Abstract

Reuse of components is a burgeoning field in chip design. Shorter time to market and assured quality are just two good reasons to reuse previously engineered components. Problems arise however when chip designers need to interface these components as they typically conform to different standards, or no standards at all. The popular model of interfacing components such as protocols is via a 'converter' that translates data between the components. We develop a theoretical model of a converter that will enable two given arbitrary protocols to communicate. This model includes buffers. We formally define correctness conditions, and guarantee that the resulting converter satisfies these conditions. We also allow the designer to define his own (CTL) conditions. As well, we allow protocols to be nondeterministic, and we ensure only valid data is sent to the converter. The verification of the conditions is carried out by a model checker (not reported in this work). We have implemented our theoretical model and we present experimental results.

Keywords: component reuse, formal verification, protocol converter, converter synthesis

1 Introduction

In the semiconductor industry, a key challenge is the ability to reuse components such as Intellectual Property (IP) that has been developed elsewhere (Mullane & MacNamee 2008). The drive to reuse existing technology has become central to design particularly in the development of Systems-on-Chip. The aims of reuse are to reduce the design development time and to increase quality. It is reported for example (Keating & Bricaud 1999) that 70% of time and money in chip design is spent on verification, including both verification for each single component and communication between multiple components. It would be a boon to the industry to have technology that enabled pre-verified IPs to be reused at will.

Reuse in design is fraught with difficulties however as most IPs have different interfaces based on different control and data protocols, different formats, or even different timing. We refer to these differences as mismatches. To reuse IPs, we can deal with these mismatches by building a converter that translates the signals from each IP to match the other. This building process is called converter synthesis. There are important fundamental issues that need to be addressed when considering synthesising a converter. For example, what kind of mismatches can we correct? What kind of mismatches can’t we correct? When does a converter not exist? Is there more than one kind of converter possible? How do we know that a converter is correct?

In Figure 1 we depict our model of a system comprising two protocols and a converter. The labels $d$, and $c$, in this figure denote data and control (resp.) that are sent by the protocols, and $d$, and $c$, data and control (resp.) that are received by the two protocols. For the sake of simplicity we limit this work to IPs that share a clock and work synchronously. Note in this depiction we have separate data and control channels connecting the IPs with the converter, and a separate buffer in the converter attached to each data channel. The buffers are used to collect data, which is required to correct certain types of mismatch. While our work is based on a converter with precisely two buffers, more or less buffers could also be contemplated, and this would not change the underlying theory.

There are many potential sources of mismatch in such a system. For example, data or even control sent by one protocol may not be handled by the other protocol, or vice versa, one protocol may need data that the other is not able to provide. Different data widths is also a reason for data mismatch. For example, one protocol may use 8-bit data, the other 16-bit data. There are mismatches that can be corrected, and mismatches that cannot be corrected. The aim of this work is to synthesise a converter that provides the “glue-logic” that corrects the former, and avoids the latter.

In the next section we describe background research. In Section 3 we formally define the compo-
ment protocols in terms of finite-state automata, and we address the issue of nondeterminism, which we will see plays an important role in finding mismatches. The synthesis of a converter is based on the notion of the parallel composition of the components' behaviour, expressed as states of an automaton. We formally define the parallel composition of two protocols, and extend this definition to include the key notions of stored data (i.e. buffers) and properties. To determine whether a converter exists or not, we use a technique of the notion of correctness. In theory many definitions of correctness are possible, and naturally, the stricter we define correctness, the more difficult it is to synthesise a converter, and the greater the likelihood that no converter exists. Examples of conditions that we use are that the buffers must never overflow or underflow, and the converter never receives invalid control signal or data from a protocol. We also allow the designer to define temporal properties that the converter must satisfy. The temporal formalism that we use is CTL (Clarke et al. 2000). During formal verification, any state and transition in the extended parallel composition that violates a correctness condition is pruned. If no state remains then a converter does not exist. Otherwise, the resulting pruned converter must be correct w.r.t. the correctness conditions. We further define a converter to minimise the converter size, which behaves exactly the same as an original converter.

We have implemented our converter and experimented with various protocols, including producer/consumer, pipeline, handshake and AMBA protocols. A brief description of our implementation can be found in Section 4. In Section 5, we present the results of synthesising converters between these protocols. In Section 6 we compare our work with related work, and we discuss the contribution of our work. Finally in Section 7 we present our conclusions and discuss future work.

2 Background

We can classify previous work on defining a methodology to build a converter into constructive approaches (Androustopoulos et al. 2003, Himor et al. 2003, Coudert et al. 2004, Shin & Gajski 2002, Watanabe et al. 2007) and reductive approaches (Akella & McMillan 1991, Androustopoulos et al. 2004, Avnit et al. 2008, D'Silva et al. 2004b, Passerone et al. 2002, Sinha et al. 2007, 2008b,a). These two approaches are complimentary. In the constructive approach, the given protocols are analysed (often some kind of decomposition is used) revealing the underlying ‘transactions’ that occur. These are then used to build a state machine ‘from the ground up’, and along the way, buffers and timing considerations may be included, for example. In the reductive approach, all possible behaviours of the converter are modelled, and those behaviours that are invalid are removed, leaving a converter consisting of only valid states and transitions.

The reductive approach in particular lends itself to formal methods. Research in this area dates back to the work by Green (1986). The work by Akella & McMillan (1991) first suggested using a product finite-state machine and removing invalid states to synthesise a protocol converter. Works by Passerone et al. (1998) and then by Androustopoulos et al. (2004) extended this research by ‘unrolling’ the product machine, analysing the data paths, and removing those that represent undesirable behaviour. When given a choice, they also removed data paths that produced lower performance in terms of the data transfer rate. This involved graph theoretic algorithms.

The resulting converter has data paths with maximum bandwidth.

D'Silva et al. (2004b) extended that work by formally modelling control and data separately, using buffers, and dealing with the problem of data mismatch explicitly. Very recently, that work was developed further by Avnit et al. (2007, 2008) who improved the underlying formal model, added formal definitions of compatibility and of correctness, and developed algorithms to implement the formal model. They use their formal correctness definition to drive the pruning algorithm that iteratively removes ‘invalid’ states and transitions from the product automaton until a ‘fixed point’ is reached. The resulting converter is called the most general converter, and is provably correct w.r.t. their correctness conditions. Work not yet carried out by the authors includes removing all unreachable states from their most general converter, optimising the converter in terms of size (for example).

The starting point of our work was the research from Sinha et al. (2007, 2008b,a). That work takes a different approach, integrating model-checking technology, normally used in verification, to synthesise a converter. Using this approach, correctness is not (only) ‘hard-wired’ into the synthesis algorithms but is also specified in CTL instead. A comparison of their work and ours requires some basic technical understanding of the approach. Accordingly, we postpone any further discussion until after the presentation of our work.

3 Formal definitions

The general communication system in Figure 1 consists of two protocols and a converter with two buffers. To build this system, we first formally define a protocol, and then develop a formal model of a converter.

Definition 3.1 Protocol

A protocol $P$ can be represented by a finite-state machine $<S, \Sigma, \delta, i, f>$ where:

- $S$ is a finite, non-empty set of states.
- $\Sigma = \Sigma_s \cup \Sigma_r$ is a finite set of send and receive actions. $\Sigma_s$ and $\Sigma_r$ may not be disjoint.
- $\delta$ is a state-transition function: $\delta \subseteq S \times 2^\Sigma \rightarrow S$, which defines transitions that are labelled by zero or more send and receive actions. For each transition $t_i \in \delta, t_i = (s_i, A(t_i), s_j)$, where $s_i, s_j \in S$ and $A(t_i) \subseteq \Sigma$ labels the transition by actions.
- $i \in S$ and $f \in S$ are the initial and final states respectively.

Notation In general text we write $a!$ to indicate a send action $a \in \Sigma_s$, and $a?$ to indicate a receive action $a \in \Sigma_r$. Furthermore, we write $a\#$ to indicate some receive action $b$, where $b \in \Sigma_r - \{a\}$ and $a \in \Sigma_r$. Transitions that are not labelled by any actions are labelled by $\tau$ instead, i.e. the protocol has received nothing in the transition. In practice, they may correspond to timeouts for example. We write $s_i \overset{A(t_i)}{\rightarrow} s_j$ if $\delta(s_i, A(t_i)) = s_j$ for states $s_i, s_j \in S$, and transition label $A(t_i) \subseteq \Sigma$. We also assume a protocol satisfies the following conditions:

- $a_1$. One transition occurs in each clock cycle.
- $a_2$. If there are both send and receive actions in a transition, then all receive actions must precede send actions.
Each state can eventually reach the final state.

**Example 3.2** The protocol $P_1$ in Figure 2 can be represented as a finite-state machine $\langle S, \Sigma, \delta, i, f \rangle$, where $S = \{s_1, s_2, s_3, s_5\}$, $\Sigma = \{\text{rin}, \text{rout}, \text{dout}, \text{etc}\}$, $\delta = \{s_1 \xrightarrow{\text{rout}} s_1, s_2, s_1 \xrightarrow{\text{rin}} s_3, \text{etc}\}$ and $i = f = s_1$. Similarly for protocol $P_2$: $S = \{s_1, s_2\}$, $\Sigma = \{a, r, d\}$, $\delta = \{s_1 \xrightarrow{\text{r}\text{d}} s_1, s_1 \xrightarrow{\text{r}\text{in}} s_2, s_2 \xrightarrow{d} s_1\}$ and $i = f = s_1$.

Note in $P_1$ in Figure 2 that the state $s_1$ has only transitions with no input. This state is therefore non-deterministic as no external component can control which outgoing transition will be taken. This can be a problem because one of the transitions of such a nondeterministic state may lead to correct behaviour, but another may be ‘invalid’, in the sense it may generate control or data that the other protocol cannot handle. For example, in $P_1$ in Figure 2, hypothetically correct but $\text{r}\text{out}$ may be correct, but $\text{r}\text{in}$ may not be. In general, any data that is sent to the converter as a result of such an invalid transition is referred to as invalid input to the converter. Recognising the existence of nondeterministic states that may generate invalid data is important in defining correct behaviour, as we shall see later. In the following definition, we define nondeterminism in a protocol.

**Definition 3.3 Non-deterministic (ND) protocol**

We are given a protocol $P = \langle S, \Sigma, \delta, i, f \rangle$ where $\Sigma = \Sigma_{\text{in}} \cup \Sigma_{\text{out}}$ as shown in Definition 3.1, and a state $s \in S$, via $n > 1$ outgoing transitions $t_1, \ldots, t_n \subseteq \delta$, can reach its children $\delta(s(A(t_1)), \ldots, \delta(s(A(t_n)))$ that are denoted by $s_1, \ldots, s_n$. For each outgoing transition label $A(t_i)$, $1 \leq i \leq n$, we denote its receive actions by $\Sigma_{r,t_i} \subseteq \Sigma_r$. For $\exists l, m \in 1..n$ and $l \neq m$, if $s$ has children $s_1 = \delta(s(A(t_1)), \ldots, s_m = \delta(s(A(t_m)))$ such that

- $s_1 \neq s_m$ and
- $\Sigma_{r,t_j} = \Sigma_{r,t_m}$

then $s$ is a nondeterministic state and $P$ is a non-deterministic protocol. Moreover, the children $s_1$ and $s_m$ are referred to as twins. We can of course simply extend the notion of twins to triplets, quadruplets and so on.

In essence, a ND state has different transitions that have identical receive actions in the labels. Note, however, that transitions from a state must be unique so the send actions must differ in that case. The consequence of this is that $\delta$ in Definition 3.1 is a function.

**3.1 Parallel composition of protocols**

Having defined the protocols, the first step in the derivation of a converter is to define the parallel composition of two protocols.

**Definition 3.4 Parallel composition of two protocols**

Given two protocols $P_1 = \langle S_1, \Sigma_1, \delta_1, i, f_1 \rangle$ and $P_2 = \langle S_2, \Sigma_2, \delta_2, i, f_2 \rangle$, where each transition $t_i \in \delta_i$ denotes $(s_i, A(t_i), s_j)$ in $P_1$ and similarly $t'_i \in \delta_2$ denotes $(s'_i, A'(t'_i), s'_j)$ in $P_2$ as shown in Definition 3.1, the parallel composition $P_1||P_2$ is defined as a finite-state machine $\langle S', \Sigma', \delta', i', f' \rangle$, where

- $S' = S_1 \times S_2$
- $\Sigma' = \Sigma_1 \cup \Sigma_2 \cup \Sigma_{\text{in}} \cup \Sigma_{\text{out}}$, where $\Sigma_{\text{in}} = \Sigma_1 \cup \Sigma_2$
- $\Sigma_{\text{out}} = \Sigma_1 \cup \Sigma_2$
- $\delta' \subseteq S' \times \Sigma' \times S' \rightarrow S'$, where each transition is labelled by zero or more send and receive actions from $\Sigma_{\text{in}}$ and $\Sigma_{\text{out}}$. For all $t_i \in \delta$ and $t'_i \in \delta_2$, $\exists t_i t'_i \in \delta'$, where $t_i t'_i = (s_i s'_i, A(t_i), A'(t'_i), s_j s'_j)$.
- $i' = i'_{i'_{i'_{i'}}} \in S'$ is the initial state and $f' = f'_{f'_{f'_{f'}}}$ in $S'$ the final state.

**Notation** Graphically, we separate the transition labels from $\Sigma_{\text{in}}$ and $\Sigma_{\text{out}}$ by "\text{\textbar}". For example, $x?y$ means that process $P_1$ receives signal $x$, at the same time as process $P_2$ sends signal $y$, where $x \in \Sigma_{\text{in}}$ and $y \in \Sigma_{\text{out}}$.

We write $s_{i,j}$ if $A(t_i)_{A(t'_i)} s_{g,h}$ if $\delta'(s_{i,j}, A(t_i), A(t'_i))$ and similarly for states $s_{i,j}$, $s_{g,h} \in S'$, where transition label $A(t_i) \subseteq \Sigma_{\text{in}}$ and $A(t'_i) \subseteq \Sigma_{\text{out}}$. Note that for convenience we write $s_{i,j}$ to denote $s_{i,j}$.

**Example 3.5** The parallel composition of protocols $P_1$ and $P_2$, shown in Figure 3, can be written as $\langle S', \Sigma', \delta', i', f' \rangle$, where $S' = \{s_{1,1}, s_{1,2}, \text{etc}\}$, $\Sigma' = \{\text{din}, \text{ack}, r, \text{etc}\}$, $\delta' = \{s_{1,1} \xrightarrow{\text{r}\text{out}} s_{1,1}, s_{1,1} \xrightarrow{\text{r}\text{out}} s_{2,1}, s_{1,1} \xrightarrow{\text{r}\text{out}} s_{2,2}, s_{1,1} \xrightarrow{\text{r}\text{out}} s_{1,2}, \text{etc}\}$, $i' = f' = s_{1,1}$.

We now define a so-called extended parallel composition of the given protocols, and in so doing, introduce data storage and the set of properties that the parallel composition is expected to satisfy. A counter is used to record the number $k$ of data items that need to be stored in a given state. The counter is denoted by variable $k$, which is incremented and decremented as data is received by and sent from the converter. The properties are expressed as CTL formulae. Each
formula that is true in a given state is included in the state.

**Definition 3.6** Extended parallel composition (EPC) of two protocols

We are given a parallel composition \( P_1 || P_2 = \langle S', \Psi, L, \Sigma', \delta', \tau' \rangle \) and a state \( s_{i,j,k} \in S' \) with \( n \) outgoing transitions \( t_1 t_1', \ldots, t_n t'_n \subseteq \delta' \).

If \( \exists l, m \in 1..n \) with \( l \neq m \), assume we have two transitions \( \delta'(s_{i,j,k}, A(t_l), A'(t'_{l})) = s_{l,r',k'} \) and \( \delta'(s_{i,j,k}, A(t_m), A'(t'_{m})) = s_{m,n',k''} \). If one of the following 3 cases occurs:

- \( t_l = t_m \) and \( s_i \in S_{P_1} \) is ND (because of \( t_l \) and \( t_m \) as shown in Definition 3.3)
- \( t_l = t_m \) and \( s_j \in S_{P_2} \) is ND (because of \( t'_l \) and \( t'_m \))
- \( s_i \in S_{P_1} \) is ND (because of \( t_l \) and \( t_m \)) and \( s_j \in S_{P_2} \) is ND (because of \( t'_l \) and \( t'_m \))

then the children \( s_{i,j',k'} \) and \( s_{m,n',k''} \) are twins. Hence state \( s_{i,j,k} \) is ND and the EPC is ND.

**Example 3.9** The EPC in Figure 4 is ND because the state \( s_{3,1,0} \) is ND; caused by the transitions \( rin!r# \) and \( rout!r# \). The states \( s_{3,1,0} \) and \( s_{2,1,0} \) are twins.

### 3.2 Raw converter

There are many EPCs corresponding to any given two protocols, each exhibiting possibly quite different behaviour. For example, the counters in the EPC may violate bounds that we would wish to impose on an implementation (where we use buffers to store data for example). By applying correctness conditions to an EPC, we define a model of a raw converter for the protocols.

**Definition 3.10** Raw converter

We are given an EPC \( P_1 || P_2 = \langle S', \Psi, L, \Sigma', \delta', \tau' \rangle \) and two buffers with size \( q_1 \) and \( q_2 \), where \( S' \subseteq S' \times K \) and \( \Sigma' = \Sigma' \cup \Sigma' \cup \Sigma' \cup \Sigma' \) as shown in Definition 3.6. As well, we are given a final state \( f' \) from \( P_1 || P_2 = \langle S', \Psi', \Sigma', \delta', \tau' \rangle \).

A raw converter \( P_1 || P_2 \) is defined by an extended finite-state machine \( \langle S^R, \Psi^R, L^R, \Sigma^R, \delta^R, \tau^R, f^R \rangle \) where:

- \( S^R \subseteq S' \) is a finite set of states, and when written as \( S^R \subseteq S'' \times K' \), where \( S'' \subseteq S' \) and \( K' = (K_{1} \times K_{2}) \), where \( K_{1} \subseteq K_{1}, K_{2} \subseteq K_{2} \).
- Each state \( s_{i,j,k} \in S^R \) where \( k = (k_{1}, k_{2}) \), we have \( 0 \leq k_{1} \leq q_{1} \) and \( 0 \leq k_{2} \leq q_{2} \).
- \( \Psi^R \subseteq \Psi \) is a set of boolean formulae.
- \( L^R \) is a state labelling function: \( S^R \rightarrow 2^{\Phi} \).
- \( \Sigma^R \subseteq \Sigma' \) and \( \Sigma^R \) can be written as \( \Sigma_{R_1} \cup \Sigma_{R_2} \), where \( \Sigma_{R_1} = \Sigma_{R_1} \cup \Sigma_{R_1} \cup \Sigma_{R_1} \cup \Sigma_{R_1} \) and \( \Sigma_{R_2} = \Sigma_{R_2} \cup \Sigma_{R_2} \).

Note we have not yet placed any limit on the counters, hence the EPC may contain an infinite number of states.
- If $S^R \neq \emptyset$ then a raw converter exists, and the given protocols are said to be compatible. Otherwise, a raw converter does not exist, and the given protocols are said to be incompatible.

We need to remember here that a converter is an interface between two protocols. In the definition of $\Sigma$, we see actions are reversed at the interfaces between the protocols and the raw converter: the send actions in each protocol become receive actions in the raw converter, and the receive actions in each protocol become send actions in the raw converter. Note as well that action $a \#'$ becomes the empty action $\tau$, and $\tau$ remains the same.

In $P_1$, in each transition $t_1 \in \delta^R$, the label consists of $A'(t) = \Sigma_{t_1} \cup \Sigma_t$ and $A'(t') = \Sigma_{t_1}' \cup \Sigma_{t_1}'$, where $\Sigma_{t_1} \subseteq \Sigma^{R_1}$ and $\Sigma_t \subseteq \Sigma^{R_2}$. Similarly in $P_2$, $\Sigma_{t_1}' \subseteq \Sigma^{R_2}$ and $\Sigma_{t_1}' \subseteq \Sigma^{R_2}$. In the raw converter, all send actions $\Sigma_{t_1}$ and $\Sigma_{t_1}'$ precede receive actions $\Sigma_t$ and $\Sigma_t'$. This is of course the opposite of what happens in each given protocol where all receive actions precede send actions. This action reversal preserves the order of the signals in the raw converter, and intuitively is what we expect.

**Example 3.11** Consider the two protocols and the resulting raw converter in Figure 5. We notice that the raw converter must send $a$ and $b$ concurrently before receiving any signal.

**Example 3.12** Corresponding to the EPC $P_1 || P_2$, shown in Figure 4, we can generate the raw converter $P_1 || R P_2$ shown in Figure 6 with $q_1 = 1$ and $q_2 = 0$. The raw converter can be written as $\langle \delta^R, \Psi^R, L^R, \Sigma^R, \delta^R, i^R, f^R, \rangle$, where $\delta^R = \{s_{1,1,00}, s_{1,1,10}, etc\}$, $\Psi^R = \{\phi_1: AG(s_{1,1,00} \rightarrow AXAF(s_{1,1,00}, \phi_2) \forall s_{i,j,k} \in \delta^R, 0 \leq k_1 \leq 1 \land 0 \leq k_2 \leq 1, \phi_3: AG(s_{1,1,00} \rightarrow AXAF((dout? \lor din!) \land d), \rangle, L^R = \{s_{1,1,00} \rightarrow \{\phi_1, \phi_2, \phi_3\}, etc\},$, $\Sigma^R = \{ack, r, d, etc\}$, $\delta^R = \{s_{1,1,00} \rightarrow \{ack\}, i^R = f^R = s_{1,1,00}$.

We mentioned earlier that nondeterminism in a protocol may lead to invalid output being sent to the converter. We in fact have this problem in the raw converter in the example above. Before we examine this example more closely, however, we formally define invalid output in an EPC.

**Definition 3.13** Invalid output

We are given a ND state $s_{i,j,k} \in S^R$ in an EPC $P_1 || P_2 = \langle S^R, \Psi, L, \Sigma^R, \delta^R, i^R, f^R, \rangle$, and a resulting raw converter $P_1 || R P_2 = \langle \delta^R, \Psi^R, L^R, \Sigma^R, \delta^R, i^R, f^R, \rangle$. Assume the state $s_{i,j,k}$ is ND because of its outgoing transitions $t_1^{l_1}$ and $t_2^{l_2}$ (both in $\delta^R$). The set of send actions for $t_1^{l_1}$ can be written as $\Sigma_{l_1}^R$ (from $P_1$) and $\Sigma_{l_2}^R$ (from $P_2$). Similarly, the set of send actions for $t_m^{l_m}$ can be written as $\Sigma_{m_1}^R$ (from $P_1$) and $\Sigma_{m_2}^R$ (from $P_2$). The set of send actions in $t_m^{l_m}$ that are different to those in $t_1^{l_1}$ can be written as $\Sigma_{m_1}^R - \Sigma_{l_1}^R$ from $P_1$ and $\Sigma_{m_2}^R - \Sigma_{l_2}^R$ from $P_2$. Let us assume the transition labels in $t_1^{l_1}$ and $t_m^{l_m}$ after reversal are called $t_1^{l_1'}$ and $t_m^{l_m'}$ respectively. If $t_1^{l_1'} \in \delta^R$ and $t_m^{l_m'} \notin \delta^R$ then every action in $\Sigma_{m_1}^R$ (if any) is an invalid output at $s_{i,j,k}$. Similarly, every action in $\Sigma_{m_2}^R$ (if any) is an invalid output at $s_{i,j,k}$.

Note that the transitions $t_1^{l_1'}$ and $t_m^{l_m'}$ generate states that are twins. The existence of twins is only a problem if there is any invalid output.

After reversal, send actions in an EPC become receive actions in a raw converter, so invalid outputs in an EPC correspond to invalid inputs in the raw converter. We define that below.

**Definition 3.14** Invalid input

If $a \in \Sigma^R$ (or $b \in \Sigma^R$) is invalid output at $s_{i,j,k}$ in the EPC defined in Definition 3.13, then $a$ (or $b$) is invalid input at $s_{i,j,k}$ in the corresponding raw converter.

We require that for all $s_{i,j,k} \in S^R$ of a raw converter, no invalid inputs occur.

**Example 3.15** Let us reconsider the EPC from the previous example, namely $P_1 || P_2$ shown in Figure 4. The state $s_{1,1,00}$ in this EPC is ND because of the transitions labelled by $rin! r#$ and $rout! r#$. Applying Definition 3.13 to this EPC, we find transition $t_1^{l_1'} = (s_{1,1,00}, rout!, r#, s_{2,1,00})$ after reversal corresponds to $t_1^{l_1'} = (s_{1,1,00}, rout!, r#, s_{2,1,00})$, which is included in $P_1 || P_2$. However, transition $t_m^{l_m'} = (s_{1,1,00}, rin!, r#$) corresponds to $t_m^{l_m'}$, which does not exist in $P_1 || P_2$. Hence action $rin!$ is invalid output at $s_{1,1,00}$ in the EPC, and $rin!$ is invalid input at $s_{1,1,00}$ in the raw converter $P_1 || P_2$. The conclusion is that the raw converter $P_1 || P_2$ is invalid. In fact, the protocols $P_1$ and $P_2$ are incompatible.

Determining whether a raw converter will handle
all input is an important part of determining correctness. We address correctness in the next section.

3.2.1 Correct Raw Converter

In Definition 3.10, we include properties that the user imposes (expressed as formulae), and various other properties are implied. For example, invalid input never occurs, and the final state is reachable from every state. We formalise all these properties by defining correctness conditions that a raw converter must satisfy.

Definition 3.16 Correctness conditions

Given an EPC, a raw converter $P_1 || P_2 = (S^R, \Psi^R, I^R, \Sigma^R, \delta^R, i^R, f^R)$ with two buffers of size $q_1$ and $q_2$ is correct if for each state $s_{i,j,k} \in S^R$, where $k = (k_1, k_2)$, all of the following conditions hold:

c1: user properties The state $s_{i,j,k}$ satisfies all formulae in $\Psi^R(s_{i,j,k})$: $s_{i,j,k} \models \Psi^R(s_{i,j,k})$.

c2: final reachability The final state $f^R$ is reachable from state $s_{i,j,k}$.

c3: valid buffer The buffer values $0 \leq k_1 \leq q_1$ and $0 \leq k_2 \leq q_2$ (i.e., there is no buffer overflow or underflow). In both initial state $i^R$ and final state $f^R, k_1 = k_2 = 0$ (i.e., every received data will eventually be sent).

c4: progress From $i^R$, some data or control signal transaction in $\Sigma^R$ can eventually be executed.

c5: valid input No invalid input occurs at $s_{i,j,k}$.

The condition c4 checks fairness and guarantees that a raw converter makes progress. For example, a raw converter that consists of only one state and one self-loop transition could satisfy all the correctness conditions but it could never communicate with another protocol. The condition c5 guarantees that no invalid input will occur in the raw converter. If a twin state is invalid, then it will violate one of the conditions, as a consequence, both transitions that lead to the twins will be excluded in the raw converter.

3.2.2 Redundant Raw Converter

Before we derive a converter from a raw converter, we first check whether the raw converter is required at all (i.e. it is not redundant). We define a redundant raw converter formally as follows.

Definition 3.17 Redundant raw converter

We are given a raw converter $P_1 || P_2 = (S^R, \Psi^R, I^R, \Sigma^R, \delta^R, i^R, f^R)$ with two buffers, where $\Sigma^R = \Sigma_{RA} \cup \Sigma_{rA} \cup \Sigma_{RB} \cup \Sigma_{rB}$, as shown in Definition 3.10. Each transition label $(A(t_1), A(t_2))$ can be written as $\Sigma_{t_1}^a \cup \Sigma_{t_1}^b \cup \Sigma_{t_1}^f \cup \Sigma_{t_1}^r$, where $\Sigma_{t_1}^a \subseteq \Sigma_{RA}$, $\Sigma_{t_1}^b \subseteq \Sigma_{RB}$ and $\Sigma_{t_1}^f \subseteq \Sigma_{rB}$. If the following conditions:

- For each transition label $(A(t_1), A(t_2))$, we have $\Sigma_{t_1}^a = \Sigma_{t_1}^b$ and $\Sigma_{t_1}^f = \Sigma_{t_1}^r = \emptyset$; or $\Sigma_{t_1}^b = \Sigma_{t_1}^f = \emptyset$, and $\Sigma_{t_1}^a = \emptyset$.

- In each state $s_{i,j,k} \in S^R$ where $k = (k_1, k_2)$, we have $k_1 = 0$ and $k_2 = 0$.

are both satisfied, then $P_1 || P_2$ is a redundant raw converter.

If a raw converter is redundant then the given protocols are compatible even without the raw converter.

Example 3.18 The raw converter $Q_1 || Q_2$ shown in Figure 5 is redundant.

3.3 Converter

The raw converter defined in Definition 3.10 may contain a very large number of states because the buffer values are included in the states. For example, Avnit et al. (2008), we merge states by extracting these buffer values from each state and making them conditions in the corresponding transitions. The result of this merge operation is the actual converter. In general, two states $s_{i,j,k}$ and $s_{i,j,k'}$ in a raw converter are merged to one state $s_{i,j}$ in the converter. The converter has precisely the same behaviour as the raw converter w.r.t. the given protocols.

Another advantage of this process is that the user can increase the buffer size to increase the data transfer rate (for example), but not increase the number of states in the converter (this is future work).

Definition 3.19 Converter

We are given a raw converter $P_1 || P_2 = (S^R, \Psi^R, I^R, \Sigma^R, \delta^R, i^R, f^R)$ and two buffers with size $q_1$ and $q_2$. The raw converter states $S^R \subseteq S^n \times K'$, where $K' = K'_1 \times K'_2$. Each state $s_{i,j,k} \in S^R$ where $k = (k_1, k_2)$. As $i^R = (i', 00)$, $f^R = (f', 00)$ and $\Sigma^R = \Sigma_{RA} \cup \Sigma_{rA}$ as shown in Definition 3.10 and Definition 3.6. A converter $P_1 || c \cdot P_2$ is defined by an extended FSM $(S^C, L^C, \Sigma^C, \delta^C, i^C, f^C)$, where:

- $S^C = S^n$ is a finite set of states, where each state $s_{i,j}$ in $S^C$ is merged from a set of states in the raw converter $S_{i,j} = \{s_{i,j,k}|s_{i,j,k} \in S^R\}$, which are called source states. For all states in $S_{i,j}$, we write the set of all their outgoing transitions as $T_{i,j} = \{t|t_m \forall s_{i,j,k} \in S_{i,j}, \delta^R(s_{i,j,k}, A(t), A(t_m)) \in S^R\}$.

- $\Sigma^C = \Sigma^R$ is a set of actions.

- $L^C$ is a labelling function: $S^C \times 2^{\Sigma^a} \times 2^{\Sigma^b} \rightarrow (2^{K'_1}, 2^{K'_2})$, where for each $s_{i,j} \in S^C$ we have its source states $S_{i,j}$ with transitions $T_{i,j}$. Then, for each $t_m \in T_{i,j}$, we have $L^C(s_{i,j}, A(t), A(t_m)) = (L_1, L_2)$ where $L_1 = \{k_1|s_{i,j,k_1,k_2} \in S_{i,j}, \delta^R(s_{i,j,k_1,k_2}, A(t), A(t_m)) \in S^R\}$, $L_2 = \{k_2|s_{i,j,k_1,k_2} \in S_{i,j}, \delta^R(s_{i,j,k_1,k_2}, A(t), A(t_m)) \in S^R\}$ are sets of natural numbers.

- $i^C = i'$ and $f^C = f'$.

Example 3.20 Consider the protocols $U_1$ and $U_2$ shown in Figure 7, and the corresponding raw converter $U_1 || U_2$ shown in Figure 8. We can derive a corresponding converter $U_1 || c \cdot U_2 = (S^C, L^C, \Sigma^C, i^C, f^C)$ shown in Figure 9, where $S^C = \{s_{1,1}, s_{1,2}, s_{2,1}, s_{2,2}, s_{1,4}, s_{2,4}\}$, $\Sigma^C = \{a, r, d, rout, dout, a, d\}$, $L^C = \{c, \{s_{1,1}, \tau(r) \rightarrow 0, 1\}, \{s_{1,1}, \text{rout}(r) \rightarrow 0, 1\}, \text{etc}\}$, $i^C = \{s_{1,1}, \text{rout}(r) \rightarrow 0, 1\}$, $f^C = \text{rout}(r) \rightarrow 0, 1$.
Figure 7: Protocols $U_1$ and $U_2$ that produce and consume data

Figure 8: A raw converter $U_1|\parallel_R U_2$ with $q = 1$

Figure 9: A converter $U_1|\parallel_C U_2$

Figure 10: The formal derivation of the converter.

Figure 11: The width of data to and from the buffers $s_{1,1}, \text{etc}$, $i^c = f^c = s_{1,1}$. Here, $i..j$ denotes a value in the range from $i$ to $j$.

4 Implementation

In Figure 10 we graphically represent the formal derivation of the converter. In this figure, we present:

1. protocols (Definition 3.1)
2. parallel composition (Definition 3.4)
3. extended parallel composition (Definition 3.6)
4. raw converter (Definition 3.10)
5. correctness conditions (Definition 3.16)
6. converter (Definition 3.19)

To implement the converter, we first need to generate the PC. We then add counter values and temporal formulae to each state in the PC to generate the EPC. The EPC is pruned by enforcing the correctness conditions, and placing bounds on the counters, resulting in a raw converter. The width of data in each of the protocols plays an important role in calculating the bounds on the counters. These bounds determine the sizes of the buffers.

4.1 Buffers

We assume the buffers are arranged between the protocols as shown in Figure 11. In the figure, protocol $P_1$ sends and receives data that is $x$ bits wide, and protocol $P_2$ sends and receives data that is $y$ bits wide.

Let us consider $Buffer1$ in this figure. The values of $x$ and $y$ will determine the size of the buffer $q_1$, in the following way:
$q_1 = 0$ if one of the following cases is true:

- at least one of $d_s$ and $d'_s$ is $\emptyset$
- $x = y$, $d_s$ and $d'_s$ are non-empty, and in the parallel composition:
  - $d_s$ and $d'_s$ always occur together in the same transition
  - $d_s$ and $d'_s$ sometimes occur in the same transition, and only $d_s$ occurs in some other transitions
  - $d_s$ and $d'_s$ sometimes occur in the same transition, and only $d'_s$ occurs in some other transitions

$q_1 = [y/x] \times x$ if $x < y$
$q_1 = x$ if $x > y$; or $x = y$ and $d_s$ and $d'_s$ always occur in different transitions

As $x$, $y$ are both finite, $q_1$ must be finite. If $q_1 = 0$ then this buffer is not necessary of course. We can similarly calculate the minimum size $q_2$ of Buffer2.

4.2 Computing the EPC

The number of states in the PC (see Definition 3.4) is $|S_{P_1}| \times |S_{P_2}|$, where $|S_{P_1}|$ and $|S_{P_2}|$ are the (finite) number of states in $P_1$ and $P_2$ respectively. The EPC (Definition 3.6) may contain an infinite number of states due to the presence of the buffers. To generate the EPC, we apply an on-the-fly, depth-first traversal algorithm to the PC. During the traversal, if we reach a state $s_{i,j,k}$ via a transition labelled with $d_i$ (say), then the counter $k_1 = k_1 + x$ because $x$ bits have been read. Similarly, if the label is $d'_i$, then $k_1 = k_1 - y$ because $y$ bits have been sent. The new counters $k_1$ and $k_2$ are added to $s_{i,j,k}$ to produce an EPC state $s_{i,j,k}$. If $s_{i,j,k}$ does not exist in the EPC, the state is added, otherwise the existing state $s_{i,j,k}$ is updated by adding another outgoing transition. The algorithm terminates when no more changes can be made to the states, corresponding to a fixed point. The resulting number $|S'|$ of states in the EPC will be at most $|S| \times q_1 \times q_2$.

Checking the correctness conditions for each state is an important task of the algorithm.

4.3 Checking the correctness conditions

If any state in the EPC violates any of the correctness conditions $c_1 \ldots c_5$, then we prune (i.e. remove) the state together with all of its transitions. We check each condition in the following way.

$c_1$ Following Sinha et al. (2006), we apply the tableau method to model check the temporal formulae.

$c_2$ We can check that each state eventually reaches the final state by checking the formula:

$\phi_1: \forall s_{i,j,k} \in S^*, AG(s_{i,j,k} \rightarrow AXAF(f^0,0))$

$c_3$ The counters are checked whenever they are modified in a state.

$\phi_2: \forall s_{i,j,k},k_1,k_2 \in S^*, (0 \leq k_1 \leq q_1) \land (0 \leq k_2 \leq q_2)$

The condition that all received data will eventually be sent is checked by $\phi_1$.

$c_4$ We check that a raw converter can eventually execute data and control actions using the formula:

$\phi_3: AG(\bar{\phi} \rightarrow (AXAF\Sigma_0))$

where $\Sigma_0 \subseteq \Sigma^*$ is a non-empty set of data and control actions in the EPC.

5 Experiments

Algorithms corresponding to the definitions in this work have been implemented in JAVA 1.6. The system comprises approximately 3000 lines and runs on a PC with an Intel Pentium CPU 3.40 GHZ. We carried out 2 series of experiments. In the first series we studied the size of the raw converter for different protocol data widths and different sizes of buffer. In the second series, we compared the size of a raw converter synthesised by our model with those from previous work.

5.1 Size of the converter

To illustrate the relationship between the size of the raw converter (i.e. $|S|$) and the size of the buffers (i.e. $q$) we have synthesised converters for the protocols $U_1$ and $U_2$ shown in Figure 7. We do this for 5 different data-width ratios: the protocols have the same data width (i.e. 1:1), one protocol’s data width is half of the other’s (1:2 and 2:1), and one is one third of the other (1:3 and 3:1). We plot the results for the raw converter $U_1 || U_2$ in Figure 12 and for the converter $U_1 || U_2$ in Figure 13. We see in Figure 12 that as the size of the buffer increases, the size of the raw...
The state space is sometimes invalidated by protocols. This is explored in more detail by Sinha et al. (2006, 2008).

Case Study 1: We have synthesised converters for protocols taken from the literature. The results are presented below.

5.2 Comparison with previous work

The column labelled $A^*$ denotes the work by Avnit et al. (2007, 2008) and shows the sizes of the converters from these authors. The difference between our converter and theirs is our converter has one less state. The state we are 'missing' is $s_{31}$, which has a buffer of size 1, has 9 states. However, also notice that if there is no buffer, then a 3-state converter exists for the 1:1 case, but no converter exists for each of the other cases.

The data in Figure 13 shows that endlessly increasing the size of the buffer will not necessarily change the size of the converter, which in all cases requires at most 6 states. However, also notice that if there is no buffer, then a 3-state converter exists for the 1:1 case, but no converter exists for each of the other cases.

Figure 13: The relationship between buffer size $q$ and the number of states $|S|$ for the converter $U_1|c|U_2$ with different ratios of data widths

Note that a raw converter that includes a buffer may require more states than a raw converter without a buffer. For example, the raw converter in Figure 8, which has a buffer of size 1, has 9 states. If we removed the buffer, the raw converter would have just 3 states. Clearly, it is not the existence of the raw converter that determines the minimum size of the buffer, because there is a raw converter even with no buffer.

The previous authors synthesise a converter for these protocols that has only 5 states. However, they do not deal with data-width mismatches explicitly. Furthermore, their converter is very restricted in that it has no explicit buffer(s), and they deal with data and control signals only in a single direction, so their converter cannot be compared with ours.

Case Study 2: Bhaduri & Ramesh (2008), D’Silva et al. (2004a) studied a pipeline protocol (consisting of 6 states) and a handshake protocol (consisting of 4 states). The protocols are not reproduced here.)

We synthesise converters for the data widths 1:1, 1:2 and 2:1. The results are shown in Table 1. In this table, $|S^R|$ and $|S^C|$ denote the number of states in the raw converter and converter (resp.). We see that if we assume different data widths in the protocols, then we generate a larger raw converter, but this raw converter results in a converter of the same size. The values of $q$ correspond to the minimum size buffer for a raw converter to exist.

The previous authors synthesise a converter for these protocols that has only 5 states. However, they do not deal with data-width mismatches explicitly. Furthermore, their converter is very restricted in that it has no explicit buffer(s), and they deal with data and control signals only in a single direction, so their converter cannot be compared with ours.

Case Study 3: Avnit et al. (2007) synthesised a raw converter for the simple protocols shown in Figure 15. If we assume a data-width ratio of 1:1, we can synthesise a raw converter $P_1|p|P_2$ using two buffers each of size 1 (one for data and another for addresses). We also synthesised converters for these protocols assuming data-width ratios 1:2 and 2:1. We show the results in Table 2.

The column labelled $A^*$ denotes the work by Avnit et al. (2007, 2008) and shows the sizes of the converters from these authors. The difference between our converter and theirs is our converter has one less state. The state we are 'missing' is $s_{31}$ in their work, which is sometimes unreachable: for ex-

Table 1: The sizes of converters for protocols in Case Study 2

| $x$ | $y$ | $q_1$ | $|S^R|$ | $|S^C|$ |
|-----|-----|-------|-------|-------|
| 1   | 1   | 1     | 24    | 19    |
| 2   | 2   | 2     | 31    | 19    |
| 3   | 1   | 2     | 40    | 19    |

Figure 14: A producer/consumer protocol from Case Study 1

Figure 15: Two simple protocols from Case Study 3

Figure 14: A producer/consumer protocol from Case Study 1

Figure 15: Two simple protocols from Case Study 3

The state space is sometimes invalidated by protocols. This is explored in more detail by Sinha et al. (2006, 2008).

Case Study 1: We have synthesised converters for protocols taken from the literature. The results are presented below.

5.2 Comparison with previous work

The column labelled $A^*$ denotes the work by Avnit et al. (2007, 2008) and shows the sizes of the converters from these authors. The difference between our converter and theirs is our converter has one less state. The state we are 'missing' is $s_{31}$ in their work, which is sometimes unreachable: for ex-
Table 2: The sizes of converters for the protocols shown in Figure 15

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>q1, q2</th>
<th>Sx</th>
<th>Sy</th>
<th>Ax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1, 2</td>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2, 8</td>
<td>8</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 3: The sizes of converters for ASB and APB protocols from Case Study 4

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>q1, q2</th>
<th>Sx</th>
<th>Sy</th>
<th>Ax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1, 2</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1, 2</td>
<td>21</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2, 1</td>
<td>9</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2, 3</td>
<td>31</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1, 4</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4, 2</td>
<td>29</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

...
lay the theoretical foundations on which the algorithmic layer can be built.

The results of our experiments has shown our methodology to be more flexible, and result is smaller converters (all things considered) than in previous work. Our ability to handle data-width mismatches and buffer sizes explicitly means the user can design a converter with different configurations, different properties, and consequently different performances. The model of data width that we have used (shown in Figure 11) is somewhat arbitrary. More complex models that allow different data widths for different data and address buffers (say) are also possible.

The algorithm that determines the minimum buffer size should consider more than just the (ratio of) data widths of the given protocols. There are other factors, such as the data transfer rate and protocol behaviours that may mean a converter will exist, or not. Currently, our buffer-size algorithm may under- or over-estimate the size that the buffer needs to be. This issue needs to be addressed in the future.

While we have studied the relationship between nondeterminism and the invalid data problem, we cannot yet identify a priori which nondeterminism results in 'bad data', and which does not.

There are practical issues in using the finite-state machine formalism: is it expressive enough to model protocol (or in general component) behaviour, and satisfy real chip designers for example? We have not yet carried out a complexity analysis of our algorithms, but do not expect this to be an issue however as the size of typical converters is not expected to be large. We have plans to extend this work to an asynchronous setting as well.

References


