Specification Enforcing Refinement for Convertibility Verification

Partha Roop, Alain Girault, Gregor Goessler, Roopak Sinha

Specification Enforcing Refinement for Convertibility Verification

Partha S Roop*, Alain Girault†, Roopak Sinha*, and Gregor Goessler†
*Department of Electrical and Computer Engineering, University of Auckland, New Zealand.
Email: p.roop,rsin077@ec.auckland.ac.nz
†POP ART project-team, INRIA Grenoble Rhône-Alpes and Grenoble University, France.
Email:alain.girault,gregor.goessler@inria.fr

Abstract

Protocol conversion deals with the automatic synthesis of an additional component, often referred to as an adaptor *or a* converter*, to bridge mismatches between interacting components, often referred to as* protocols*. A formal solution, called convertibility verification, has been recently proposed, which produces such a converter, so that the parallel composition of the protocols and the converter also satisfies some desired specification. A converter is responsible for bridging different kinds of mismatches such as* control*,* data*, and* clock *mismatches. Mismatches are usually removed by the converter by* disabling undesirable paths *in the protocol composition (similar to controllers in supervisory control of Discrete Event Systems (DES)).*

We generalize this convertibility verification problem by using a new refinement called specification enforcing refinement (SER) between a protocol composition and a desired specification. The existence of such a refinement is shown to be a necessary and sufficient condition for the existence of a suitable converter. We also synthesize automatically the converter if a SER refinement relation exists. The proposed converter is capable of the usual disabling actions to remove undesirable paths in the protocol composition. In addition, the converter can perform forcing actions *when disabling alone fails to find a converter to satisfy the desired specification. Forcing allows the generation of control inputs in one protocol that are not provided by the other protocol. Forcing induces* state-based hiding*, an operation not achievable using DES control theory.*

Index Terms

Protocol conversion, forced simulation, discrete controller synthesis.

1. Introduction

System-on-Chip (SoC) [6] design involves the interconnection of many pre-designed components, called IPs. While, a set of selected IPs may meet the functional requirements, their protocols may not be consistent, leading to several kinds of mismatches. The most common types of such mismatches are *control*, *data*, and *clock* mismatches. Control mismatches happen when the sequencing of control signals between protocols is inconsistent. Data mismatches happen when the data-widths of the two protocols differ and additional buffers are needed to manage loss-less data communication. Clock mismatches are common between IPs having different clock frequencies. The first approach to demonstrate the problem and some informal steps for a solution was proposed in [7]. Many techniques have been proposed since then to solve one or more of these incompatibilities using automated algorithmic techniques [8]–[10]. Their goal is to synthesize some additional glue logic, termed as a converter/interface/adaptor (from now on termed converter) to bridge these mismatches. While these techniques were automated, they failed to address several questions. These include how to formally model protocols and their interaction, and when such models are available, how to determine if a converter exists for a given set of protocols? Moreover, once the existence of a converter is determined, how to synthesize it? More recently, a set of formal techniques have been proposed [1]–[5] to address these questions. Table 1 compares these approaches over a range of features. The features listed as columns of Table 1 are: the *modelling language* for protocols, the language for describing desired *specifications*, *multiple protocols* (two and more than two), type of *conversion algorithm*, whether the approach can handle *uncontrollable events*, *event buffering*, whether *data-width mismatches* are handled, whether *clock mismatches* are handled, and finally the type of *control action* used. Among the proposed techniques, most approaches use Labeled Transition Systems (LTS) to describe both protocols and specifications, except [5] where CTL temporal logic is used for the specification part. Also, except the approach of [3]–[5] which use oversampling [11] to bridge clock mismatches, all other techniques ignore clock mismatches.

Central to protocol conversion is the use of a suitable controller that is used to remove undesirable paths in the protocol composition. This is done using the well known idea of *disabling* from Discrete Event Systems (DES) control theory [12]. Here, a supervisor or controller is synthesized

Partha Roop was supported by research and study leave from Auckland University and a research fellowship for experienced researchers from the Alexander von Humboldt foundation. Alain Girault was supported by a Marie Curie International Outgoing Fellowship within the 7th European Community Framework Programme.

Approach	Input (Protocols)	Input (spec.)	Multiple Protocols	Algorithm	Uncontrollable Signals	Buffering	Data	Multi-clock	Control Action
Kumar et al. [1]	LTS	LTS	✓	supervisory control	✓	×	×	×	disabling
Passerone et al. [2]	LTS	LTS	×	game-theoretic	×	✓	×	×	disabling
D'Silva et al. [3]	SPA	×	✓	refinement	×	✓	✓	✓	disabling
Tivoli et al. [4]	LTS	×	✓	controlled coverability	✓	✓	×	implicit	disabling
Sinha et al. [5]	LTS	CTL	✓	model-checking	✓	✓	✓	✓	disabling
SER Refinement	LTS	LTS	✓	refinement	✓	✓	×	×	disabling, forcing

Table 1. Features of various protocol conversion approaches

to control a *plant* so that the controlled system (composition of the controller and the plant) satisfies the desired *specification*. The role of the controller is to disable all *controllable* paths that violate the specification while leaving uncontrollable transitions untouched. In this domain, the plant and the specification are described as labelled transition systems (LTS) over an alphabet partitioned into controllable and uncontrollable events. While a converter is like a DES controller, the convertibility verification problem is not identical to DES supervisory control. Firstly, in convertibility verification there is a need to buffer events as an event generated by one protocol may be needed by another protocol at a later time. Secondly, there may be data and clock mismatches between protocols that are specific problems not addressed by DES supervisory control. Yet, both domains need to deal with controllable and uncontrollable events.

Kumar and Nelvagal proposed a formulation mapping the convertibility verification problem to the DES supervisory control problem [1] in a simplistic setting. Passerone et al. [2] developed a game theoretic formulation to solve the convertibility verification problem. Subsequently, in D'Silva et al. [3] a refinement based solution is developed for checking protocol compatibility. Recently, Sinha et al. [5] proposed a module checking based solution to the convertibility verification problem. Unlike earlier approaches, this approach bridges control, data-width and clock mismatches. Finally, Tivoli et al. [4] proposed a contrasting solution to the same problem that they termed as adaptor synthesis. This formulation ensures that timing constraints are met, events between protocols are adequately buffered, and that the composition is deadlock free. Also, while all other formulation presented in Table 1 use 1-place buffers for event buffering, the approach of [4] generalizes this to arbitrary N-place buffers.

Table 1 summarizes the features of all these formal approaches to convertibility verification. A key feature of the existing formal solutions (rows 1 to 5) is that they are based on disabling-based controllers. The last row of the table compares the existing solutions to the solution proposed in this paper. We extend the capability of the controller to allow, not only disabling actions, but also *forcing actions* [13]. Forcing actions are introduced to solve specific needs of the problem domain, namely the need for *state based hiding*, which is not possible using conventional disabling-only controllers. We will elaborate on this aspect through a motivating example in the next section.

2. Example and method overview

Figure 1 shows an overview of the convertibility verification problem. The actual protocols and their composition is shown in Fig. 2. We use CCS [14] style primed and unprimed symbols to indicate outputs and inputs respectively. E.g., in Figure 1, a is produced by the handshake protocol and read by the serial protocol. At each instant, a protocol can perform an input or output action (e.g., a or a'), or perform a T event, standing for the absence of input and output (written T_1 for P_1 and T_2 for P_2).

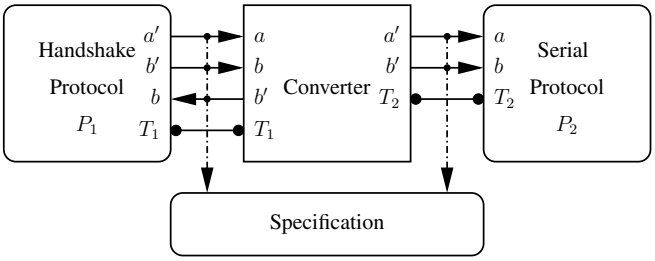


Figure 1. Overview of convertibility verification

Compared to the example in [2], the handshake protocol P_1 has an additional initialization step through an input b that is needed prior to establishing a handshake. Once initialized, P_1 outputs the signal a in the next instant. It then outputs the signal b after some random number of T events, modelled using self loops labelled by T_1. On the other hand, the serial protocol P_2 expects to read input b immediately in the instant following the reception of a. The synchronous parallel composition of P_1 and P_2, noted $P_1 \parallel P_2$, is also depicted in the same figure. We used synchronous parallel composition along with T events to indicate that a given protocol just delays. Other kinds of products, such as the interleaved parallel of CCS [14], were not considered to avoid non-determinism.

We provide a desired specification as shown in Fig. 3. This specification has a notion of completed transactions

through the introduction of *marked states* (depicted by a double circle). This specification enforces that *every input must be preceded by its corresponding output* (either in the same or a previous step). Moreover, *the transmission and reception of a must precede that of b.*

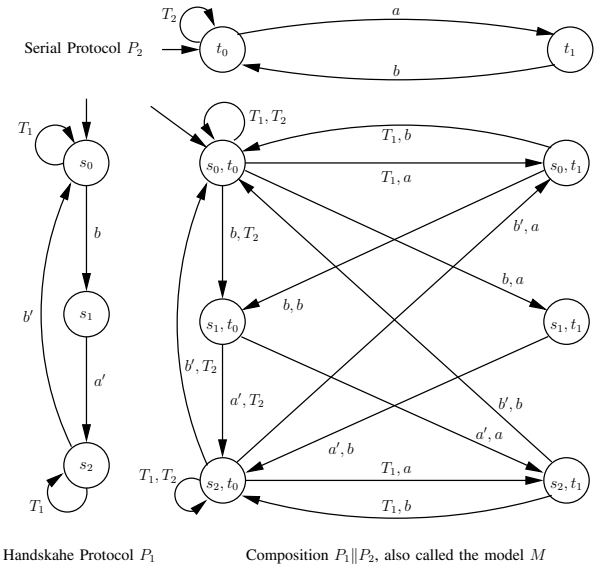


Figure 2. Handshake and serial protocols and their synchronous parallel composition

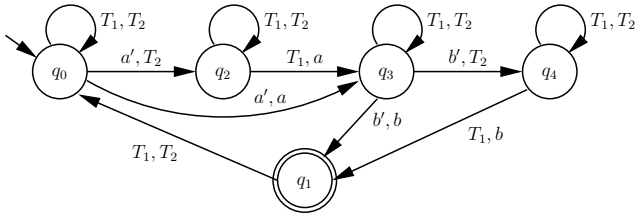


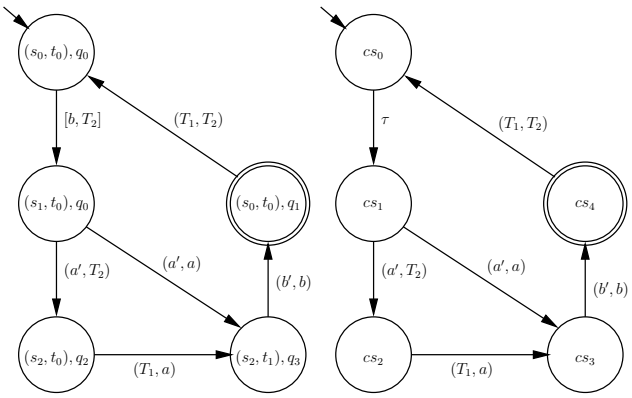
Figure 3. A desired specification with a marked state

Overview of the proposed methodology: The handshake-serial protocol pair is a mismatched protocol because of the following reasons:

- Initialization input b required by P_1 is not provided by P_2 at all.
- After an a is output by P_1, P_2 expects a b immediately while P_1 may produce b after any number of T_1 events.

Given such mismatching protocols and a desired specification, the goal of convertibility verification is to determine the existence of a converter that can bridge the mismatches, so that the overall system with the converter satisfies the desired specification. The converter can buffer inputs and forward them when necessary; it can also disable controllable paths in the composition. In the example of Fig. 2, P_2 requires a b immediately after having read the input a. Hence, when the protocol composition is in state (s_1, t_0), the converter reads the a produced by P_1, and forces P_2 to make a T_2 transition during this step. During the next transition, the converter transmits this a to P_2, while P_1 does a T_1 transition. We therefore say that the converter *buffers* the event a. In addition to buffering a, the converter also disables all other transitions of (s_1, t_0). In the proposed setting, we limit ourselves to 1-place buffers only for efficient synthesis considerations.

Note that by relying only on the usual converter actions (buffering and disabling), we would not be able to generate a converter for the handshake-serial protocol pair. This is because this protocol pair requires a (b, T_2) input in its initial state, while the specification allows the input of b only after the generation and consumption of a has been completed. Also note that the input (b, T_2) is not present in the converter's buffers initially. Thus, using conventional techniques based on DES supervisory control, we would fail to produce a correct converter. Hence, we propose a new way to control the protocol composition using a converter, as shown in Figure 4(a). The converter first *forces* the transition from (s_0, t_0) to (s_1, t_0) by generating the (b, T_2) input. These are inputs required by the protocols and are not present in the buffers. Forced inputs are marked within square brackets "$[\,]$" in the converter to distinguish them from other inputs that are either read from the converters buffers (events generated by the protocols in the past and not been consumed yet), or are directly read from the other protocol in the current instant. Since forced inputs are not produced by any of the protocols and have not been consumed from the buffers, they can be hidden by the composition of the protocols and the converter, thus satisfying the specification.



(a) Converter for handshake-serial example.

(b) Converter composed with the protocols.

Figure 4. A sample converter and its composition with the protocols

The composition of the converter with the protocols is shown in Fig. 4(b). Note that the forced transition from (s_0, t_0) to (s_1, t_0) is hidden in the composition (hence labelled by τ). Forcing enables *state based hiding*, different from global hiding achieved by DES supervisory control

(based on the hiding operator of CCS [14]). For instance, in the composed system of Fig. 4(b), the event b is hidden in the initial state $cs0$, while it is visible later in the state $cs3$. This is not directly achievable using DES controllers.

Forcing guides a controlled system to a successor state whenever the current state fails to satisfy the requirements of the specification. Like in DES supervisory control, where only controllable transitions may be disabled, only *forceable* transitions can be forced, and the user must specify a subset of inputs that can be forced. We elaborate on details of forcing in the next section.

We solve the convertibility verification problem as follows. A protocol pair (P_1, P_2) is said to satisfy a specification S when the language accepted by the synchronous parallel composition of the protocols, $L(P_1, P_2)$ is a subset the specification's one, $L(S)$. We only look at visible traces when checking this, because of forcing actions. Otherwise, the protocols are mismatching. In this case, we propose a new refinement relation from the composite protocols $P_1 \| P_2$ to the specification S. *We also show that the existence of such a refinement relation is a necessary and sufficient condition for the existence of the converter.* Finally, we provide an algorithm to synthesize the converter given such a relation. Our method for protocol conversion is based on DES supervisory control [12] and forced simulation [13]. While we have motivated the proposed method using the case of two protocols, the proposed approach generalizes straightforwardly to an arbitrary number of protocols.

3. Convertibility verification using specification enforcing refinement

3.1. Preliminaries

Error-free communication between protocols implies that traces in the protocol composition always respect the event sequencing described in the specification. We start by introducing the models of the protocols and of the specification.

We model protocols with labelled transition systems (LTS).

Definition 1: An LTS is a tuple $P = \langle \Sigma, Q, \rightarrow, q^\circ \rangle$, where Σ is the alphabet of actions, Q is the set of *states*, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the *transition relation*, and $q^\circ \in Q$ is the *initial state*. The transition relation is also written as $q \xrightarrow{a} q'$ iff $(q, a, q') \in \rightarrow$. The language $\mathcal{L}(P)$ is the set of all finite and infinite words generated by the LTS P.

In the case of a protocol, the alphabet Σ is partitioned into the set of input actions Σ_I, the set of output actions Σ_O, and the singleton $\{T\}$: $\Sigma = \Sigma_I \uplus \Sigma_O \uplus \{T\}$. The event T models the absence of input and output actions. Output events are emitted by the components while input events are received. We use primed symbols to represent output events and unprimed symbols to represent input events. Fig. 2 depicts the LTS of the handshake protocol to the left and that of the serial protocol at the top. The handshake protocol awaits for event b in state s_0 and outputs event a in state s_1. We term all the events labeling the outgoing transitions from a given state q as $Label(q) = \{a | \exists q' \text{ s.t. } q \xrightarrow{a} q'\}$.

Protocol interaction is defined with the synchronous parallel composition of LTSs. The parallel composition of handshake and serial protocols is depicted in Fig. 2.

Definition 2: Let $P_1 = \langle \Sigma_1, Q_1, \rightarrow_1, q_1^\circ \rangle$ and $P_2 = \langle \Sigma_2, Q_2, \rightarrow_2, q_2^\circ \rangle$ be two LTSs. The *synchronous product* of P_1 and P_2, noted $P_1 \| P_2$, is the LTS defined as:

$$P_1 \| P_2 \stackrel{\text{def}}{=} \langle \Sigma_1 \times \Sigma_2, Q_1 \times Q_2, \rightarrow, (q_1^\circ, q_2^\circ) \rangle$$

where $(q_1, q_2) \xrightarrow{(a,b)} (q_1', q_2')$ iff $q_1 \xrightarrow{a}_1 q_1'$ and $q_2 \xrightarrow{b}_2 q_2'$.

A specification describes the desirable interaction between protocols; it is represented as an LTS with a set of *marked states* [12]. Marked states specify completed transactions between protocols. A specification of the desired behaviour between the handshake and serial protocols is shown in Fig. 3; q_0 is its initial state and q_1 is its sole marked state.

Definition 3: Let $P_1 = \langle \Sigma_1, Q_1, \rightarrow_1, q_1^\circ \rangle$ and $P_2 = \langle \Sigma_2, Q_2, \rightarrow_2, q_2^\circ \rangle$ be two protocols. A specification over the pair P_1, P_2 is a tuple $S = \langle \Sigma_1 \times \Sigma_2, Q_S, \rightarrow_S, q_s^\circ, Q_S^m \rangle$ such that $\langle \Sigma_1 \times \Sigma_2, Q_S, \rightarrow_S, q_s^\circ \rangle$ is an LTS. The subset $Q_S^m \subseteq Q_S$ is the set of the marked states. The language $\mathcal{L}(S)$ is the set of all finite words that terminate in Q_S^m or infinite words that pass through states of Q_S^m infinitely often.

Thanks to the associativity of the synchronous product, our formalization generalizes straightforwardly to an arbitrary number of protocols. The only restriction is that communications between the protocols are point-to-point; formally, for each LTS protocol $P_i = \langle \Sigma_i, Q_i, \rightarrow_i, q_i^\circ \rangle$ such that $\Sigma_i = \Sigma_{Ii} \uplus \Sigma_{Oi} \uplus \{T\}$, and for each $e \in \Sigma_{Ii}$, there exists a unique protocol P_j such that $j \neq i$ and $e \in \Sigma_{Oj}$ (and vice-versa).

3.2. Refinement Relation

We now provide a solution for convertibility verification using a new refinement relation. In this section, we define the refinement that enforces a desired specification over a protocol composition to ensure error free communication of the two components. We use the well known idea as surveyed in [15] that an implementation (denoted M) respects a specification (denoted S) whenever a refinement exists from M to S. In this event, each observable behaviour of M is also an observable behaviour of S. We start by defining the notion of satisfaction of a specification for a given protocol composition model. This is similar in spirit to [16].

Definition 4: A model of two interacting protocols, $M = P_1 \| P_2$, satisfies a specification, denoted $M \models S$, iff $L(M) \subseteq L(S)$.

Since forcing leads to some actions being hidden in the resulting system, we weaken the classical definition of spec-

ification satisfaction of [12], [16] to deal with these hidden transitions (with internal τ actions) in the composition. This is similar in spirit to the approach taken in [17] where an interface protocol is introduced.

Definition 5: An LTS $M = (\Sigma, Q, \rightarrow, q^0)$ weakly satisfies a specification S, denoted $M \models_w S$ if and only if $L([M]) \subseteq L(S)$ where $L([M]) = \{\hat{\alpha} | \alpha \in L(M)\}$, and $\hat{\alpha}$ is the word obtained by deleting all τ actions from the word α.

Now, we introduce a new refinement relation, called *specification enforcing refinement* (SER), from $P_1 \parallel P_2$ to a specification S. We will subsequently show that this refinement guarantees the existence of a converter. For notational clarity, we represent $P_1 \parallel P_2$ as an LTS M that is equal to the composition of the two protocols: $M = \langle \Sigma_1 \times \Sigma_2, Q_1 \times Q_2, \rightarrow_M, (q_1^\circ, q_2^\circ) \rangle$ with $q_M \xrightarrow{\sigma} q'_M$ iff $(q_1, q_2) \xrightarrow{(a,b)} (q'_1, q'_2)$ and σ is a shorthand for (a, b).

Exactly like in the framework of DES supervisory control, we partition the set Σ_M into two subsets: the subset Σ_{Mc} of controllable events and the subset Σ_{Mu} of uncontrollable events. We introduce two additional subsets: the subset Σ_{Mb} of buffered events and the subset Σ_{Mf} of forceable events. While Σ_{Mc} and Σ_{Mu} are static sets (i.e., they don't change over time), Σ_{Mb} is a dynamic set since it depends on the current set of inputs in the converter's buffers. The set Σ_{Mf} is obtained by removing all current buffered inputs from the set of controllable inputs Σ_{Mc}, i.e., $\Sigma_{Mf} = \Sigma_{Mc} - \Sigma_{Mb}$. This is because buffered inputs have been produced in the environment; hence, they are visible and can't be hidden through forcing by the converter. Like Σ_{Mb}, Σ_{Mf} is also a dynamic set (i.e., its contents change over time). In the current setting, only outputs are uncontrollable, because the converter can not exert any influence on their generation. We now formally define these sets. We introduce a predicate $inBuff$ that returns true when a given input event is in the converter's buffers.

Definition 6: Given an LTS $M = P_1 \parallel P_2$, the subset Σ_{Mc} of the controllable events of M is $\Sigma_{Mc} = \{\sigma | \sigma = (a, b) \wedge a \in \Sigma_{I1} \wedge b \in \Sigma_{I2}\}$, the subset Σ_{Mu} of uncontrollable events is $\Sigma_{Mu} = \Sigma - \Sigma_{Mc}$, the subset Σ_{Mb} of buffered events is $\Sigma_{Mb} = \{\sigma | \sigma \in \Sigma_{Mc} \wedge inBuff(\sigma)\}$, and finally, the subset Σ_{Mf} of forceable events is $\Sigma_{Mc} - \Sigma_{Mb}$.

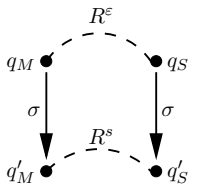
We now define the new refinement relation.

Definition 7: Let $M = \langle \Sigma_M, Q_M, \rightarrow_M, q_M^\circ \rangle$ and $S = \langle \Sigma_S, Q_S, \rightarrow_S, q_S^\circ, Q_S^m \rangle$ be the LTS of the protocol composition and the specification respectively. A relation $R \subseteq Q_M \times Q_S \times \Sigma_{Mc}^*$ is called a *specification enforcing refinement* (SER) from M to S, if the three conditions below hold. The notation $q_M \; R^s \; q_S$ is used as a shorthand for $(q_M, q_S, s) \in R$, where s is any word over Σ_{Mc} whose maximum length is bounded by $|Q_M|$.

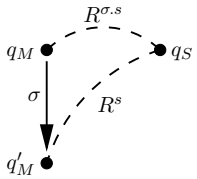
- **[Matched-state]**: If $q_M \; R^\varepsilon \; q_S$, then *there exists* $\sigma \in Label(q_M) \cap Label(q_S)$, $q'_M \in Q_M, q'_S \in Q_S$, and $s \in \Sigma_{Mc}^*$ such that $q_M \xrightarrow{\sigma} q'_M$ and $q_S \xrightarrow{\sigma} q'_S$, and $q'_M \; R^s \; q'_S$.
- **[Forced-state]**: If $q_M \; R^{\sigma s} \; q_S$ for $\sigma s \in \Sigma_{Mc}^+$, then $\sigma \in \Sigma_{Mf}$ and there exists $q'_M \in Q_M$ such that $q_M \xrightarrow{\sigma} q'_M$ and $q'_M \; R^s \; q_S$.
- **[Init-state]**: $q_M^\circ \; R^s \; q_S^\circ$ for some $s \in \Sigma_{Mc}^*$.

According to Definition 7 above, there are two ways how states $q_M \in Q_M$ and $q_S \in Q_S$ can be related via an SER R:

1. q_M and q_S are *directly related* if q_M has at least one transition having the same label as a transition from q_S (a matching transition pair). Moreover, for the matching transition pair $q_M \xrightarrow{\sigma} q'_M$ in M and $q_S \xrightarrow{\sigma} q'_S$ in S, the successor states q'_M and q'_S are also related via some forcing sequence $s \in \Sigma_{Mc}^*$. Hence, $q_M \; R^\varepsilon \; q_S$.



2. q_M and q_S are *related via some forcing sequence* $\sigma.s$ if there exists a successor state q'_M in M such that q'_M is reachable from q_M via a *forceable event* σ where q'_M and q_S are related via R. In this case, $q_M \; R^{\sigma.s} \; q_S$.



These possibilities are formalized in the first two conditions of Definition 7. In addition, the start states are required to be related via some forcing sequence s, which corresponds to the third condition. Note that the length of the forcing sequence is bounded by the size of the model. This restriction is needed to ensure that we have only bounded forcing steps. Unbounded forcing steps are possible if the converters have *forcing cycles*. We forbid such converters since our application domain requires converters to be synthesizable.

Between the handshake-serial example in Fig. 2 (M) and the specification shown in Fig.3 (S), an SER exists as follows:

$$R = \{((s_0, t_0), q_0, [b, T_2]), ((s_1, t_0), q_0, \varepsilon), ((s_2, t_0), q_2, \varepsilon), ((s_2, t_1), q_3, \varepsilon), ((s_0, t_0), q_1, \varepsilon)\}$$

In general, there may be many SER relations between M and S. For example, $R' = \{((s_0, t_0), q_0, \varepsilon,)\}$ is also a valid SER relation between M and S.

3.3. Marked compatibility

The goal of convertibility verification is to determine the conditions under which a suitable converter between M and S exists. Note that an SER refinement between M and S alone doesn't guarantee the existence of such a converter. For example, consider the model M of the composite protocols as shown in Fig. 2 and the specification as shown in Fig. 3. We can define an SER $R' = \{((s_0, t_0), q_0, \varepsilon)\}$. The corresponding *trivial converter* will just enable the self-loop transition in the initial states of the two protocols. However,

such a converter doesn't ensure completed transactions in the protocols. Marked states (e.g., state q_1 in Fig. 3) in the specification are used to represent completed transactions.

To prevent synthesis of trivial converters, we define marked compatible SERs.

Definition 8: Let R be an SER relation between M and S. A path $q_M \rightarrow q_{M_1} \rightarrow q_{M_2} ... \rightarrow q_{M_n}$ in M is a compatible path to a path $q_S \rightarrow q_{S_1} \rightarrow q_{S_2} ... \rightarrow q_{S_n}$ in S if $(q_M, q_S, s) \in R$ for some $s \in \Sigma^*_{Mc}$ and for all $i \in [1..n]$: $(q_{M_i}, q_{S_i}, s_i) \in R$ for some $s_i \in \Sigma^*_{Mc}$.

Definition 9: An SER relation R between M and S is marked compatible if for every $(q_M, q_S, s) \in R$ there exists $(q'_M, q'_S, s') \in R$ such that there exists a path from q_M to q'_M and a *compatible path* from q_S to $q'_S \in Q^m_S$, i.e., q'_S is a marked state of S.

Definition 10: Let M and S be a model of protocol composition and a specification, respectively. $M \lesssim_{\text{SER}} S$ if there exists a specification enforcing refinement from M and S that is marked compatible.

3.4. Converters

We now synthesize converters between protocols. A converter is an LTS whose role is to appropriately guide the protocols so that the composite system satisfies the desired specification. The role of the converter is to act as an intermediary so that all protocol communications are consistent with the specification. We only consider converters with a bounded number of states since our application domain is embedded systems (SoCs) and in this domain it is important that we are able to synthesize the generated converters. In our framework, a converter can perform the following actions:

1) Disabling a transition of the protocols: Remove undesirable communication paths that violate the specification. This operation is identical to the controllers in DES [12]. The disabled transitions must be controllable.
2) Forcing a transition of the protocols: Automatically guide the protocols to a successor state from its current state so that the future state is consistent with a specification state. This is done by generating the suitable forceable inputs on a path.
3) Buffering a communication between the protocols: If a given input generated by one of the protocols cannot be consumed by the receiving protocol, a converter can buffer this event so that it can be forwarded in the future.

3.5. Converter Synthesis from an SER Relation

Converters can be derived automatically once an SER is established between M and S. Given an SER R, the states of the converter Q_C are exactly the elements of R. We now formalize the relationship between an SER relation and a corresponding converter. We start by defining *precisely-forced* SERs so as to ensure that forcing is always performed in a unique fashion from any *forced state*.

Definition 11: An SER relation R is *precisely-forced* iff $(q_M, q_S, s_1) \in R$ and $(q_M, q_S, s_2) \in R$ implies that $s_1 = s_2$.

Given an SER R, a precisely-forced SER R' can be automatically derived from R and always exists. From a precisely-forced SER, we now build a converter that derives from it:

Definition 12: Let R be a precisely-forced SER between a model M and a specification S. The converter derived from R is the LTS $C_R = \langle \Sigma_M \cup [\Sigma_{Mc}],\ R,\ \rightarrow_C,\ ((q^\circ_M, q^\circ_S), s_0) \rangle$, where $[\Sigma_{Mc}] = \{\, [\sigma] \mid \sigma \in \Sigma_{Mc} \,\}$ and $[\sigma]$ denotes the forced action over event σ, and $\rightarrow_C$ is defined by the following two rules:

- **[Matched-event]**: If $(q_M, q_S, \varepsilon) \in R \wedge (q'_M, q'_S, s') \in R \wedge q_M \xrightarrow{\sigma} q'_M \wedge q_S \xrightarrow{\sigma} q'_S$, then $(q_M, q_S, \varepsilon) \xrightarrow{\sigma} (q'_M, q'_S, s')$.
- **[Forced-event]**: If $(q_M, q_S, \alpha.s) \in R \wedge q_M \xrightarrow{\alpha} q'_M$, then $(q_M, q_S, \alpha.s) \xrightarrow{[\alpha]} (q'_M, q_S, s)$.

For the handshake-serial protocol pair shown in Fig. 2 and the specification S shown in Fig. 3, a converter generated by our approach is shown in Fig. 4(a). It first forces the transition from (s_0, t_0) to (s_1, t_0) by generating the events $[b, T_2]$. Subsequently, it reads the a produced by P_1 and buffers it while allowing P_2 to remain in its initial state through a T_2 transition. It then forwards the buffered a to the P_2 while allowing P_1 to remain in its current state through a T_1 transition. Note that there is also the choice of directly allowing the (a', a) transition in the state $cs1$ instead of first buffering a and later forwarding a. Hence, the generated converter keeps all possibilities.

Having established the relationship between a given SER and the associated converter, we now define well-formed converters. Well-formed converters ensure that protocols always complete their transactions.

Definition 13: Let R be a SER between a model M and a specification S. A converter $C = \langle \Sigma_C, Q_C, \rightarrow_C, q^\circ_C \rangle$ derived from R is said to be *well-formed* if the two following conditions hold:

- **[Forced-alone]**: For all $q, q' \in Q_C$ and $\alpha \in \Sigma_C$ such that $q \xrightarrow{[\alpha]}_C q'$, if $q \xrightarrow{\sigma}_C q''$ and $q \xrightarrow{[\beta]}_C q'''$ for some $q'', q''' \in Q_C$ and some $\sigma, \beta \in \Sigma_C$, then $\sigma = [\alpha]$, $q' = q''$, $\beta = \alpha$, and $q' = q'''$.
- **[Marked-path]**: For any state $q \in Q_C$, there always exists a path to a state $q' \in Q_C$ such that $q' = (q_M, q_S, s)$ and $q_S \in Q^m_S$.

The state graph of a well-formed converter has only one successor for states where forcing is performed. Other states may have more than one successor. Moreover, from every state of a well formed converter, a marked state can always be reached. A state in the converter is called a marked state if the corresponding component of R is of the form

(q_M, q_S, s) such that q_S is a marked state. It is easy to note that any converter C derived from a deterministic and marked compatible SER is always well-formed.

In our framework, event buffering is achieved thanks to the state space of the converter. This is the case of the event a in the converter of Figure 4.

Lemma 1: Let R be a precisely-forced SER between a model M and a specification S, and let C_R be the converter derived from R. If R is marked compatible, then C_R is well-formed.

Proof: Let $C_R = \langle \Sigma_C, Q_C, \rightarrow_C, q_C^\circ \rangle$ be the converter derived from R.

Proof of Condition **[Forced-alone]**: Let $q, q' \in Q_C$ and $\alpha \in \Sigma_C$ such that $q \xrightarrow{[\alpha]}_C q'$. We prove by contradiction that $\nexists\, q'' \neq q' \in Q_C$ and $\sigma \neq \alpha \in \Sigma_C$ such that $q \xrightarrow{\sigma}_C q''$. Since $[\alpha]$ is a forced action, Rule **[Forced-event]** implies that $q = (q_M, q_S, \alpha.s) \in R$. Furthermore, since $q \xrightarrow{\sigma}_C q''$, Rule **[Matched-event]** implies that $(q_M, q_S, \varepsilon) \in R$. Hence, according to Definition 11, we conclude $\alpha.s = \varepsilon$, which is a contradiction.

Similarly, it can be show by contradiction that $\nexists\, q''' \neq q' \in Q_C$ and $\beta \neq \alpha \in \Sigma_C$ such that $q \xrightarrow{[\beta]}_C q'''$. This follows directly from the fact that R is precisely-forced.

Proof of Condition **[Marked-path]**: Direct consequence of Definition 9. □

We now define the product operation for composing a converter C with a pair of protocols represented as an LTS M.

Definition 14: Let $C = \langle \Sigma_C, Q_C, \rightarrow_C, q_C^\circ \rangle$ and $M = \langle \Sigma_M, Q_M, \rightarrow_M, q_M^\circ \rangle$ be a converter and a model respectively. The *forced composition* $C \,/\!/\, M$ of C and M is $C \,/\!/\, M \stackrel{\text{def}}{=} \langle \Sigma_M \cup \{\tau\}, Q_C \times Q_M, \rightarrow, (q_C^\circ, q_M^\circ) \rangle$, where $\rightarrow$ is defined by the following two rules:

- **[Tau-trans]**: $(q_C, q_M) \xrightarrow{\tau} (q'_C, q'_M)$ if $q_C \xrightarrow{[\alpha]}_C q'_C$ and $q_M \xrightarrow{\alpha}_M q'_M$ for some $\alpha \in \Sigma_{Mf}$.
- **[Event-trans]**: $(q_C, q_M) \xrightarrow{\sigma} (q'_C, q'_M)$ if $q_C \xrightarrow{\sigma}_C q'_C$ and $q_M \xrightarrow{\sigma}_M q'_M$ for all $\sigma \in \Sigma_M$.

Lemma 2: The forced composition $C /\!/ M$ is deterministic if both C and M are deterministic and if the converter C is well-formed.

Proof: The proof follows directly from the fact that both C and M are deterministic and from the Rules **[Tau-trans]** and **[Event-trans]**. □

The next result states that a marked compatible SER relation between M and S is a necessary and sufficient condition for the existence of a correct converter.

Theorem 1: Let M and S be deterministic LTSs of the model and the specification respectively. There exists a well-formed and deterministic converter C such that $C /\!/ M \models_w S$ if and only if $M \lesssim_{\text{SER}} S$.

Proof: **Sufficient Condition:** The proof is constructive. Given $M \lesssim_{\text{SER}} S$, there exists a precisely-forced and marked compatible SER R. We can construct a converter C using Definition 12. Since R is marked compatible C is well formed (Lemma 1). Also, C is deterministic since both M and S are deterministic. Thus, $C /\!/ M$ is also deterministic. Now it is easy to see that all τ-projected traces of $C /\!/ M$ are contained in the trace set of S.

Necessary Condition: Given a well-formed converter C such that $C /\!/ M \models_w S$, we need to prove that $M \lesssim_{\text{SER}} S$. Since $L([C /\!/ M]) \subseteq L(S)$ and as C is well-formed and C, M and S are deterministic, this result follows. □

4. Prototype tool and results

A local, on-the-fly tableau construction algorithm, similar to [18], [19], is used for converter synthesis. The recursive algorithm is shown in Alg. 1 and is described as follows.

A tableau is a table of assertions which is organized as a graph in the proposed algorithm. Each assertion is of the form ($\mathtt{q}_M$,buf, $\mathtt{q}_S$) which is true when the states $\mathtt{q}_M$ and $\mathtt{q}_S$ (of the synchronous composition M of the given protocols and specification S respectively) are related via a SER. The set `buf` is used to describe the events contained in the buffers of the converter (being generated) under which the assertion must hold. Each assertion is described as a vertex of the graph (tableau), called a *node*. A node's attributes are the states $\mathtt{q}_M$ and $\mathtt{q}_S$, the set `buf`, and a variable *type* that can be either "Disabling" or "Forcing". The type is used to avoid forcing cycles in the converter and is described later. A node can have outgoing edges to other nodes in the graph, which are called its *children nodes*. The assertion represented by a node holds if and only if the *sub-assertions* represented by its children nodes hold.

The inputs to the recursive algorithm `checkSER` are states $\mathtt{q}_M$ and $\mathtt{q}_S$ of a model and a specification, a set `buf` of signals assumed to be contained in the converter's buffers, and `h`, an ordered list of nodes that have been visited before the current call to `checkSER` is made. The first call to the algorithm is made with the parameters q_m°, $\mathtt{buf} = \emptyset$, q_s°, and $\mathtt{h} = \emptyset$. This forms the initial assertion of the tableau and requires that the initial states q_m° and q_s° of the model and specification be related to each other via a SER relation under the assumption that the converter contains no events in its buffers initially (`h` is empty because no previous call has been made).

The algorithm, given the arguments $\mathtt{q}_M$, `buf`, $\mathtt{q}_S$, and `h`, proceeds as follows. It first creates a node `curr` with the attributes $\mathtt{q}_M$, `buf`, and $\mathtt{q}_S$. Next, it checks if a similar node `anc` is contained in `h` (line 3). If such a node `anc` is found, this means that a cycle in the converter (being generated) is found. In this case, the algorithm checks if the trace from `anc` to `curr` contains a node `nxt` where $\mathtt{nxt}.q_S$ is a final state in the specification ($q_S \in Q_S^m$) and also that $\mathtt{nxt}.q_M$ and $\mathtt{nxt}.q_S$ are related directly (via ε). If such a node is found, the cycle is acceptable (is marked compatible and has no forcing cycles), and the algorithm returns `anc`

(line 7). Otherwise, it returns **nil** signifying that the cycle is either not marked compatible or contains forcing cycles (line 11).

Algorithm 1 node checkSER(q_M, buf, q_S, h)

```
 1: curr := createNode(q_M, buf, q_S);
 2: //FINITIZE: Check for Cycles
 3: if curr = anc for some anc ∈ h then
 4:    nxt := anc; {allow only compatible, non-forcing cycles}
 5:    while nxt != nil do
 6:       if nxt.type = Disabling and nxt.q_S ∈ Q_S^m then
 7:          return anc;
 8:       end if
 9:       nxt := h.getNodeAfter(nxt);
10:    end while
11:    return nil;
12: end if
13: h' := h.append(curr);
14: //DISABLING:
15: curr.type := Disabling;
16: common := Label(q_M) ∩ Label(q_S); {can be matched}
17: uncSet := Label(q_M) ∩ Σ_Mu; {must be matched}
18: if uncSet ⊆ common then
19:    for each a ∈ uncSet do
20:       q_Ma := q'_M : q_M -a-> q'_M;
21:       q_Sa := q'_S : q_S -a-> q'_S;
22:       nxt := checkSER(q_Ma, adjustBuf(buf, a), q_Sa, h');
23:       if nxt = nil then
24:          curr.removeAllChildren();
25:          Jump to line 41;
26:       else
27:          curr.addChild(a,nxt);
28:       end if
29:    end for
30:    conSet := common − uncSet;
31:    for each a ∈ common such that a ∈ buf do
32:       q_Ma := q'_M : q_M -a-> q'_M;
33:       q_Sa := q'_S : q_S -a-> q'_S;
34:       nxt := checkSER(q_Ma, adjustBuf(buf, a), q_Sa, h');
35:       if nxt != nil then
36:          curr.addChild(a, nxt);
37:       end if
38:    end for
39:    return curr;
40: end if
41: //FORCING:
42: curr.type := Forcing;
43: if Label(q_M) ∩ Σ_Mu = ∅ then
44:    //If state is forcible
45:    for each forcible event a ∈ Label(q_M) s.t. a ∉ buf do
46:       q_Ma := q'_M : q_M -a-> q'_M;
47:       nxt := checkSER(q_Ma, buf, q_S, h');
48:       if nxt!= nil then
49:          curr.addChild([a], nxt);
50:          return curr;
51:       end if
52:    end for
53: end if
54: return nil;
```

In case the node curr needs to be expanded further (no cycle is detected), the algorithm adds it to h to form a new history stack h′. The algorithm then attempts to match curr.q_M and curr.q_S using either the null sequence ε (lines 14–40) or some forcing sequence a (lines 41–53). If both attempts fail, the algorithm returns **nil** (line 54).

In order to match curr.q_M and curr.q_S over ε (by possibly disabling one or more transitions in curr.q_M), the algorithm sets curr.type to "Disabling" and computes the sets common and uncSet (lines 15–17). common contains all events that trigger transitions in both curr.q_M and curr.q_S whereas uncSet contains events that trigger uncontrollable transitions in curr.q_M. A converter must enable all transitions triggered by uncSet in curr.q_M and each enabled transition must match a transition curr.q_S. Hence, the states cannot be matched if uncSet is not a subset of common because one or more uncontrollable transitions in curr.q_M cannot be matched by any transition in curr.q_S. Otherwise, the algorithm checks if the states q_{M_a} and q_{S_a} reached from curr.q_M and curr.q_S via each uncontrollable event $a \in$ uncSet are related to each other (lines 19–29). If for any a, q_{M_a} and q_{S_a} are not related (checked via a recursive call to checkSER), curr.q_M and curr.q_S cannot be related to ε. If however all q_{M_a} and q_{S_a} pairs are related, the algorithm then checks if any other states reached via *controllable* events (events in the set common-uncSet) are also related to each other (lines 30–38). Each pair that matches is retained as a (successful) sub-assertion. The algorithm then returns curr to indicate that matching over ε is successful.

In case matching over ε fails, the algorithm attempts to apply forcing (lines 41–53). It first resets curr.type to "Forcing" and checks if the state curr.q_M is indeed forcible (that is, it has no uncontrollable transitions). Next, it checks if each transition label a of curr.q_M can be forced such that the resulting state q_{M_a} is related to curr.q_S. Note that an event a is considered only if it is not present in buf. The algorithm returns success when such an event a is found. The resulting (successful) node returned by the recursive call to checkSER is added as the sole child of curr.

In case both disabling and forcing tests fail, the node returns failure (**nil**). The worst-case complexity of the algorithm is $O(|Q_M|^2 \times |Q_S|^2 \times 2^{|\Sigma_{Mc}|})$ where $|Q_M|$ and $|Q_S|$ are the sizes of the state sets of M and S, and $|\Sigma_{Mc}|$ is the size of the controllable event set of M.

The algorithm is intuitively described by using the tableau shown in Fig. 5 (generated for the handshake-serial example). The inputs to the algorithm are the initial states (s_0, t_0) and q_0 of the model and the specification, and an empty *set of buffered events* and an empty set of previously visited nodes. These inputs form the initial assertion (node) A1 of the tableau, which is recursively broken down into sub-assertions (children nodes) using the disabling and forcing rules. A node returns success if its children return success. An infinite resolution of nodes into children nodes is prevented by termination conditions to ensure that duplicate

nodes are not resolved further. For example, in Fig. 5, the nodes A2 and A8 are not resolved further because they are identical to the previously processed node A1. A2 returns failure because the path from A1 to A2 does not contain any node corresponding to the marked state q_1, whereas A8 returns success because the path from A1 to A8 contains a node (A7) that corresponds to q_1. A5 is resolved in the same manner as A6, and returns success because (A6) has already returned success. The algorithm exits when the initial assertion (A1) returns success to indicate that a successful tableau has been generated.

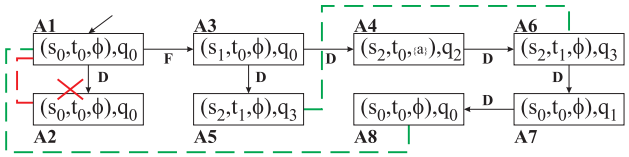


Figure 5. Tableau for the handshake-serial example

Each node in a successful tableau corresponds to a unique state in the converter. For example, each node in the tableau shown in Fig. 5 corresponds to a unique state in the converter shown in Fig. 4(a), (and the initial node A1 corresponds to the initial state of the converter). If `checkSER` succeeds, the returned node, called the root node of the tableau, is passed to the procedure `extractConverter` to generate the converter. `extractConverter` operates as follows. It first creates a data structure MAP that maps nodes in the tableau to states in the converter (MAP is initially empty). It then calls the recursive algorithm `extract` and passes to it the root node of the tableau.

Algorithm 2 `STATE extractConverter(root)`

```
1: Create new map MAP
2: initialState = extract(root);
3: return initialState
```

In `extract`, we first checked if the argument `curr` is contained in the MAP. If so, the corresponding converter state is returned. Otherwise, a new converter state q_C is created and map is adjusted to remember the correspondence between `curr` and q_C (line 4–5). Then, each child node `nxt` of `curr` is processed by calling `extract` to obtain its corresponding state q'_C and a transition from q_C to q'_C is created.

Algorithm 3 `STATE extract(curr)`

```
1: if curris present in MAP then
2:    return map.get(curr)
3: end if
4: create new converter state q_C
5: MAP.put(curr, q_C)
6: for each nodenxts.t. curr→ a nxt do
7:    State q'_C = extract(nxt)
8:    add transition q_C -a-> q'_C
9: end for
10: return MAP.get(curr)
```

Implementation Results

Tab. 2 shows a set of results obtained by executing the SER algorithm over some well-known conversion problems described in literature [1]–[3], [8], [19]. Each entry in the table describes the protocols and specification involved and the types converters obtained by using classical approaches and the SER conversion algorithm (**D**=disabling, **DF**=disabling and forcing). Problem 1 is the handshake-serial problem presented in [2] while problem 1A is an extension of problem 1 that involves an extra input transition in the handshake protocol (see Fig. 2). Problems 2, 3, 4, and 5, that are taken from other articles on protocol conversion, are extended in similar fashion to problems 2A, 3A, 4A, and 5A. While classical techniques can handle problems 1, 2, 3, 4, and 5 only, we could generate converters for these problems as well as their variants 1A, 2A, 3A, 4A, and 5A. Although the implementation does not address the question of finding *optimal* converters (those with minimum number of forcing steps), it can generate *all possible* converters. The algorithm not only finds converters where previous approaches fail, it also preserves the full design space by finding all possible solutions.

5. Conclusions

Protocol conversion is required while creating a complex system (such as a System-on-Chip) from pre-designed components (called IPs) which have mismatching communication protocols. Convertibility verification automatically determines if a suitable glue-logic, called a converter, exists to bridge such mismatches. Converters are inspired by controllers from DES supervisory control theory and hence bridge mismatches through *disabling* of undesirable communication paths while also performing additional control actions such as event buffering. This paper presents a more generalized converter synthesis technique that performs *forcing of actions* in addition to the conventional disabling. Forcing actions are used to hide extra control sequences that are required by the protocols but not by the desired specification. Forcing induces *state-based hiding* that is not possible using standard hiding operators in DES supervisory control.

We have proposed a new refinement relation, called specification enforcing refinement (SER), between a given protocol composition and a desired specification. We have also shown that the existence of this relation is a necessary and sufficient condition for the existence of a suitable converter that enforces the desired specification over the protocols. The proposed approach generalizes existing approaches to convertibility verification, and we have demonstrated it by finding converters for many protocol mismatches that can't

Problem	Specification/ Properties	Classical converter types	SER converter types
1. Handshake-serial [2]	I/O sequencing	**D**	**D**
1A. Adapted handshake-serial	I/O sequencing	**-**	**DF**
2. ABP receiver, NS sender [8]	No packet loss	**D**	**D, DF**
2A. Adapted ABP receiver, NS sender	No packet loss	**-**	**D, DF**
3. ABP sender, NS receiver [1]	No packet loss	**D**	**D, DF**
3A. Adapted ABP sender, NS receiver	No packet loss	**-**	**D, DF**
4. Handshake-Pipeline [3]	Correct data exchange	**D**	**D, DF**
4A. Adapted Handshake-Pipeline	Correct data exchange	**-**	**D, DF**
4. Producer-Consumer [19]	No over/under flows	**D**	**D, DF**
4A. Adapted Producer-Consumer	No over/under flows	**-**	**D, DF**

Table 2. Implementation Results

be bridged using existing techniques. Future work will involve extending the formulation to handle data-width and clock mismatches, and finding optimal converters. We will then apply the algorithm to solve protocol mismatches in real SoCs based on standard buses such as the AMBA.

References

[1] R. Kumar and S. Nelvagal, "Protocol conversion using supervisory control techniques," in *IEEE International Symposium on Computer-Aided Control System Design*, 1996, pp. 32–37.

[2] R. Passerone, L. de Alfaro, T. Henzinger, and A. Sangiovanni-Vincentelli, "Convertibility verification and converter synthesis: Two faces of the same coin," in *International Conference on Computer Aided Design (ICCAD)*, 2002. [Online]. Available: http://www.gigascale.org/pubs/217.html

[3] V. d'Silva, S. Ramesh, and A. Sowmya, "Synchronous protocol automata: A framework for modelling and verification of SoC communication architectures," *IEE Proc. Computers & Digital Techniques*, vol. 152, no. 1, pp. 20–27, 2005.

[4] M. Tivoli, P. Fradet, A. Girault, and G. Goessler, "Adaptor synthesis for real-time components," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Braga, Portugal, 2007, pp. 185–200.

[5] R. Sinha, P. S. Roop, S. Basu, and Z. Salcic, "Multi-clock SoC design using protocol conversion," in *Design Automation and Test in Europe (DATE)*, April 2009.

[6] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd, *Surviving the SoC Revolution: A guide to platform based design*. Kluwer Academic, 1999.

[7] P. Green, "Protocol conversion," *IEEE Trans. on Communications*, vol. 34, no. 3, pp. 257–268, March 1986.

[8] S. Lam, "Protocol conversion," *IEEE Trans. on Software Engineering*, vol. 14, no. 3, pp. 353–362, 1988.

[9] G. Borriello, "A new interface specification methodology and its application to transducer synthesis," Ph.D. dissertation, University of California, Berkeley, 1988.

[10] S. Narayan and D. Gajski, "Interfacing incompatible protocols using interface process generation," in *Design Automation Conference (DAC)*, 1995, pp. 468–473.

[11] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer International Series in Engineering and Computer Science, 1994.

[12] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, Jan. 1989.

[13] P. Roop, A. Sowmya, and S. Ramesh, "Forced simulation: A technique for automating component reuse in embedded systems," *ACM Trans. on Design Automation of Electronic Systems*, vol. 6, no. 4, pp. 602–628, 2001.

[14] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989.

[15] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Trans. on Design Automation of Electronic Systems*, vol. 4, no. 2, pp. 123–193, 1999.

[16] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991. [Online]. Available: citeseer.ist.psu.edu/abadi91existence.html

[17] N. Maretti, "Mechanized verification of refinement," in *International Conference on Theorem Provers in Circuit Design - Theory, Practice and Experience (TPCD)*. London, UK: Springer-Verlag, 1994, pp. 185–202.

[18] R. Cleaveland, "Tableau-based model checking in the propositional mu-calculus," *Acta Informatica*, vol. 27, no. 9, pp. 725–747, 1990.

[19] R. Sinha, P. S. Roop, S. Basu, and Z. Salcic, "An approach for resolving control and data mismatches in SoCs," School of Engineering, University of Auckland, Report 667, 2008, www.ece.auckland.ac.nz/∼roop/documents/tech667.pdf.