A Formal Approach To The Protocol Converter Problem

Karin Avnit[†], Vijay D'Silva[§], Arcot Sowmya[†], S. Ramesh[‡], Sri Parameswaran[†]
[†]The University of NSW, Sydney, Australia. {kavnit,sowmya,sridevan}@cse.unsw.edu.au
[§]ETH, Zurich, Switzerland. vdsilva@inf.ethz.ch
[‡]GM India Science Lab, Bangalore India. rameshari1958@gmail.com

Abstract

In the absence of a single module interface standard, integration of pre-designed modules in System-on-Chip design often requires the use of protocol converters. Existing approaches to automatic synthesis of protocol converters mostly lack formal foundations and either employ abstractions that ignore crucial low level behaviors, or grossly simplify the structure of the protocols considered. We present a state-machine based formal model for bus based communication protocols, and precisely define protocol compatibility, and correct protocol conversion. Our model is expressive enough to capture features of commercial protocols such as bursts, pipelined transfers, wait state insertion, and data persistence, in cycle accurate detail. We show that the most general, correct converter for a pair of protocols, can be described as the greatest fixed point of a function for updating buffer states. This characterization yields a natural algorithm for automatic synthesis of a provably correct converter by iterative computation of the fixed point. We report our experience with automatic converter synthesis between widely used commercial bus protocols, such as AMBA AHB, ASB, APB, and OCP, considering features which are beyond the scope of current techniques.

1. Introduction

Hardware module reuse is a standard solution to deal with the increasing complexity of chip architectures and growing pressure to reduce time to market. Much research has been dedicated to the *converter synthesis problem* of SoC communication - the synthesis of converters to mediate between incompatible protocols. Practical converter synthesis algorithms are crucial to realizing the dream of "plug-n-play" style SoC design. Despite attempts at automation, converter synthesis is still performed manually, consuming development and verification time and risking human error.

A comprehensive research effort toward correct converter synthesis must precisely answer three questions: (a) What is a protocol? (b) When are two protocols compatible? and (c) What is a correct converter for a pair of incompatible protocols? A precise protocol definition is required to gauge the practical utility of the model. An answer to the second question must define protocol compatibility and provide an algorithm to check it. A definition of converter correctness leads to algorithms for converter verification and provably correct converter synthesis. Precise definitions of this form are largely absent in the literature, so existing algorithms cannot be checked for correctness and thus do not provide the high quality guarantee required in electronic design. Finally, existing approaches fail to model protocols accurately, or employ high levels of abstractions that make automatic translation to HDL (Hardware Description Languages) impossible.

Our work is aimed at automatic synthesis of a *provably correct* protocol converter. We present a simple and powerful FSM based formal model for on-chip communication protocols, that enables for the first time detailed modeling of complex commercial bus protocols. We propose comprehensive definitions of protocol compatibility and of correct protocol converters, and derive algorithms for compatibility checking as well as converter synthesis. We report on our experiments with different commercial protocols.

1.1. Related Work

The problem of automatic converter synthesis for incompatible protocols has been addressed in the literature from different perspectives. We focus on work done in the context of hardware design using FSM-based models.

In early work [2], protocols were represented as state machines and their cross product was used to construct a converter. This work was highly innovative but preliminary. This approach was later extended in [3,6,7] and is the foundation of our work. In [6,7] a formalism for modeling protocols using synchronous FSMs and an algorithm for wrapper synthesis are proposed. It distinguishes between control and data signals, and methods for dealing with mismatched data types and clock periods are suggested but not integrated into the given algorithm. In [3], a product of FSMs is again used to construct protocol converters, and the product is optimized to increase bandwidth. An alternative to converter synthesis is to use a standard communication scheme and to map disparate protocols into this scheme, as presented in [12] and in [8]. Such *template-based solutions* are not specific to the protocols being interfaced and hence, not optimal, and in some cases might not be practical. A third approach is to decompose protocols into smaller operations and combine operations to obtain a converter, as presented in [9] and recently in [13]. Passerone et al. [11] specify mismatched synchronous protocols as regular expressions

978-3-9810801-3-1/DATE08 © 2008 EDAA

and in later work [10] attempted a game theory formalization. No algorithm is presented, so it is unclear how the technique can be applied to arbitrary protocols.

All approaches discussed above either model protocols at a high level of abstraction, or with several simplifying restrictions, and thus, preclude completely automatic synthesis. No existing work distinguishes between control and data paths, thus the obtained converter is usually large and impractical. Our formalism allows for precise, cycle-accurate modeling of protocols and converters with distinct control and data paths. We faithfully model complicated commercial protocols, and converters can easily (even automatically) be translated into HDL.

In most existing work, the definition of protocol compatibility is neglected or incorrectly assumed to be trivial, and protocol conversion is discussed without considering when such a converter is needed, or even what criteria a *correct converter* should satisfy. We define protocol compatibility as constraints to ensure continuous and correct data transfer between two protocols, and formalize correct protocol conversion based on this notion. Relying on the definitions and formalism, we propose algorithms for checking compatibility and for automatic synthesis of protocol converters.

1.2. Original Contribution and Overview

Our contributions are concerned with the three questions stated in the introduction about protocols, protocol compatibility, converter correctness.

1. **Protocols**: We introduce a formal, FSM based model for SoC bus protocols. It is the first to distinguish between and capture interaction of the control and data path as in pipelining, burst transfers and data persistence. We can express low level details such as sensitivity to different clock edges and protocols involving different, phase aligned clocks.
2. **Protocol Compatibility**: We provide a precise definition of protocol compatibility, which takes into account the level of complexity permitted by the model and describe an algorithm for checking compatibility.
3. **Converter Correctness**: Converters and converter correctness are defined, similar to protocols and protocol compatibility. We show that the most general, correct converter is the greatest fixed point of an update function of conditions on buffer states. This yields a natural algorithm for synthesis of the most general converter by iterative computation of the fixed point.

The features of our model allow us to synthesize converters for protocols, which are beyond the scope of existing techniques. Aspects such as the control and data path separation lead to exponentially smaller converter FSMs as compared to existing methods.

2. Formal Definitions

2.1. Protocol Model

We model protocols as synchronous finite state machines with bounded counters, that communicate using channels. Channels are of two types: control and data. For an input control channel, a protocol can test for the presence or absence of a signal value, denoted $c?$ and $c\#$ respectively. These tests act as guards of transitions and a transition is enabled only when all of its guards are satisfied. For an output control channel, a protocol can write to (or assert) the channel, denoted $c!$. A protocol can also read values from or write values to a data channel d, denoted $d?$ and $d!$ respectively. A *channel action* is a read, a write or a value test on a channel. Let A_Σ denote the set of possible actions on a set of channels Σ and Tau denote an empty action.

Let k_d be a bounded counter associated with a data channel d and K be a set of such counters. The set of counter actions $A_K = \{reset(k), k{+}{+}, k = v | k \in K, v \in \mathbb{N}\}$ are a reset, increment, or test for equality with a natural number. We write $d!{+}{+}$, and $d!reset$, as a short hand for $d!$ followed by $k_d{+}{+}$ and $d!$ followed by $reset(k_d)$. A similar notation is used for read actions.

Commercial bus protocols support *burst* operations, in which a number of continuous memory locations are either written to or read from. It is also a common requirement that a data item put on the bus must remain valid for more than one clock cycle. Existing methods for modeling protocols are not expressive enough to such notions of data repetition and persistence. We use bounded counters to model such data path behavior and to separate the data and control paths in a protocol or converter. The counter value is changed when new data is written to or read from the associated channel. Explicitly representing counter values would result in large FSMs, which we avoid.

We require that the number of increment actions preceding a reset is a priori bounded to guarantee that counter values are bounded. Transfers of unbounded length are modeled by resetting the counter with every data transfer, so that counter values do not represent the number of data transfers but allow for a distinction between different data items.

Definition 1 (Protocol) *A protocol P is an FSM with bounded counters $(Q_P, C_P, D_P, K_P, \rightarrow_P, q_s, q_f)$, where Q_P is the set of states, $C_P = C_P^I \cup C_P^O$ is a set of input and output control channels, $D_P = D_P^I \cup D_P^O$ is a set of input and output data channels, $K_P = \{k_d | d \in D_P\}$ is a set of bounded counters, q_s is the initial state and q_f is the final state. Let $A_P = A_{C_P} \cup A_{D_P} \cup A_{K_P}$ be the set of actions on the control channels (A_{C_P}), data channels (A_{D_P}) and counters (A_{K_P}) and $\mathcal{P}(A_P)$ denote the power set of A_P. The transition relation is $\rightarrow_P \subseteq Q_P \times \mathcal{P}(A_P) \times Q_P$. All sets and relations above are finite.*

We drop the subscripts in the sets above when the context is clear. A transition (q, S, q') is denoted $q \xrightarrow{S} q'$. For a set of actions S, let $control(S)$, $data(S)$ and $counters(S)$ respectively denote the control channels, data channels and counters occurring in S.

2.1.1. Protocol Examples We now show how to model two common commercial protocols, the AMBA APB slave and AMBA ASB bus to slave protocols, in this setting. Due

to space limitations we only model "write" transfers in Figure 1, but "read" transfers can be modeled in a similar way.

The AMBA ASP protocol specification involves both falling and rising clock edges. The ASB bus initiates a transfer with a slave by asserting the control channel DSEL before a falling clock edge. If BWRITE is asserted simultaneously (as in the transition from state 0 to 2 in Figure 1), the transfer is a write. The slave either reads the data written to the bus or responds by inserting a wait or error state or indicates that a memory boundary has been reached (transitions from state 2 in Figure 1). Additional transfers are pipelined by asserting DSEL (transition from state 4 to 2).

The AMBA APB specification only involves the rising clock edge. An APB slave is idle (states $0, 1$ in Figure 1) until it is selected by assertion of PSEL (the transition from state 1 to 2). If PSEL and PWRITE are asserted, a WRITE transaction begins. In a WRITE transaction the slave may read the address and data in either the first or the second clock cycle (transitions from state 1 to state 2 and state 3 to state 0 in Figure 1). The model of the APB considers both the rising and falling edges of the clock to enable comparison with the ASB model. The falling edge transitions are labelled with Tau. Detailed descriptions of both protocols are available at [4].

(a) AMBA APB protocol

(b) AMBA ASB protocol

Figure 1. AMBA protocol models

2.2. Parallel Composition

We are interested in the behavior of protocols executing concurrently, which is described by the parallel composition of the protocols. We define a binary predicate may to identify transitions in two protocols that may occur simultaneously during concurrent execution.

Definition 2 (may) *The predicate* $\mathtt{may}(S_1, S_2)$ *is true for two sets of actions* S_1, S_2 *iff* $\forall c \in control(S_1 \cup S_2)$:
if $c? \in S_1$, *then* $c! \in S_2$, *if* $c\# \in S_1$, *then* $c! \notin S_2$,
if $c? \in S_2$, *then* $c! \in S_1$, *if* $c\# \in S_2$, *then* $c! \notin S_1$.

Note that the definition relates only to control channels. The predicate $\mathtt{may}(S_1, S_2)$ is true for two sets of actions S_1 and S_2 if their guards (checks on control input channels) are satisfied. Thus two transitions labelled with these actions may occur concurrently in the composition of the protocols.

Definition 3 (Parallel Composition (PC)) *The parallel composition of two protocols* $P_1 = (Q_1, C_1, D_1, K_1, \rightarrow_1, q1_s, q1_f)$ *and* $P_2 = (Q_2, C_2, D_2, K_2, \rightarrow_2, q2_s, q2_f)$ *is a finite state machine with bounded counters,* $P_1 \| P_2 = (Q_1 \times Q_2, C_1 \cup C_2, D_1 \cup D_2, K_1 \cup K_2, \rightarrow, (q1_s, q2_s), (q1_f, q2_f))$, *where* $(q1, q2) \xrightarrow{S} (q1', q2')$ *is a transition of* $P_1 \| P_2$ *iff* $q1 \xrightarrow{S_1}_1 q1'$, *and* $q2 \xrightarrow{S_2}_2 q2'$, $S = S_1 \cup S_2$, *and* $\mathtt{may}(S_1, S_2)$ *is true.*

PC describes all possible concurrent states of two protocols, subject to control compatibility, assured by may. It includes all pairs of transitions that might be taken when the two protocols communicate with each other. For two protocols that do not share a channel naming convention, channel mapping information is needed in order to compute the parallel composition. The parallel composition of AMBA ASB and APB models from Figure 1, under a mapping of the channels $\{\mathrm{DSEL} \mapsto \mathrm{PSEL}, \mathrm{BWRITE} \mapsto \mathrm{PWRITE}, \mathrm{BA} \mapsto \mathrm{PADDR}, \mathrm{BD} \mapsto \mathrm{PWDATA}\}$, is illustrated in Figure 2, showing only reachable states.

Figure 2. The parallel composition $ASB \| APB$

3. Protocol Compatibility

In this section, we address the question: Given two protocols, are they compatible? Protocol compatibility is defined in terms of the following constraints for ensuring correct data flow:

1. Data is read only when written.
2. A data item is read as distinct exactly once.
3. No deadlocks occur and livelocks can be avoided.

The first requirement ensures that data, which is read by one protocol, is guaranteed to be valid by the other protocol. The second requirement ensures that if a data item is on the bus for more than one cycle, the protocols are aware that it is the same item, though it may be read multiple times. The third condition ensures that every transaction progresses and *can* terminate in a finite number of steps. Note that to guarantee absence of livelocks, we would have to include fairness constraints in the protocol specifications, which is beyond the scope of this work.

We introduce a few definitions required to formally define protocol compatibility. For conciseness, we assume there are two protocols P_1 and P_2, sharing a data channel written to by P_1 and read from by P_2.

Definition 4 (Path) *A path* π *in a protocol* P *is a sequence of transitions* $\pi = q_0 \xrightarrow{S_1} q_1 \ldots \xrightarrow{S_k} q_k$ *such that for* $0 \leq i < k$, $q_i \xrightarrow{S_{i+1}} q_{i+1}$ *is a transition in* P. *Let*

1. $|\pi|$ *denote the number of transitions in* π*, also referred to as the length of* π.
2. $New(\pi, d) = \{i \in \mathbb{N} \,|\, 0 < i \leq |\pi|$ *and* $reset(k_d) \in S_i$ *or* $k_d{++} \in S_i\}$ *be the set of indices of transitions in* π *in which new data is accessed on channel* d.
3. *For an action* S *on a transition in* $P_1 \| P_2$, $S \downharpoonleft P_1$ *be the actions of* P_1 *in* S.
4. *For a path* $\pi = (q1_0, q2_0) \xrightarrow{S_1} \ldots \xrightarrow{S_k} (q1_k, q2_k)$ *in* $P_1 \| P_2$*, where* $q1_j, \in P_1$ *and* $q2_j \in P_2$*, the projection of* π *on* P_1, $\pi \downharpoonleft P_1 = q1_0 \xrightarrow{S_1 \downharpoonleft P_1} q1_1 \ldots \xrightarrow{S_k \downharpoonleft P_1} q1_k$*. The projection* $\pi \downharpoonleft P_2$ *is similarly defined.*
5. *The path* π *is loop-free iff for all* $0 \leq i < k$ *and* $i < j \leq k$, $q_i \neq q_j$.
6. *The path* π *is a simple cycle iff* $q_0 \xrightarrow{S_1} q_1 \ldots q_{k-1}$ *is a loop-free path and* $q_0 = q_k$.

Let $Paths(P, q_j, q_k)$ denote the (possibly infinite) set of paths in P from state q_j to state q_k. We define compatibility of two protocols P_1 and P_2 in terms of constraints over the paths between the initial and final state in $P_1 \| P_2$.

Definition 5 (Compatibility) *Two protocols* P_1 *and* P_2 *are compatible iff:*

1. $Paths(P_1 \| P_2, (q1_s, q2_s), (q1_f, q2_f)) \neq \emptyset$.
2. *For any path* π *from initial to final state of* $P_1 \| P_2$*, it holds that:*
   (a) *for every transition label* S, $d? \in S \Rightarrow d! \in S$.
   (b) $|New(\pi \downharpoonleft P_1, d)| = |New(\pi \downharpoonleft P_2, d)|$.
   (c) *Let* $i_1 < i_2 \cdots < i_n$ *be the sorted sequence of indices in* $New(\pi \downharpoonleft P_1, d)$ *and* $j_1 < j_2 \cdots < j_n$ *be the sorted sequence of indices in* $New(\pi \downharpoonleft P_2, d)$*. For index* $1 \leq \ell \leq n$ *it holds that* $j_{\ell-1} < i_\ell \leq j_\ell < i_{\ell+1}$ *where* j_0 *is defined as* 0 *and* i_{n+1} *is defined as* $|\pi| + 1$.
3. *Every reachable state in* $P_1 \| P_2$ *is on a path to the final state.*

The first requirement in Definition 5 guarantees that the protocols can complete a transaction together. For every path from the initial to the final state: Condition (2a) ensures that only valid data is read by requiring that a protocol does not read data from a channel unless it is guaranteed to hold a valid value. Condition (2b) ensures that the same number of distinct data items are written and read by the two protocols. This does not however guarantee that the *same* distinct data items are written and read, which is provided by the next condition. Condition (2c) ensures that a new data item is written only after the previous one has been read and that multiple reads of an unchanged data item are not counted as distinct. These conditions together guarantee the first two conditions for correct data flow. The final condition (3), guarantees the absence of deadlocks as every state has an outgoing transition and provides the possibility to avoid livelocks as every transaction can reach the final state and terminate. In the general case where there is more than one data channel, condition (2) should hold for every channel independently.

Deriving an algorithm for automatic compatibility checking from Definition 5 is straight forward, with the exception of requirement 2 in the definition, which deals with a possibly infinite set of paths from initial to final states in the parallel composition of protocols. We overcome this obstacle by observing that every path can be described as the composition of a sequence of loop-free paths and simple cycles, and the sets containing them is finite. Requirement (2) is checked by checking that every loop-free path and simple cycle complies with conditions (2a)-(2c) and in addition, that the composition of the loop-free paths with the cycles, preserves the requirement on the alternation of read and write actions. Due to space restrictions, we refer the reader to [5], where complete details of the algorithms for compatibility checking and extraction of loop-free and simple cycles are presented.

4. Converter Synthesis

Two incompatible protocols may still be able to communicate using a *converter*, an FSM with bounded counters and finite buffers. Correctness of a converter for a protocol pair (P_1, P_2) is defined in terms of the following constraints on the composition of the protocols with the converter:

1. Data is read only when written.
2. A data item is read as distinct exactly once.
3. No deadlocks occur, and livelocks can be avoided.
4. Fidelity constraints:
   (a) Every data item written by P_1(/P_2) to the converter will be written by the converter to P_2 (/P_1).
   (b) New data items are only introduced by P_1 or P_2.

Conditions (1)–(3) are precisely those required for compatibility except that they now need to consider data flow through the converter. The fidelity constraint (4a) ensures that all data read by the converter is eventually written out, and condition (4b) guarantees that the converter does not create any data. Before we formalise these conditions, we need to precisely define what a converter is.

4.1 Converter Model and Correctness

A converter is an FSM with bounded counters and a buffer b_d for each input data channel d. The state of b_d, also denoted as b_d, is the number of items in the buffer and $size(b_d)$ is the buffer size. To prevent buffer overflow, the converter should read into the buffer only if $b_d < size(b_d)$, and should write from the buffer only if $b_d > 0$ to prevent underflow. A *guard* is a condition $x \leq b_d \leq y$ on a buffer state with $0 \leq x \leq y \leq size(b_d)$. The conjunction or disjunction of a set of guards is also a guard.

Definition 6 (Converter) *A converter* M *is an FSM with finite counters and buffers* $(Q_M, C_M, D_M, K_M, B_M, \rightarrow_M, q_s, q_f, out)$*, where* Q_M, C_M, D_M, K_M, q_s *and* q_f *are as in Definition 1. The function* $out : D_M^I \rightarrow D_M^O$ *maps input data channels to output ones, and* $B_M = \{b_d | d \in D_M^I\}$ *is a set of data buffers. Let* G_{B_M} *be the set of guards on buffers in* B_M *and* A_M *be as before. The transition relation is* $\rightarrow \subseteq Q_M \times G_B \times A_M \times Q_M$.

For example, $(q_1, 3 \leq b_d \leq 7, \{req?, d?{+}{+}\}, q_2)$ is a transition in which, if the control channel req is asserted and there are at least 3 and at most 7 items in the buffer b_d, a new data item is read on d and a transition is made to q_2.

Let (P_1, P_2) be an incompatible protocol pair and M be a converter. For notational simplicity, in what follows, we assume that there is only one data channel d_1 written to by P_1 and read from by M and a corresponding channel $d_2 = out(d_1)$, written to by M and read by P_2 (as shown in Figure 3), and $B = size(b_{d_1})$.

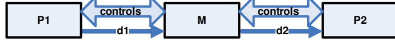

Figure 3. System structure

Definition 7 (Converter Correctness) *A correct converter M for the protocol pair (P_1, P_2) satisfies that:*

1. *P_1 and M are compatible. So are P_2 and M.*
2. *Any path $\pi \in Paths(M, q_s, q_f)$ satisfies that:*
   (a) $|New(\pi, d_1)| = |New(\pi, d_2)|$.
   (b) *let $i_1 < i_2 \cdots < i_n$ be the sorted sequence of indices in $New(\pi, d1)$ and $j_1 < j_2 \cdots < j_n$ be the sequence of sorted indices in $New(\pi, d2)$. For any $1 \leq \ell \leq n$ it holds that $i_\ell \leq j_\ell \leq i_{\ell+B}$.*

   where q_s is a state in $(P_1 \| M \| P_2)$ in which P_1 and P_2 are in initial states and b_{d_1} is empty, and q_f is a state at which P_1 and P_2 are in final states and b_{d_1} is empty.

Condition (1) ensures that data items are read as distinct only once, when written, that no deadlocks occur and livelocks can be avoided, as in compatibility. Conditions (2a) and (2b) guarantee the fidelity constraints, that the converter passes all given data and does not create data. These conditions subsume requiring the absence of over- and underflow as the former leads to data loss and the second to data creation. In the general case, condition (2) of Definition 7 should hold for every pair $(d, out(d))$ of data channels.

Definition 7 allows us to precisely define two problems :
Converter Verification: Given a converter M for a protocol pair (P_1, P_2), check if it is correct.
Converter Synthesis: Given an incompatible protocol pair (P_1, P_2), synthesize a correct converter M.

We now present an algorithm for the latter problem.

4.2. Automatic Converter Synthesis

The converter synthesis algorithm takes as input two protocols and a map between their data channels, and returns the most general correct converter, that is, a possibly nondeterministic converter containing all correct behaviors, from which smaller, deterministic converters can be extracted. The first step is to compute the inverted product of the two protocols, containing all possible behaviors of a converter. Then, the behavior of this machine is restricted to the correct behavior for a converter by adding the required guards. Due to mutual dependencies, guards have to be iteratively updated until a fixed point is reached. This fixed point describes a converter with guards for a correct converter. The inverted product is described in Section 4.2.1, and the restriction to correct behavior in Section 4.2.2.

4.2.1. Inverted Product Recall that the pairs of transitions occurring simultaneously in $P_1 \| P_2$ are those with actions S_1, S_2, satisfying $\texttt{may}(S_1, S_2)$. The *product* of P_1 and P_2, denoted $P_1 \,|||\, P_2$ is defined by removing the requirement $\texttt{may}(S_1, S_2)$ in Definition 3. Thus, the product $P_1 \,|||\, P_2$ retains all pairs of transitions, and describes the most general behavior of two protocols.

The inverse of an action S is obtained by replacing $c? \in S$ by $c!$ and vice versa, and $c\# \in S$ by τ for all channels $c \in control(S) \cup data(S)$. The *inverted product* is obtained from $P_1 \,|||\, P_2$ by replacing the action on each transition by its inverse. Counter actions are left unchanged. Let $\textsc{Invert}(P)$ be a procedure inverting the protocol P. Observe that the inverted product describes and enables all behaviors of the two protocols.

4.2.2. A Correct Converter from the Inverted Product A converter is obtained from the inverted product by adding the guard $0 \leq b_d \leq size(b_d)$ to all transitions. The guard may have to be strengthened to ensure absence of over-/underflow. For example, if a transition may be followed by three consecutive reads, its guard should be $0 \leq b_d \leq size(b_d) - 3$. Further, a state q may have several outgoing transitions with different guards, and the guard computed for incoming transition to q must consider all those guards. For presentation purposes, we additionally assume that the conditions on outgoing transitions from any state are mutually exclusive. The full algorithm for the general case is presented in [5].

For a transition t with guard $x \leq b_d \leq y$, let $\ell(t)$ be x and $u(t)$ be y. As we consider only one data channel and buffer, the guard for t can be written as $[\ell(t), u(t)]$, denoting the lower and upper bounds on b_d. The *data constraint* $[\ell(q), u(q)]$ on a control state q describes the acceptable buffer states for incoming transitions to q, where $\ell(q) = \max(\ell(t_1), \ldots, \ell(t_n))$ and $u(q) = \min(u(t_1), \ldots, u(t_n))$ and $t_1, \ldots t_n$ are the outgoing transitions from q. That is, $\ell(q)$ is the greatest of the lower bounds and $u(q)$ is the least of the upper bounds on outgoing transitions.

Define the function t_update which maps a transition $(q_1, [x, y], S, q_2)$ to $(q_1, [x', y'], S, q_2)$, where $[x', y']$ is (a) $[\max(0, \ell(q_2) - 1), \max(-1, u(q_2) - 1)]$ if $d?{+}{+}$ or $d?reset \in S$, or (b) $[\min(\ell(q_2) + 1, B + 1), \min(u(q_2) + 1, B)]$ if $d!{+}{+}$ or $d!reset \in S$, or (c) $[x, y]$ otherwise. This function updates the guard of a transition, considering the data constraint of the destination state and the data actions in the transition. If a read occurs, the data constraints are decremented and if a write occurs, they are incremented. Note that a guard can be updated to an empty range, meaning that it should never be enabled. For a set of transitions T, define $update(T) = \{t_update(t) | t \in T\}$. An example of the repeated application of $update$ to a set of transitions is shown in Figure 4. The intervals $[x, y]$ in red show the guards and data constraints updated in each application.

The converter synthesis algorithm is now defined in terms of $update$. The function $update$ is monotonic in the guards assigned to the transitions of the converter. Iterative application of $update$ to a converter initialized with the guards $[0, B]$ yields the greatest fixed point of $update$.

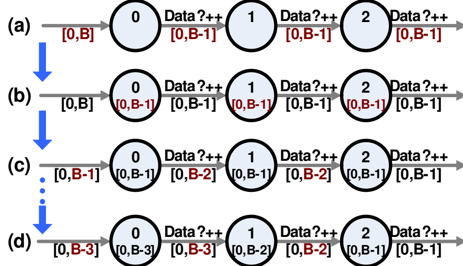

Figure 4. Application of $update$

Algorithm 1 Algorithm for Converter Synthesis

SYNTHESIZE CONVERTER(P_1,P_2,out)

Input: Protocols and mapping.

Output: Most general correct converter.

1: $IP :=$ INVERT($P_1 ||| P_2$)
2: $C := IP$ with guard $[0, B]$ on all transitions
3: $T :=$ Transitions of C
4: **repeat**
5: $old_T := T$
6: $T := update(T)$
7: **until** $(old_T = T)$
8: return C

In cases where it is impossible to construct a correct converter with the specified buffer size, the algorithm will result in either an initial state range that does not include an empty buffer (meaning that no transition is enabled at the starting point) or a final state range that does not include an empty buffer (meaning that some data transfers never end).

5. Experimental Results

Using the presented formalism, we have successfully modeled the protocols of the Advanced Peripheral Bus (APB), Advanced System Bus (ASB) and Advanced High-performance Bus (AHB) of the AMBA bus protocol family, as well as some configurations of the Open Core Protocol (OCP) [1] - widely used commercial protocols, employing different protocol features at different complexity and performance level. We have modeled master, slave and bus specifications in unprecedented detail, for both read and write operations.

Table 1. Automatic converter synthesis

		Conditions tested				
Protocols		BWidth ratio			Operations	
Initiator	Reactor	1:1	1:2	2:1	Read	Write
ASB(12)	APB(7)	✓	✓	✓	✓	✓
APB(7)	ASB(10)	✓	✓	✓	✓	✓
AHB(8)	APB(3)	✓	✓	✓	✓	✓
APB(3)	AHB(6)	✓	✓	✓	✓	✓
APB(3)	OCP(5)	✓	✓	✓	✓	✓
OCP(5)	APB(3)	✓	✓	✓	✓	✓
ASB(12)	OCP(9)	✓	✓	✓	✓	✓
OCP(9)	ASB(10)	✓	✓	✓	✓	✓

The algorithms for completely automated compatibility checks and converter synthesis, have been implemented in a PC based tool and applied to pairs of protocol models, at different bandwidth ratios. The experiments conducted for automatic converter synthesis for incompatible protocols are listed in Table 1, where the numbers next to the protocol name represents number of states in the protocols' model. In all listed experiments a converter was successfully created, with the exception of abort operations that are currently not covered. It is a property of our converter synthesis algorithm that the number of states in a synthesized converter is bounded by the product of the number of states of its protocol models, an improvement by orders of magnitude to existing methods, as the converter size no longer depends on the product of buffer sizes.

6. Conclusions

In this paper we have presented three important elements of provably correct on chip communication: (1) An efficient algorithm for automatic converter synthesis. (2) A comprehensive framework for modeling hardware protocols, that is the first to allow precise and detailed modeling of commercial protocols at a low level of abstraction, and enables direct translation to HDL. (3) General and formalized definitions for protocol compatibility and the protocol converter synthesis problem, introducing criteria for correctness of any suggested converter.

The combination of low level abstraction and efficient converter synthesis algorithm, produces for the first time, provably correct, practical converters that can be easily and even automatically translated to HDL. The methods presented have been successfully applied to various commercial protocols.

References

[1] Open core protocol international partnership, http://www.ocpip.org.
[2] J. Akella and K. L. McMillan. Synthesizing converters between finite state protocols. In *ICCD*, pages 410–413. IEEE Computer Society, 1991.
[3] V. Androutsopoulos, D. Brookes, and T. Clarke. Protocol converter synthesis. *Computers and Digital Techniques*, 151(6):391–401, 2004.
[4] ARM. Amba specification, http://www.arm.com/.
[5] K. Avnit, V. D'Silva, A. Sowmya, S. Ramesh, and S. Parameswaran. Protocol compatibility and automatic converter synthesis. Technical Report 0718, UNSW, Australia, August 2007.
[6] V. D'Silva, S. Ramesh, and A. Sowmya. Bridge over troubled wrappers: Automated interface synthesis. In *VLSI Design*, pages 189–194. IEEE Computer Society, 2004.
[7] V. D'Silva, S. Ramesh, and A. Sowmya. Synchronous protocol automata: A framework for modelling and verification of soc communication architectures. In *DATE*, pages 390–395. IEEE Computer Society, 2004.
[8] D. Gajski, H. Cho, and S. Abdi. General transducer architecture. Technical Report TR 05-08, CECS Center for Embedded Computer Systems University of California, Irvine, August 2005.
[9] S. Narayan and D. Gajski. Interfacing incompatible protocols using interface process generation. In *DAC*, pages 468–473, 1995.
[10] R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: two faces of the same coin. In *ICCAD*, pages 132–139. ACM, 2002.
[11] R. Passerone, J. A. Rowson, and A. L. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *DAC*, 1998.
[12] J. Smith and G. D. Micheli. Automated composition of hardware components. In *DAC*, pages 14–19, 1998.
[13] S. Watanabe, K. Seto, Y. Ishikawa, S. Komatsu, and M. Fujita. Protocol transducer synthesis using divide and conquer approach. In *ASP-DAC*. IEEE, 2007.