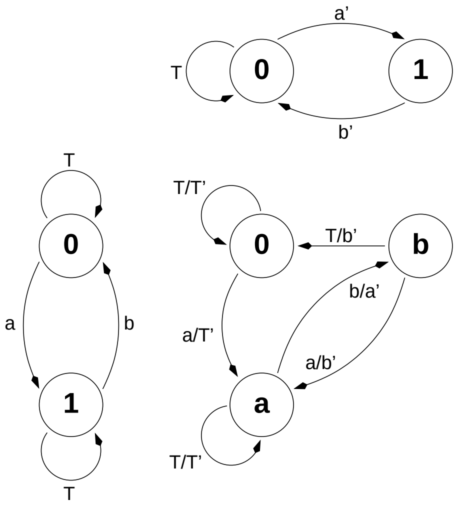

Figure 7: Interface cross Correctness, phase 3

In the previous section we have used an automaton to describe a protocol. Here we use a slightly enhanced version of an automaton that provides a function that explicitly tells for each state the set of available transitions; we call such structure a *transition structure*. Given an alphabet Σ, a transition structure over Σ consists of the following:

- A state space Q.
- An *initial state* $q_0 \in Q$.
- A *transition-availability function* $\sigma \colon Q \to 2^{\Sigma}$.
- A *transition-outcome function* $\delta \colon Q \times \Sigma \to Q$.

Here the transition-outcome function corresponds to the traditional transition relation of automata. The transition-availability function, on the other hand, specifies for each state the set of symbols that the transition structure is sensitive to. We say that a transition structure is *non-blocking* if for every $q \in Q$ we have $\sigma(q) \neq \emptyset$, so that there is at least one symbol available at every state. The transition-outcome function is meaningful only for those symbols that are available in a state. In the following, we will use the symbol "$-$" to indicate that the actual value is immaterial.

It is easy to translate the two protocols of Figure 1 and the specification of Figure 2 in the above representation. For the handshake protocol we have:

- $\Sigma_1 = \{\top, a, b\}$
- $Q_1 = \{0, 1\}$
- $q_1 = 0$
- $\sigma_1 =$

0	$\{\top, a\}$
1	$\{\top, b\}$

- $\delta_1 =$

	$\top$	a	b
0	0	1	$-$
1	1	$-$	0

For the serial protocol we have:

- $\Sigma_2 = \{\top', a', b'\}$
- $Q_2 = \{0', 1'\}$
- $q_2 = 0'$
- $\sigma_2 =$

$0'$	$\{\top', a'\}$
$1'$	$\{b'\}$

- $\delta_2 =$

	$\top'$	a'	b'
$0'$	$0'$	$1'$	$-$
$1'$	$-$	$-$	$0'$

And finally the specification is obtained as follows.

- $\Sigma_0 = \Sigma_1 \times \Sigma_2 = \{(\top, \top'), (\top, a'), (\top, b'), (a, \top'), (a, a'), (a, b'), (b, \top'), (b, a'), (b, b')\}$
- $Q_0 = \{0, a, b\}$
- $q_0 = 0$
- $\sigma_0 =$

0	$\{(\top, \top'), (a, \top'), (a, a'), (b, \top'), (b, b')\}$
a	$\{(\top, \top'), (\top, a'), (a, a'), (b, a')\}$
b	$\{(\top, \top'), (\top, b'), (a, b'), (b, b')\}$

- $\delta_2 =$

	$(\top, \top')$	$(\top, a')$	$(\top, b')$	$(a, \top')$
0	0	$-$	$-$	a
a	a	0	$-$	$-$
b	b	$-$	0	$-$

	(a, a')	(a, b')	$(b, \top')$	(b, a')	(b, b')
0	0	$-$	b	$-$	0
a	a	$-$	$-$	b	$-$
b	$-$	a	$-$	$-$	b

In the previous section, the protocol conversion problem is solved by constructing the product machine of the automata that represent the protocols. Here we solve the same problem as a game between two players. Intuitively, with reference to Figure 3, synthesizing a converter corresponds to solving a game: can the converter, by reading outputs a and b, provide inputs a' and b' that satisfy both the protocols and the specification? The game is played between the protocols and the specification, on one side, and the converter, on the other side. A move of the protocols and specification consists in choosing whether to emit a, b, or $\top$, and is thus essentially a move of the first protocol. A move of the converter consists in choosing a conversion function $f : \{a, b, \top\} \rightarrow \{a', b', \top'\}$, that is used to convert a symbol of the first protocol into one of the second. The goal of the game, for the converter, consists in ensuring that, if the first protocol emits $a, b, \top$ according to its definition (Figure 1.a), then the symbol produced by the conversion function corresponds to a transition both of the second protocol, and of the specification. If the game can be won, the converter consists simply of an implementation of a winning strategy.

We represent a game using a structure similar to a transition structure, but with two alphabets (one per player), two move availability functions, and a joint move-outcome function. Precisely, given two alphabets Σ_1 and Σ_2, a *game structure* G over Σ_1 and Σ_2 consists of the following:

- A state space Q.
- An *initial state* $q_0 \in Q$.
- Two *move-availability functions*, σ_1: $Q \rightarrow 2^{\Sigma_1} \setminus \emptyset$ for player 1, and σ_2: $Q \rightarrow 2^{\Sigma_2}$ for player 2.
- A *move-outcome function* δ: $Q \times \Sigma_1 \times \Sigma_2 \rightarrow Q$.

Similarly to the transition structures, the move-availability functions provide for each state the set of moves that each player is allowed to play. We require that player 1 has at least one move available at every state. The move-outcome function gives us the state that is reached in response to a particular pair of moves of the players. As for the transition-outcome function in transition structures, this is meaningful only in conjunction with moves that are available in each state.

When does one player win the game? The two players start from the initial state q_0 and each chooses one of their available moves for that state. The game then transitions to the state indicated by the value of the function δ corresponding to the current state and the chosen moves. Player 2 wins if it always has a move available for all the states that are reached during the game. Player 1, on the other hand, wins if the game transitions to a state q such that player 2 has no available moves (i.e. $\sigma_2(q) = \emptyset$).

Instead of randomly choosing a move for each state during the game, a player can play according to a *strategy*. A strategy for a player is simply a pre-determined way of choosing the move for each state. In general, a strategy is a function that maps the current history of the game to one of the available moves. In practice, if only safety properties are considered in the specification, a simpler strategy that only looks at the current state is sufficient. These strategies are known as *memory-less* strategies. A memory-less strategy for player 2 is a function $\xi_2 : Q \rightarrow \Sigma_2$ such that for $q \in Q$, if $\sigma_2(q) \neq \emptyset$, then $\xi_2(q) \in \sigma_2(q)$: in other words, the strategy assigns only symbols that can be played by player 2, provided at least one such symbol exists.

Given a strategy ξ_2 for player 2, not all states in G can be reached. We define the *space reached in G under ξ_2* to be the smallest set $R(\xi_2) \subseteq Q$ such that (1) $q_0 \in R(\xi_2)$ and (2) for all $q \in R(\xi_2)$ and $m_1 \in \sigma_1(q)$, we have $\delta(q, m_1, \xi_2(q)) \in R(\xi_2)$. We say that a strategy is *winning* if a player never loses when playing according to the strategy. Given our definitions, ξ_2 is a winning strategy for player 2 if for all reachable states $q \in R(\xi_2)$ we have $\sigma_2(q) \neq \emptyset$. In other words, the strategy ξ_2 steers the game in such a way that player 2 has always moves available to play.

We are now ready to set up the protocol conversion problem as a game. We will use the convention that if $A = (Q, q_0, \sigma, \delta)$ is a transition structure with alphabet M, then $Q[A] = Q$, $q_0[A] = q_0$ and so on. Thus, let A and B be the transition structures for the sender and the receiver protocol with alphabets M and M', respectively; we require that the transition structure A for the sender is non-blocking. Let C be the transition structure for the specification, with alphabet $M \times M'$. Then the protocol conversion problem is defined as a game structure G with alphabets $\Sigma_1 = M$ and $\Sigma_2 = [M \to M']$ (the notation $[M \to M']$ denotes the set of all functions from M to M') such that:

- $Q[G] = Q[A] \times Q[B] \times Q[C]$.
- $q_0[G] = (q_0[A], q_0[B], q_0[C])$.
- $\sigma_1[G](q_A, q_B, q_C) = \sigma[A](q_A)$ and $\sigma_2[G](q_A, q_B, q_C) = \{f : M \to M' \mid \forall m \in M.\ f(m) \in \sigma[B](q_B) \ \wedge$ $(m, f(m)) \in \sigma[C](q_C)\}$.
- $\delta[G](q_A, q_B, q_C, m, f) =$ $(\delta[A](q_A, m), \delta[B](q_B, f(m)), \delta[C](q_C, m, f(m)))$.

Note that the game structure is defined on the Cartesian product of the states of the communicating protocols and the specification. Intuitively, these definitions define, for each move of A, the set of possible responses of the converter that agree with the protocol of B while satisfying the specification C. It is possible that for some of the states the set of possible responses is empty. The objective is thus to find a strategy that avoids such states. In other words, a *protocol converter* is a winning player-2 strategy for G. If no such strategy exists, then the protocol conversion problem cannot be solved, and the two protocols are thus not adaptable.

In the case of our example, the definitions translate to a game structure G over Σ_1 and $\Sigma_1 \to \Sigma_2$ with

- $Q[G] = Q_1 \times Q_2 \times Q_0$
- $q[G] = (q_1, q_2, q_0)$
- $\sigma_1[G] =$

$(0, *, *)$	$\{\top, a\}$
$(1, *, *)$	$\{\top, b\}$

- $\sigma_2[G] =$

$(*, 0', 0)$	$\{(\top \to \top', a \to a', b \to \top'),$ $(\top \to \top', a \to \top', b \to \top')\}$
$(*, 0', a)$	$\{(\top \to \top', a \to a', b \to a'),$ $(\top \to a', a \to a', b \to a')\}$
$(*, 0', b)$	$\emptyset$
$(*, 1', 0)$	$\emptyset$
$(*, 1', a)$	$\emptyset$
$(*, 1', b)$	$\{(\top \to b', a \to b', b \to b')\}$

According to the definitions, we must find a winning player-2 strategy for this game structure. Let $T_{nonblock} = \{q \in Q \mid \sigma_2(q) \neq \emptyset\}$ be the set of states where player 2 is non-blocked. The game can then be cast as a *safety* game: player 2 must never leave $T_{nonblock}$. The solution to this safety game is entirely classical [10, 11]. To solve the games, for a set of states $U \subseteq Q$ we define the set $Cpre(U) \subseteq Q$ of *controllable predecessors* of U as the set consisting of all states $q \in Q$ where there is $m_2 \in \sigma_2(q)$ such that for all $m_1 \in \sigma_1(q)$, we have $\delta(q, m_1, m_2) \in U$. In formulas,

$$\begin{aligned} &Cpre(U) \\ &\quad = \ \{q \in Q \mid \exists m_2 \in \sigma_2(q).\forall m_1 \in \sigma_1(q).\delta(q, m_1, m_2) \in U\} \end{aligned}$$

Intuitively, $q \in Cpre(U)$ if player 2 can reply to every move of player 1 and ensure that U is not left. Then, we compute the sequence of sets $U_0, U_1, U_2, \ldots \subseteq Q$ by letting $U_0 = T_{nonblock}$ and, for $k \geq 0$, $U_{k+1} = T_{nonblock} \cap Cpre(U_k)$. For each $k \geq 0$, the set U_k consists of the states from which player 2 can stay in $T_{nonblock}$ for at least k rounds of the game: hence, the set of states where player 2 can win is given by $U^* = \lim_{k \to \infty} U_k$. Since $U_{k+1} \subseteq U_k$ for all $k \geq 0$, the limit can be computed in at most $|Q|$ steps, terminating as soon as we reach k such that $U_{k+1} = U_k$. Player 2 can win the game if the initial state is winning, that is, $q_0 \in U^*$; in this case, a winning strategy ξ_2 for player 2 can be constructed by choosing, at each $s \in U^*$ with $\sigma_1(q) \neq \emptyset$, a move $m_2 \in \sigma_2(q)$ such that $\delta(q, m_1, m_2) \in U^*$ for all $m_1 \in \sigma_1(q)$; at other states the strategy plays arbitrarily. A protocol converter can be obtained directly by implementing this strategy. The converter keeps track of the state of the game; at each state q, it uses the conversion function $\xi_2(q)$ to convert the move m_1 chosen by the first protocol, and then goes to the new state $\delta(q, m_1, \xi_2(q)(m_1))$. The complexity of synthesizing the interface adaptor is linear in the size of the game structure G.

The resulting converter, in our example, can be represented as the automaton of Figure 7. At state 0 the converter plays the move f_0 defined by $f_0(\top) = \top'$, $f_0(a) = \top'$, $f_0(b) = \top'$ (this latter choice is arbitrary). At state 1 the converter plays the move f_1 defined by $f_1(\top) = \top'$, $f_1(a) = \top'$ (arbitrary), and $f_1(b) = a'$. At state 2 the converter plays the move f_2 defined by $f_2(\top) = b'$, $f_2(a) = b'$ and $f_2(b) = \top'$ (arbitrary). Once the moves m, f are played, the strategy changes state according to the edge labeled $m, f(m)$ in Figure 7.

4. Fairness Constraints

The theory and the algorithms that we have presented so far are appropriate for describing specifications that consist in safety constraints. It is often necessary in communication protocols to also define liveness constraints, for example to specify that the transfer of the data does eventually occur. Indeed, the specification automaton of Figure 2 specifies that symbols should not be dropped nor duplicated by the channel, but it does not specify that a symbol a or b should be eventually delivered, if followed by infinitely many occurrences of the symbol $\top$. Our approach can be extended in this direction, although the algorithms used to solve the protocol conversion problem become more complicated, and may be non-linear [12].

Figure 8 shows an automaton that is equivalent to the one that we have used so far for the specification, and that is suitable for specifying the eventual delivery of symbols. In particular, we had to duplicate two of the states to separate the transitions on data from those that occur when no data is present. Fairness constraints can be defined in a variety of formalisms. Here we use temporal logic [13], and insist that the automaton never stays in state **a**, **aa**, **b** or **bb** forever. Formally:

$$\Box\Diamond\neg a \wedge \Box\Diamond\neg aa \wedge \Box\Diamond\neg b \wedge \Box\Diamond\neg bb.$$

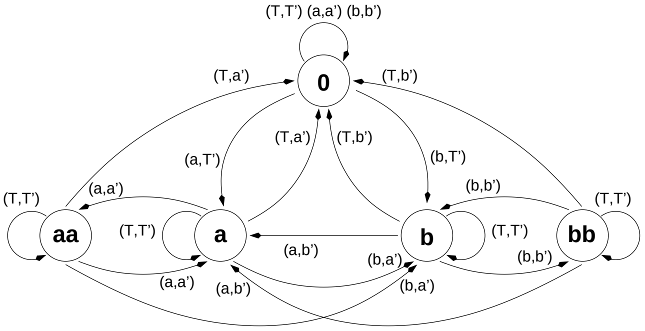

Figure 8: Specification for fairness constraints

When the specification includes a fairness condition ϕ (such as the above), we must modify the definition of winning strategy for player 2. In particular, we need to consider *history-dependent* strategies, that choose the move as a function not only of the current state, but also of the past history of the game [12]. A (history-dependent) strategy ξ_i for player $i \in \{1, 2\}$ is a mapping $\xi_i : Q^* \rightarrow \Sigma_i$ that associates with each sequence of states a move in the alphabet of the player. Again, for $i \in \{1, 2\}$ we require that for all $\alpha \in Q^*$ and $q \in Q$, if $\sigma_i(q) \neq \emptyset$ then $\xi_i(\alpha, q) \in \sigma_i(q)$: in other words, as long as at least one move is available, the strategy chooses only moves that are available. Given two strategies ξ_1 and ξ_2 for players 1 and 2 and a state $q \in Q$, we define $\mathit{outcome}(q, \xi_1, \xi_2)$ to be the set consisting of the infinite sequences $p_0, p_1, p_2, \ldots$ such that:

- $p_0 = q$;
- for all $k \geq 0$, we have $\sigma_1(p_k) \neq \emptyset$ and $\sigma_2(p_k) \neq \emptyset$;
- for all $k \geq 0$, we have $p_{k+1} = \delta(p, \xi_1(p_0, \ldots, p_k), \xi_2(p_0, \ldots, p_k))$.

We note that the set $\mathit{outcome}(q, \xi_1, \xi_2)$ either contains a single sequence, which is entirely contained in $T_{nonblock}$, or it is empty. The latter case arises when, starting from q, the strategies ξ_1, ξ_2 would cause $T_{nonblock}$ to be left. Then, a strategy ξ_2 for player 2 is *winning* if, for all player-1 strategies ξ_1, there is $\overline{p} \in \mathit{outcome}(q_0, \xi_1, \xi_2)$ such that $\overline{p} \models \phi$. Again, a protocol converter corresponds to a winning strategy. Games with such fairness conditions and winning conditions are examples of games with ω-regular winning conditions [11]. Such games can be solved by the methods of [11, 14, 15, 12]; a winning strategy can be easily derived from a game solution. We note that, when fairness is present, the winning strategy may require memory. Nevertheless, the protocol converter can again be synthesized as an automaton, since the amount of required memory is finite (see e.g. [12]).

5. Comparison with Trace Theory

As seen in the previous sections, the models of interfaces that we use (including the ones found in [2] and [4]) constrain the set of possible environments they can work with by considering illegal those inputs that cannot be handled. This is unlike other state-based models like I/O automata [16] that must always be ready to handle any possible input. For this reason, the latter models are often called *input-enabled*, or *receptive*.

Receptiveness and environmental constraints are not, however, mutually exclusive. The two notions coexist and are particularly well-behaved in the trace theory proposed by Dill [1]. In this framework, developed for the verification of speed-independent asynchronous circuits, a trace of a model (which corresponds to the sequence of inputs and outputs on a path in our automata) is defined to be either a success or a failure. A model, called a trace structure, contains a set of successes and a set of failures. A trace structure must be receptive in that any sequence of inputs should be accepted as either a success or a failure (by producing appropriate outputs). However, the set of failures can be used to represent the behaviors that should be avoided by an environment to work correctly with the trace structure.

The theory defines several concepts that are relevant to our work. Here we provide only a brief description and refer the interested reader to [1] for a rigorous treatment of the subject.

Trace structures can be composed in parallel by taking the intersection of the sets of the success traces. At the same time, the composition contains the failures from the trace structures that are being composed that are compatible with the behavior (successes or failures) from the other. The result is shown to also be receptive. A composition is called *failure-free* if the resulting set of failures is empty.

A trace structure T is said to *conform to* a trace structure T' if for all possible environments E, if the composition of E and T' is failure-free, so is the composition of E and T. Conformation can be seen as a refinement relation, or substitutability relation, because if T conforms to T', then we can replace T for T' without introducing failures in any environment that composes with T' without failures. Two trace structures are said to be *conformation equivalent* if they both conform to each other.

In our state-based and game-theoretic models, the "failures" are implicitly defined as those traces that are not accepted by the automaton because a transition on a symbol does not exist. The set of failures is, in other words, determined by the set of successes. The same is achieved in trace theory by a construction that takes a trace structure and produces another trace structure in *canonical form*, that is conformation equivalent to the original. In canonical form, the set of successes and the set of inputs together determine the set of failures, so they don't have to be represented explicitly. In addition, two trace structures in canonical form are shown to be equal exactly when they are conformation equivalent.

Parallel composition of trace structures in canonical form is not guaranteed to yield a trace structure that is again in canonical form. If that is the desired result, the composition must be followed by a conversion to canonical form. The same holds true for our models. In fact, it turns out that the process of removal of illegal states outlined in the previous sections corresponds to that of *autofailure manifestation* used to convert a general trace structure to one in canonical form. An autofailure is a successful trace that becomes a failure as a consequence of the occurrence of output symbols that no environment can stop. From the standpoint of conformation, the result doesn't change if autofailures are added to the set of failures.

In the context of protocol specification, the concept of a *mirror* plays an important role. The mirror of a trace structure T is the most general environment (again a trace structure) that can be composed with T with no failures. In other words, T can be composed without failures with all and only the trace structures that conform to its mirror. Computing the mirror is particularly simple once a trace structure is in canonical form: it is enough to exchange the inputs with the outputs.

The problem of the synthesis of the protocol converter can now be analyzed in the context of trace theory. Specifically, we adapt the formulation of the *rectification* problem found in [9]. There, rectification is defined as the process of replacing a subnetwork of a circuit by another subnetwork such that the resulting system conforms to a certain specification. More formally, let $\|$ denote parallel composition, $\subseteq$ denote conformation and *mir* the operation of taking the mirror. Then, if T is the specification, T_1' the replacement subnetwork and T_2 the rest of the circuit

$$T_1' \parallel T_2 \subseteq T \text{ if and only if } T_1' \subseteq \mathit{mir}(T_2 \parallel \mathit{mir}(T)).$$

The original theorem in [9] also includes the possibility of retaining only part of the symbols in the alphabet. This is useful to make very precise the set of signals (symbols) that the replacement subnetwork can use to perform its function. If the function $\mathit{proj}(A)$ retains only the symbols in A, then if A is the alphabet of the specification T and A_1' the alphabet of the replacement subnetwork T_1', we have

$$\mathit{proj}(A)(T_1' \parallel T_2) \subseteq T$$

if and only if

$$T_1' \subseteq \mathit{mir}(\mathit{proj}(A_1')(T_2 \parallel \mathit{mir}(T))).$$

In our specific case, we take T_2 to be the composition of the two protocols and T to be the specification of a correct transaction. Then the T_1' that is obtained by the above formula corresponds to the trace structure for the converter.

Preliminary experiments show the viability of this approach and that it produces results that are equivalent to our earlier procedures when using the same examples. However a thorough comparison is made difficult by the fact that the timing models used in the two solutions are different (synchronous in our case, asynchronous in trace theory verifier based on [1] that was available to us). Nevertheless, the algebraic structure of the trace theoretic approach is very convenient and the formulation in terms of the rectification problem is clean and concise. A generalization of this approach is part of our current research.

6. Conclusions

We presented a general, algorithmic framework for checking whether incompatible interaction protocols of component interfaces can be made compatible by inserting a converter between them which satisfies specified requirements. The framework is based on and extends the game-theoretic approach to interfaces of [4, 5]. If a winning strategy for the game exists, then the interfaces are adaptable and the winning strategy yields the converter as a by-product. The approach also extends and formalizes previous work on the synthesis of protocol converters [2].

We noted an interesting relationship between our approach and the one of [9]. In this work, trace theory was used to study the so-called rectification problem, which can also be used to synthesize protocol converters. Our research now concentrates on combining and generalizing our approach and the trace-theory approach. In particular, we are investigating frameworks that encompass multiple models of computation and enable the synthesis of converters across different models.

7. Acknowledgment

The authors would like to thank Jerry Burch of the Cadence Berkeley Labs for many insightful discussions and suggestions.

8. REFERENCES

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

[2] R. Passerone, J. A. Rowson, and A. L. Sangiovanni-Vincentelli, “Automatic synthesis of interfaces between incompatible protocols,” in *DAC*, (San Francisco, CA), June 1998.

[3] K. Shimizu, D. L. Dill, , and A. J. Hu, “Monitor-based formal specification of PCI,” in *FMCAD*, (Austin, Texas), 2000.

[4] L. de Alfaro and T. A. Henzinger, “Interface automata,” in *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pp. 109–120, ACM Press, 2001.

[5] L. de Alfaro and T. A. Henzinger, “Interface theories for component-based design,” in *Proceedings of the First International Workshop on Embedded Software, Lecture Notes in Computer Science*, pp. 148–165, Springer-Verlag, 2001.

[6] S. Chaki, S. Rajamani, and J. Rehof, “Types as models: Model checking message-passing programs,” in *Proc. 29th ACM Symp. Princ. of Prog. Lang.*, 2002.

[7] A. Sangiovanni-Vincentelli, M. Sgroi, and L. Lavagno, “Formal models for communication-based design,” in *Proceedings of the Eleventh International Conference on Concurrency Theory*, August 2000.

[8] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Y. Watanabe, and G. Yang, “Concurrent execution semantics and sequential simulation algorithms for the metropolis meta-model,” in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, (Estes Park, CO), May 2002.

[9] J. R. Burch, D. L. Dill, E. S. Wolf, and G. D. Micheli, “Modeling hierarchical combinational circuits,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD’93)*, pp. 612–617, November 1993.

[10] J. Thatcher and J. Wright, “Generalized finite automata theory with an application to a decision problem of second-order logic,” *Mathematical System Theory*, vol. 2, pp. 57–81, 1968.

[11] J. Büchi and L. Landweber, “Solving sequential conditions by finite-state strategies,” *Trans. Amer. Math. Soc.*, vol. 138, pp. 295–311, 1969.

[12] W. Thomas, “On the synthesis of strategies in infinite games,” in *Proc. of 12th Annual Symp. on Theor. Asp. of Comp. Sci.*, vol. 900 of *Lect. Notes in Comp. Sci.*, pp. 1–13, Springer-Verlag, 1995.

[13] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. New York: Springer-Verlag, 1991.

[14] Y. Gurevich and L. Harrington, “Trees, automata, and games,” in *Proc. 14th ACM Symp. Theory of Comp.*, pp. 60–65, ACM Press, 1982.

[15] E. Emerson and C. Jutla, “Tree automata, mu-calculus and determinacy (extended abstract),” in *Proc. 32nd IEEE Symp. Found. of Comp. Sci.*, pp. 368–377, IEEE Computer Society Press, 1991.

[16] N. Lynch and M. Tuttle, “Hierarchical correctness proofs for distributed algorithms,” in *Proceedings of the* $6^t h$ *ACM Symposium on Principles of Distributed Computing*, pp. 137–151, 1987.