# A Model Checking based Converter Synthesis Approach for Embedded Systems

Roopak Sinha
Department of Electrical and Computer Engineering
University of Auckland
rsin077@ec.auckland.nz

Partha S. Roop
Department of Electrical and Computer Engineering
University of Auckland
p.roop@ece.auckland.ac.nz

Samik Basu
Department of Computer Science
Iowa State University
sbasu@cs.iastate.edu

November 2006

**Techreport: 537** (Iowa State University)

©2006 All rights reserved

## A Model Checking based Converter Synthesis Approach for Embedded Systems

**Roopak Sinha  Partha Roop  Samik Basu**

**Abstract**

Protocol conversion problem involves identifying whether two or more protocols can be composed with or without an intermediary, referred to as a converter, to obtain a pre-specified desired behavior. We investigate this problem in formal setting and propose, for the first time, a temporal logic based automatic solution to the *convertibility verification and synthesis*. At its core, our technique is based on local model checking technique and determines the existence of the converter and if a converter exists, it is automatically synthesized. A number of key features of our technique distinguishes it from all existing formal and/or informal techniques. Firstly, we handle both data and control mismatches (for the first time), using a single unifying model checking based solution. Secondly, the proposed approach uses temporal logic for the specification of correct behaviors (unlike earlier automaton based specifications) which is both elegant and natural to express event ordering and data-matching requirements. Finally, we have have experimented extensively with the examples available in the existing literature to evaluate the applicability of our technique in wide range of applications.

# 1 Introduction

Application-specific and ubiquitous computers, called embedded systems, are often designed by reusing several pre-designed blocks to reduce designer effort and verification time while improving the overall design quality. During this process, a designer puts together a set of intellectual property (IP) blocks (using some well known tools and design methodology). For example, a system-on-a-chip (SoC) is built by reusing IPs connected using a central bus such as AMBA [DRS04]. A major problem with this reuse is the inherent *mismatch* between protocols of IPs. This has been an active area of research for over two decades [BK87, Gre86], even before SOCs came into being (because of earlier component-reuse techniques). Mismatches occur because components are developed independently without any intention of eventual integration, and can result from control signal mismatches [Gre86], inconsistent naming conventions [SL89], different clock speeds and difference in data-widths [DRS04].

**Problem Definition.** Mismatches are corrected, if possible, by synthesizing extra glue-logic, called a *converter*[PdAHSV02] to control communication between two given protocols in order to satisfy given specifications. Figure. 1 presents the protocol conversion problem between two protocols $P_1$ and $P_2$. The reader protocol $P_1$ has three control inputs `next`, `ack` and `more` and has one control output `req`. Its data input port $\texttt{DIn}_8$ is $8$-bits wide. On the other hand, the writer protocol $P_2$ has one control output `ack`, two control inputs `req` and $reset$, and a data output port $\texttt{DOut}_{16}$ of size $16$. $P_1$ and $P_2$ can share the control signals `req` and `ack` to communicate with each other. However, the input $reset$ for $P_2$ or the inputs `more` and `next` for $P_1$ are not emitted by the dual protocol. Also, the data ports $\texttt{DIn}_8$ and $\texttt{DOut}_{16}$ do not have the same sizes which may result in lossy data

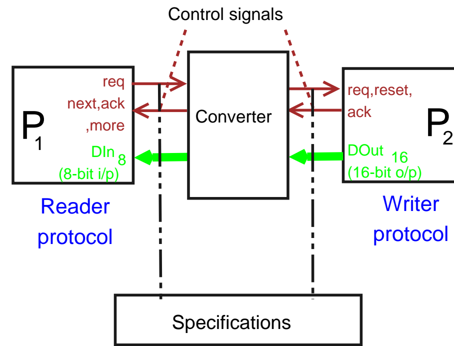


Figure 1: Convertibility Verification Problem

communication. Furthermore, the unrestricted interaction between the two protocols may lead to faulty execution paths or to blocking states (for example, where $P_2$ is waiting for the input $reset$ not provided by $P_1$).

Given the above inconsistencies between $P_1$ and $P_2$, it may be required to *restrict* or *guide* their communication in such a way that intended behavior of the combined system can be achieved. These restrictions may include control and/or data constraints (also known as *specification* as shown in Figure 1). For example, an appropriate control constraint could require that control signals between the protocols are exchanged in specific orders (e.g., a request `req` from $P_1$ is always followed by an acknowledge `ack` from $P_2$ before any any data can be read by $P_1$). Similarly, a data constraint may require that $P_2$ never tries to write more data for $P_1$ before $P_1$ has read all data available to it previously (to prevent overflows). Given the description of the protocols and the desired specifications (which capture control/data constraints), protocol conversion investigates whether a converter can be generated which can guide protocol interaction in order to satisfy the given constraints.

**Related Work.** Protocol conversion has been studied extensively in literature and many different solutions have been proposed. Broadly, approaches can be categorized as either being informal (algorithmic but lacking in any mathematical rigor) or formal (based on some mathematical proof technique such as refinement or model checking).

Protocol conversion was first proposed in [Gre86] which focused on the need for protocol conversion and proposed informal ways of handling mismatches. Early work mostly proposed ad-hoc solutions such as converters for protocol gateways [Boc90] and interworking networks [BI89] have been proposed. For embedded systems, initial informal solutions were proposed by Borriello et al. [BK87], where timing diagram based specifications of protocols were used. Subsequently, Narayan and Gajski [NG95] proposed an algorithm that could bridge both control mismatches and data-width mismatches. However, the algorithm is fairly limited due to the following: firstly, the protocol descriptions were only linear traces of automata where no branching was allowed. Secondly, the algorithm could bridge only limited data-width mismatches where data ports had to be exact multiples. Finally, the algorithm had no way of answering the fundamental question: given protocol descriptions of $P_1$ and $P_2$ how do we determine if a converter exists that can bridge the mismatches.

Unlike these ad-hoc solutions, formal approaches try to provide an answer to the above question using some rigorous technique. These require protocols and specifications to be represented formally and use mathematical proofs to address the protocol conversion problem [GBB$^+$06, DRS04, PdAHSV02]. [PdAHSV02] presents a formal approach where protocols are described using finite automata. The desired specifications (which capture only control constraints in their setting) are also described as an automaton. A game-theoretic formulation is used to check for the existence of a converter and also to generate a converter. While this formulation presents, for the first time, an elegant formal verification based solution, it suffers from several limitations. Firstly, the protocol descriptions use automaton that do not distinguish between inputs and outputs and may be triggered by either of them (this can be extremely problematic as justified in section 2.3). Secondly, the protocol conversion problem only solved *control mismatches of half-duplex protocols* and no solutions to the data-width mismatch problem was provided. Finally, no practical protocol conversion tool was developed for benchmarking and evaluating the proposed approach. [DRS04, dRS05] present synchronous protocol automata, which is a much better representation compared to earlier approaches as it faithfully models clocks while distinguishing between inputs and outputs. Authors propose a compatibility relation (similar to other well known refinement and bisimulation like relations) to check for protocol compatibility. However, no solution for data-width mismatches was proposed and no data and control constrains could be captured (though authors performed an additional pass of model checking to ensure converter correctness). In [GBB$^+$06], a hybrid simulation/verification approach to protocol conversion in SoC designs is proposed, where both simulation and formal verification is combined. However, the proposed approach also suffers from similar limitations.

The protocol matching problem seems superficially similar to the problem of controller synthesis in discrete event systems (DES) [RW89]. Early approaches in DES [KG95] were limited to both plant and specifications being described using labeled transition systems (a type of automata-based specification). More recently, specifications have been captured in temporal logics, which are a very natural way of describing requirements involving *liveness*, *safety* and *fairness*. In [Ant95] the logic CTL is used to describe specifications. In [JK06], the logic CTL$^*$ is used for specifications while the plant is modeled as an LTS. Then, an automatic synthesis approach for the controller is presented. Authors show that the DES control problem for CTL$^*$ specifications is equivalent to the module checking problem [KVW01]. While both DES approach and converter synthesis approach try to synthesize a controller (or a converter), there are fundamental differences between the two approaches. Firstly, the in DES setting a controller may only *disable events* in the plant to ensure that all desired specifications are met. This is unlike the role of the converter that performs disabling of transitions in addition to performing other desirable actions such as the buffering of an event for later forwarding, and the generation of extra control signals. Secondly, in the DES setting, there is no need for bridging data-width and clock mismatches, which are critical requirements for convertibility verification.

**Contributions of our solution.** In this paper, for the first time, we provide an uniform solution mechanism for convertibility verification and converter synthesis using the well studied model checking based approach. The key issues while developing a solution and the main contributions of our approach are summarized below:

1. How to represent the protocols $P_1$ and $P_2$?: we introduce the notion of *synchronous Kripke structure* (SKS) to represent protocols. SKS can distinguish between inputs and outputs and

also can effectively model data-widths. Hence, in our opinion, they provide an ideal modeling paradigm for IP protocols. Unlike synchronous protocol automata [DRS04] which is primarily designed for checking for the existence of a compatibility relation (a refinement based solution) between two protocols, our model is ideal for a model checking based solution and is created to tackle both control and data-width mismatches.

2. How to describe the required specifications?: we develop a temporal logic based formulation where the temporal logic ACTL is used to describe both control-constraints and data-constraints. We believe that temporal logics are the most natural way of writing these requirements succinctly and effectively. We use ACTL since these control and data constraints need to be always universally quantified. LTL was not considered (though it is more expressive) because of much higher complexity of LTL convertibility verification (which is PSPACE complete [KVW01]).
3. What kind of approach for convertibility verification and converter synthesis is required?: we develop a local model checking based formulation for both convertibility verification and converter synthesis. In our approach, we perform disabling, buffering and event forwarding in composite system ($P_1||P_2$). More critically, we also perform the checking of data-constraints using a formulation similar to [AD94] using bounded integer data-counters that are valid only in specified bounded regions. Unlike previously known approaches to data-constraint matching during protocol conversion, our approach can deal with arbitrary data-widths of the two protocols (earlier solutions required data-widths to be multiples of each other).

**Organization.** The rest of this paper is organized as follows. Section 2 presents synchronous Kripke structure and temporal logic `ACTL` used for describing the behavior and desired properties of protocols in our setting. Section 3 presents the model checking based algorithm for converter synthesis. Section 4 provides implementation results with concluding remarks in section 5.

# 2 Model, Specification & Composition

For converter synthesis, three types of mismatches need to be considered: control, data-width and clock. In the paper we concentrate only on the first two types of mismatches and assume that there are no clock mismatches[1]. Hence, in our setting, both the IPs and the converter use the same clock for synchronization. The proposed protocol conversion algorithm takes as input the synchronous Kripke structure (SKS) descriptions of two protocols and a set of ACTL properties representing the desired behavior of their composition. It then employs a local model checking based algorithm to verify the existence of a converter which regulates the behavior of protocols to (a) resolve the incompatibilities, if any and (b) ensure conformance to the desired behavior of the composition.

[1] We have also developed a technique for bridging clock mismatches using multi-clock Kripke structures (See http://www.cs.iastate.edu/∼sbasu/research/multiclock.pdf) which is not the subject matter of the current paper.

## 2.1 Protocol Description using SKS

We introduce Synchronous Kripke Structures (SKS) for protocol description. An SKS extends standard Kripke structures by introducing transition triggers and outputs and also has a notion of synchrony through a clock that is used for sampling all inputs.

**Definition 1 (SKS)** *A Synchronous Kripke structure (SKS) is a finite state machine represented as a tuple* $\langle AP, S, s_0, \Sigma_I, \Sigma_O, R, clk\ L\rangle$ *where* $AP$ *is a set of atomic propositions;* $S$ *is a finite set of states;* $s_0 \in S$ *is the initial state;* $\Sigma_I = \Sigma_I^+ \cup \Sigma_I^-$ *is a finite and non-empty set of all possible input events (where* $\Sigma_I^+$ *is the set of input events and* $\Sigma_I^-$ *is the set of negated events) and* $\Sigma_O$ *is the finite non-empty set of output events.* $\Sigma_I$ *and* $\Sigma_O$ *include the empty output event .;* $R \subseteq S \times \mathtt{t} \times \Sigma_I \times \Sigma_O \times S$ *is the transition relation; and* $L : S \rightarrow 2^{AP}$ *is the state labeling function. Finally, the event* $\mathtt{t}$ *represents the ticking of the* $clk$.

All transitions trigger with respect to the *ticks* of this clock. Moreover, each transition has an input trigger event. A transition will trigger only when the associated input event is *present* (in case the trigger is the negation of an event, the event must be *absent*) during the current *tick*. When a given transition triggers, a specific output (which is specified in the transition) is also generated by the protocol. Because of ideas of synchrony borrowed from synchronous languages [Lef05, BCE$^+$03], SKS can model both the presence and *absence* of events. In case no matching input triggers are present, the protocol remains in its current state. We use $\sim$ (e.g., $\sim a$) to represent the absence of input signals, $*$ to represent *any* input signal and use . to represent no signal. We represent the relations $(s, \mathtt{t}, i, o, s') \in R$ also as $s \xrightarrow{i/o}_{\mathtt{t}} s'$. We represent autonomous transitions that trigger on the *tick event* as $s \xrightarrow{./o}_{\mathtt{t}} s'$ (not requiring presence of any external stimuli, except the tick).

States of a SKS are labeled using atomic propositions exactly like standard Kripke structures. However, we use some atomic propositions that have an integer suffix (such as $\mathtt{DIn}_8$) to indicate data input or output over ports of specific widths. These are subsequently used by the our algorithm for updating counter values (to be illustrated in section 3).

Two protocols represented as SKS have finite sets of inputs and outputs. We require that at least some input signals in one protocol match with outputs emitted by the other ($\Sigma_{I1} \cap \Sigma_{O2} \neq \emptyset$, $\Sigma_{12} \cap \Sigma_{O1} \neq \emptyset$). This restriction allows for meaningful bi-directional exchange of signals between the two protocols, even under the control of a converter (described in the following section).

In our current setting, we assume that the same $clk$ is used for both the protocols and also the converter. This allows us to use well known results from the theory of synchronous languages to deal with issues such as *causality* and *constructiveness* [BCE$^+$03] effectively. For example, we reject transitions of the form $s_1 \overset{a/\sim a}{\rightarrow} s_1$ as semantically invalid. This assumption, will make protocol composition and converter synthesis simpler, compared to multi-clock systems.

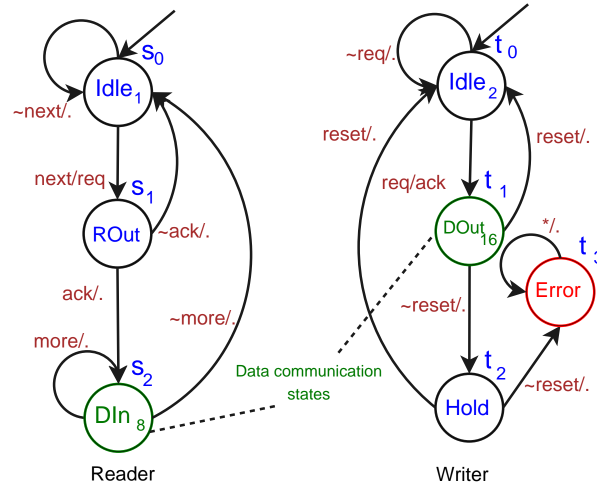


Figure 2: Reader-writer protocol pair

**Illustrative example: Reader-Writer protocols.** Figure. 2 shows the SKS representations of the reader and writer protocols (presented in Figure. 1), which are typical protocols in SoCs [RMK03]. The reader protocol $P_1$, in its initial state $s_0$ awaits the `next` input to start a new transaction. Once `next` is received, it sends a request (output `req`) to the writer and makes a transition to state $s_1$. In $s_1$, if the acknowledge input `ack` is present in the next clock-tick, the reader evolves to state $s_2$ (otherwise resets back to $s_0$). In $s_2$, the reader reads an 8-bit data packet (represented by the label $\texttt{DIn}_8$). If the input signal `more` is provided, a transition back to $s_2$ is made and more 8-bit data is read (otherwise, the protocol resets back to $s_0$).

The writer protocol $P_2$, on the other hand, awaits a request `req` in its initial state $t_0$ and once a request is received, emits the acknowledge output `ack` and moves to state $t_1$. On reaching $t_1$, the writer writes one 16-bit data for the reader (represented by $\texttt{DOut}_{16}$). It then awaits a reset signal $reset$ to go back to $t_0$. In case, $reset$ is not present in the next clock tick, it moves to state $t_2$. In case $reset$ is still not present, it moves to state $t_3$ representing error (labeled by `Error`).

These protocols have a number of incompatibilities:

1. *Control signal mismatches*: The reader protocol sends a request (`req`) in its state $s_0$ and awaits an acknowledge (`ack`) in state $s_1$ (after one tick). However, the writer generates an acknowledge as soon as a request is received. Unless the acknowledge signal is somehow *buffered*, the reader will never be able to read the acknowledge signal.
2. *Missing input signals*: The input signals `next` and `more` for the reader and the input $reset$ for the writer are not emitted by any protocol. Unless these are generated, the composite system of the two IPs will simply block.
3. *Data-width mismatches*: The reader reads data in 8-bit packets while the writer writes 16-bit data. Furthermore, the reader can read multiple packets in each transaction while the writer

can only write once per-transaction. Therefore, it is possible that the reader may attempt to read more data than is actually presented by the writer, resulting in an *underflow* (this can happen if some how many *more* inputs are asserted).

4. *Error states*: State $t_3$ in the writer protocol is an error state and once the protocol reaches it, not further communication with the reader is possible. Therefore it is essential to prevent the writer from making a transition to $t_3$ under all circumstances.

To detect and resolve these incompatibilities, we define of synchronous parallel composition of two SKS which represents all possible (*unrestricted*) behavior of the composition (including the behavior with incompatibilities).

**Definition 2 (Protocol Composition)** *Given two SKS* $P_1 = \langle AP_1, S_1, s_{0_1}, \Sigma_{I1}, \Sigma_{O1}, R_1, L_1, clk \rangle$ *and* $P_2 = \langle AP_2, S_2, s_{0_2}, \Sigma_{I2}, \Sigma_{O2}, R_2, L_2, clk \rangle$*, their parallel composition, denoted by* $P_1||P_2$ *is* $\langle AP_{1||2}, S_{1||2}, s_{0_{1||2}}, \Sigma_{I1||2}, \Sigma_{O1||2}, R_{1||2}, L_{1||2}, clk\rangle$ *where* $AP_{1||2} = AP_1 \cup AP_2$ *;* $S_{1||2} = S_1 \times S_2$*;* $s_{0_{1||2}} = (s_{0_1}, s_{0_2})$*; and* $\Sigma_{I1||2} \subseteq \Sigma_{I1} \times \Sigma_{I2}$*;* $\Sigma_{O1||2} \subseteq \Sigma_{O1} \times \Sigma_{O2}$*.* $R_{1||2} \subseteq S_{1||2} \times t \times \Sigma_{I1||2} \times \Sigma_{O1||2} \times S_{1||2}$ *such that*

$$(s_1 \xrightarrow{i_1/o_1}_t s_1') \ \wedge \ (s_2 \xrightarrow{i_2/o_2}_t s_2') \Rightarrow ((s_1, s_2) \xrightarrow{(i_1,i_2)/(o_1,o_2)}_t (s_1', s_2'))$$

*Finally,* $L_{1||2}((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$*.*

Note that in the composition, there may be some transitions which have triggers and outputs of the form $(a, b)/(b, a)$ i.e., the first protocol expects input $a$ while producing output $b$ while the second protocol expects input $b$ while producing output $a$. Such protocols, when composed synchronously, will produce *non-causal* [BCE+03] specifications - specifications that do not have a precise meaning in the synchronous setting. Protocol compositions that are non-causal will also be semantically rejected (using standard causality analysis tools such as [Lef05]). Figure. 3 presents the synchronous composition of protocols in Figure. 2[2].

## 2.2 ACTL Specifications

Once the composition of protocols is generated, the next step is to describe the the intended communication between the participating protocols. This is performed using (ACTL) which is a branching time temporal logic with universal path quantifiers. It is defined over a set of propositions using temporal and boolean operators as follows:

$$\phi \rightarrow P \mid \neg P \mid tt \mid ff \mid \phi \wedge \phi \mid \phi \vee \phi \mid \texttt{AX}\phi \mid \texttt{A}(\phi \ \texttt{U} \ \phi) \mid \texttt{AG}\phi$$

[2]Transitions in Figure. 3 are color-coded as follows: red transitions lead to error states, blue transitions reset the composition to its initial state, while remaining transitions appear in black.

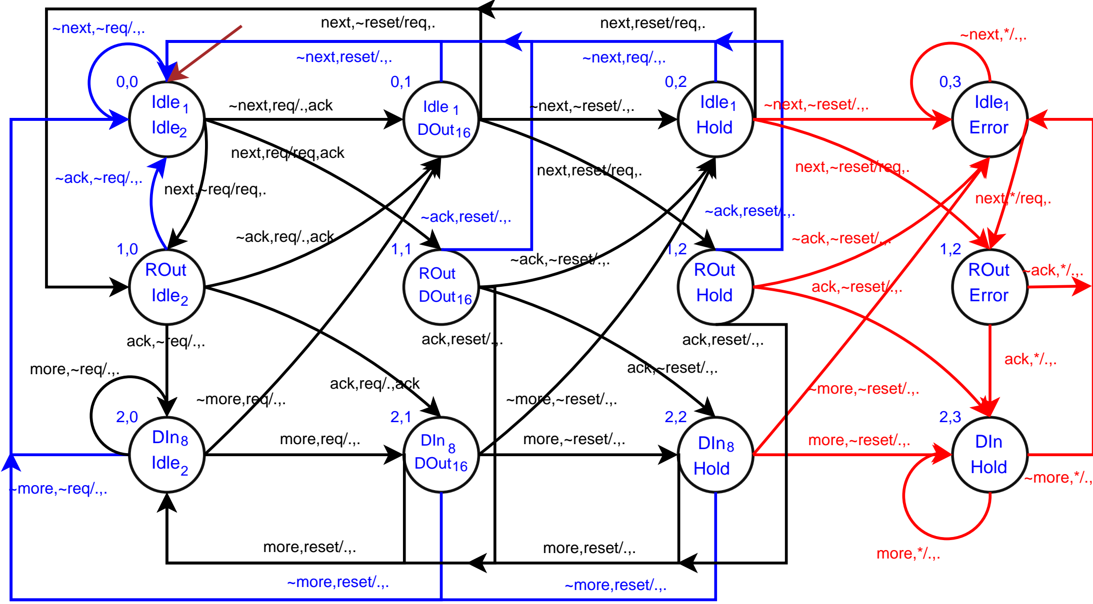


Figure 3: Parallel Composition $P_1||P_2$ of the reader-writer protocol pair

$$
\begin{array}{ll}
1: & [\![p]\!] = \{s \mid p \in L(s)\} \quad 2: [\![\neg p]\!] = \{s \mid p \notin L(s)\} \\
3: & [\![tt]\!] = S \quad 4: [\![ff]\!] = \emptyset \\
5: & [\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!] \quad 6: [\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!] \\
7: & [\![\mathtt{AX}\varphi]\!] = \{s | \forall s \longrightarrow_{\mathtt{t}} s' \wedge s' \in [\![\varphi]\!]\} \\
8: & [\![\mathtt{A}(\varphi \ \mathtt{U} \ \psi)]\!] = \{s | \forall s = s_1 \rightarrow s_2 \rightarrow \ldots \\
 & \qquad \wedge \ \exists j.s_j \in [\![\psi]\!] \ \wedge \forall i < j.s_i \in [\![\varphi]\!]\} \\
9: & [\![\mathtt{AG}\varphi]\!] = \{s | \forall s = s_1 \rightarrow s_2 \rightarrow \ldots \wedge \forall i.s_i \in [\![\varphi]\!]\}
\end{array}
$$

Figure 4: Semantics of ACTL

Semantics of an ACTL formula, $\varphi$ denoted by $[\![\varphi]\!]_M$ are given in terms of set of states in a Kripke structure (or a KS), $M$, which satisfies the formula (see Figure. 4). A state $s \in S$ is said to satisfy a ACTL formula $\varphi$, denoted by $M, s \models \varphi$, if $s \in [\![\varphi]\!]_M$. Typically, the context of the semantics, i.e., $M$ in $[\![\ ]\!]_M$ is implicit, and omitted. We also say that $M \models \varphi$ to indicate $M, s_0 \models \varphi$. In this paper, we restrict ourselves to formulas where negations are applied to propositions only.

## 2.3 Control Constraints in ACTL

For the producer-consumer example in Figures. 2 and 3 the following control-constraints of their composition are specified in ACTL:

[$\varphi_1$] $\texttt{AG}(\neg\texttt{Error})$: The writer never enters the error state $t_3$.

[$\varphi_2$] $\texttt{AG}((\texttt{Idle}_1 \wedge \texttt{Idle}_2) \Rightarrow \texttt{AX}(\neg\texttt{DOut}_{16} \vee \texttt{ROut}))$ When both protocols are in their initial states, the writer cannot write data before a request is received from the reader.

[$\varphi_3$] $\texttt{AG}(\texttt{DOut}_{16} \Rightarrow \texttt{A}(\neg\texttt{Idle}_1\ \texttt{U}\ \texttt{DIn}_8))$: Once data is written by the writer, the reader does not reach its initial state before reading data written by the writer.

[$\varphi_4$] $\texttt{AG}(\texttt{Idle}_2 \Rightarrow (\texttt{Idle}_1 \vee \texttt{A}(\neg\texttt{DOut}_{16}\ \texttt{U}\ \texttt{Idle}_1)))$: Once the writer resets back to its initial state, the reader must also reach its initial state before more data can be written.

[$\varphi_5$] $\texttt{AG}(\texttt{ROut} \Rightarrow (\texttt{DOut}_{16} \vee \texttt{A}(\neg\texttt{DIn}_8\ \texttt{U}\ \texttt{DOut}_{16})))$: Once a request is made by the reader, it waits for the writer to write data before reading.

Note that control-constraints that involve AU formulas will result in the generation of empty converters if protocol descriptions have transitions which may be triggered by outputs (label transition systems such as used in [PdAHSV02]). This is because, a converter has no control over output transitions and will not be able to disable them. Hence, if the protocols have loops that are triggered by outputs alone, AU properties will fail to hold. This is one of the motivations of developing SKS with i/o-transitions (instead of transitions with single labels; either input or output but not both) to describe protocols in this paper.

## 2.4 Data Constraints in ACTL

The above properties describe desired behavioral sequence of the composition. However, they do not take into consideration the data-width mismatches of the protocols resolution of which may also require some desired behavioral pattern of the composition.

As discussed earlier, the reader-writer protocol pair has a data-width mismatch as $P_1$ reads 8-bit data while the writer $P_2$ writes 16-bit data. In order to restrict the protocols such that the data-communication between the two never results in an *underflow* or an *overflow*, we introduce *data counters* as follows.

First, we formally describe the desired data-communication behavior. Given the data-widths $N$ and $M$ (which are integers) of the outputs and inputs respectively, we compute the minimum width needed for the communication medium (usually a buffer) between the two protocols. If $N < M$, then the minimum capacity must be $N \times f$ such that $f$ is the smallest integer for which $N \times f \geq M$; otherwise the minimum capacity is $N$. This assumption is needed to ensures that there are enough preceding outputs before any one input. While the minimum bound of communication medium buffer can be computed as above, the maximum bound is be any value greater than the minimum bound. In our setting, we assume that the maximum bound of the communication medium buffer is $\texttt{LCM}(N, M)$. Given a capacity $K$ of the communication medium between these bounds, the maximum number of outputs possible when the medium is empty is $x = \lfloor K/N \rfloor$; while the maximum number of inputs possible when the medium is full is $y = \lfloor K/M \rfloor$. We use an auxiliary counter for every input/output pair such that the counter is *incremented* by $y$ for every output and *decremented* by $x$

for every input. We then verify that the counter always remains between $0$ and $x \times y$ using the *global* ACTL property $\mathtt{AG}(0 \leq counter \leq (x \times y))$.

**Reader-Writer Example Revisited.** Consider the reader-writer example in Figure. 2. Data outputs from the reader ($\mathtt{DOut}_{16}$ in $P_2$) are 16-bits while data inputs by the reader ($\mathtt{DIn}_8$ in $P_1$) are 8-bits. Hence, $N = 16$ and $M = 8$. As $N > M$, we set the communication medium capacity to 16-bits. Given this $16$-bit capacity of the communication medium, the maximum number of write operations possible when the medium is empty is $x = \lfloor K/N \rfloor = 1$. Similarly, the maximum number of read operations is $y = \lfloor K/M \rfloor = 2$. Given these values for $x$ and $y$, we introduce a counter $counter$ which is incremented by $2$ ($y$) every time a $\mathtt{DOut}_{16}$ is encountered and is decremented by $1$ ($x$) when a $\mathtt{DIn}_8$ is encountered. To verify that the counter always remains between $0$ and $x \times y = 2$, we use the following property $\varphi_d \equiv \mathtt{AG}(0 \leq counter \leq 2)$.

## 2.5 Converter as SKS

Recall that (Figure. 1), a converter acts as an intermediary between the participating protocols in a composition. Inputs to the converter are outputs from the protocols while outputs from the converter act as inputs to the protocols. In addition to relaying outputs from one protocol to the inputs of another, the converter is also capable of generating missing inputs to protocols. As the converter is providing the inputs to the protocols, it can effectively regulate the behavior of the protocols by disabling or enabling transitions in the protocols by not providing or providing the appropriate triggering inputs of the transitions.

**Definition 3 (Converter)** *Given protocols $P_1 = \langle AP_1, S_1, s_{0_1}, \Sigma_{I1}, \Sigma_{O1}, R_1, L_1, clk \rangle$ and $P_2 = \langle AP_2, S_2, s_{0_2}, \Sigma_{I2}, \Sigma_{O2}, R_2, L_2, clk \rangle$, a converter $\mathcal{C}$ for $P_1$ and $P_2$ is an SKS: $\langle AP_\mathcal{C}, S_\mathcal{C}, s_{\mathcal{C}0}, \Sigma_{\mathcal{C}I}, \Sigma_{\mathcal{C}O}, R_\mathcal{C}, L_\mathcal{C}\rangle$ where $AP_\mathcal{C} = \emptyset$, $\Sigma_{\mathcal{C}I} \subseteq (\Sigma_{O1} \times \Sigma_{O2})$ and $\Sigma_{\mathcal{C}O} \subseteq (\Sigma_{I1} \times \Sigma_{I2})$.*

Observe that, the converter is not required to satisfy any propositional property ($AP = \emptyset$) as all desired propositional properties must be satisfied by the participating protocols.

**Definition 4 (Control consistent converter)** *A converter is said to be control-consistent if and only if in all paths from $s_{\mathcal{C}0}$ whenever there is a transition which outputs signal 'a' it is always preceded by a transition with input trigger on signal 'a' if $a \in \Sigma_{01} \cup \Sigma_{02}$, i.e., if a protocol is capable of providing a signal $a$, then the converter must not generate that signal.*

The protocols, when composed with the converter, can only make moves if the converter provides the appropriate triggering inputs to the protocols, i.e., if the composite protocol wants to make a move from state $s$ to $t$ where the transition is of the form $(i_1, i_2/o_1, o_2)$, then this move is only allowed when the converter state $c$ composed with $s$ has a transition on $(o_1, o_2/i_1, i_2)$.

# 3 Converter Synthesis Algorithm

The composition in Figure. 3 does not satisfy the desired ACTL specifications. A converter is necessary to regulate the composition behavior to avoid violation of the properties. For example, a candidate converter will disable protocol transitions from $(s_0, t_0)$ to $(s_0, t_1)$ on absence of next signal and presence of req signal because that will lead to violation of property $\varphi_2$ (see Section 2.3).

In this section, we present an algorithm to identify whether there exists a converter that can regulate a given composition to ensure conformance to the pre-specified desired properties. Our algorithm is inspired by (but is substantially different from) on on-the-fly CTL model checking ([BCG95]) and is presented in Algorithm 1. The function isConv takes as argument

1. the state s of the composite protocols,
2. the set of counter valuations I keeping track of data communications (see Section 2.4),
3. the set of formulas FS (referred to as obligations) representing the desired properties (control and data constraints) to be satisfied at that state,
4. a history set H which keeps track of visited state formula pairs to ensure termination of our algorithm, and
5. an event set E which keeps track of the outputs from the protocols that can be relayed by a candidate converter (consistent-converter). This is used for checking if a given input has been *buffered* by the converter to be relayed later.

The function returns a structure of type NODE which is the root of a graph witnessing whether or not the behavior starting from state s can be regulated/converted to ensure satisfiability of formulas in FS. s is said to be convertible if and only if the return of the function isConv is a non false node.

**Discussion.** If FS $= \emptyset$, then there is no obligation on the state s and the function returns a TRUE_NODE denoting the existence of converter which allows all possible behavior from s (Lines 1–3).

Lines 5–11 handles termination of our algorithm and will be discussed below. At Lines 4 and 12, a new node is created using s and FS information and the history set is updated to H_1 by inserting this new node. A formula F is removed from FS to create FS_1. If F is a propositional constant true then converter existence is checked against s with respect to FS_1 as any state can satisfy true (Lines 13, 14). On the other hand, s is not convertible if F is equal to false as no state can satisfy false (Lines 15, 16). Similarly, if F is a proposition $p$ ($\neg p$), s must satisfy $p$ (not satisfy $p$); otherwise converter does not exist (Lines 17–28).

If F is a conjunction of formula, convertibility is checked against s with respect to FS_1 and each of the conjuncts (Lines 29,30). On the other hand, for disjunctive formula, the function is recursively called for each disjunct and the return nodes (Line 33,34), if non-false, are attached to the current node as children (Lines 35–40). In that case, the current node type is updated to be OR_NODE (Line

32). If the return nodes are both false, then there exists no converter (Lines 41, 42) as none of the choices can be satisfied in the presence of any converter at state s.

If F is AG$\varphi$, the obligation on s is updated to include $\varphi \wedge$ AXAG$\varphi$, denoting $\varphi$ must be satisfied at s and all its (converter) permitted successors must satisfy AG$\varphi$ (Lines 46,47). Similarly, if F is equal to A($\varphi$ U $\psi$), the obligation on s is updated to include $\psi \vee (\varphi \wedge$ AXA($\varphi$ U $\psi$) (Lines 48,49).

The control reaches Line 51 only when none of the other rules are applicable, i.e., FS only contains formulas of the form AX, which captures the next state obligation of s. Accordingly, FS_AX aggregates all the formulas that must be satisfied in all destinations of (converter) permitted transitions from s. The converter permitted transitions are the ones which can be forced by a candidate converter by providing missing signals or the ones that can be enabled by the same by relaying outputs from one protocol to the input of the other (control-consistent converter, see Definition 3). This is captured in Line 54 where it is stated that if the input trigger $i_j$ can be provided by the output of a protocol ($i_j \in \Sigma_{O1} \cup \Sigma_{O2}$), then it must be present in the set of already seen signal-set E (i.e, is already buffered by the converter). If such a trigger is consumed from E to enable a transition, the E-set is appropriately updated in Line 55; the trigger actions used are removed and output signals are added ($\cup_{\texttt{nodup}}$ represents union without duplicates). Furthermore, the counter valuations (see Section 2.4) are updated appropriately to record data read/written at the new next state s' (Line 56). The function is recursively invoked to identify the convertibility of all the next states in the context of FS_AX, new signal set, history and counter valuations. The result for a next state is added as child to the current node only when it is not a false node. Furthermore, the corresponding label the edge from current node to the newly added child node denotes the corresponding transition label of a candidate converter. If no child is added then there exists no converter for s as the AX formulas are unsatisfiable. In that case, a false node is returned.

Observe that, the recursive process may not terminate as formulas are expanded at Lines 46 and 48. To ensure termination, we rely on fixed point characterization of ACTL. Specifically, semantics of AG$\varphi$ is the greatest fixed point of $Z = \varphi \ \wedge$ AX$Z$ as AG properties are infinite path properties. As a result, proof of whether a state satisfies an AG property can depend on itself. On the other hand, the semantics of A($\varphi$ U $\psi$) is the least fixed point of $Z = \psi \ \vee \ (\varphi \ \wedge$ AX$Z)$ as A($\varphi$ U $\psi$) properties are finite path properties where $\psi$ must be satisfied after finite number of steps. From the above observations, if a state s with a formula set FS is revisited in the recursive call (Line 5), it is checked whether there exists any least fixed point obligation. If yes, this recursive path is not a witness to conformance of obligation in FS by s and false node is returned; otherwise the current node is returned (Line 6–10) resulting in a loop.

Another important aspect that is worth mentioning is the presence of counter valuations. Even if the counters can potentially take infinite number of valuations (which will also lead to non-termination of the recursive procedure), the valuations of the counters that are of interest are always partitioned finitely. For example for data-width requirements of the form AG($i \leq$ counter $\leq j$), any counter valuations less than $i$ or greater than $j$ results in violation of the property when theisConv terminates and returns false-node. As such, there are exactly $j - i + 3$ partitions of counter valuations; $j - i + 1$ partitions each recording desired valuations between $i$ and $j$ and the rest recording undesirable valuations.

**Algorithm 1** NODE isConv(s, I, FS, H, E)

```
1: if FS = ∅ then
2:   return TRUE_NODE
3: end if
4: curr = createNode(s, I, FS);
5: if curr ∈ H then
6:   if FS contains an AU formulas then
7:     return FALSE_NODE
8:   else
9:     return curr
10:  end if
11: end if
12: H_1 = H ∪ {curr}; FS_1 := FS − F
13: if F = TRUE then
14:   return isConv(s, I, FS_1, H_1, E)
15: else if F = FALSE then
16:   return FALSE_NODE
17: else if F = p (p ∈ AP or p is a counter-constraint) then
18:   if F is not satisfied in s then
19:     return FALSE_NODE
20:   else
21:     return isConv(s, I, FS_1, H_1, E)
22:   end if
23: else if F = ¬p (p ∈ AP or p is a counter-constraint) then
24:   if F is satisfied in s then
25:     return FALSE_NODE
26:   else
27:     return isConv(s, I, FS_1, H_1, E)
28:   end if
29: else if F = φ ∧ ψ then
30:   return isConv(s, I, FS_1 ∪ {φ, ψ}, H_1, E)
31: else if F = φ ∨ ψ then
32:   curr.type := OR_NODE
33:   child_1 := isConv(s, I, FS_1 ∪ {φ}, H_1, E)
34:   child_2 := isConv(s, I, FS_1 ∪ {ψ}, H_1, E)
35:   if child_1 ≠ FALSE_NODE then
36:     curr.addChild(child_1)
37:   end if
38:   if child_2 ≠ FALSE_NODE then
39:     curr.addChild(child_2)
40:   end if
41:   if child_1 = child_2 = FALSE_NODE then
42:     return FALSE_NODE
43:   else
44:     return curr
45:   end if
46: else if F = AGφ then
47:   return isConv(s, I, FS_1 ∪ {φ ∧ AXAGφ},H_1, E)
48: else if F = A(φ U ψ) then
49:   return isConv(s, I, FS_1∪{ψ ∨ (φ ∧ AXA(φ U ψ))},H_1, E)
50: end if
51: curr.type := AX_NODE
52: FS_AX = {φ | AXφ ∈ FS}
53: for each s --(i1,i2/o1,o2)-->_t s' do
54:   if foreach j = 1, 2 : i_j ∈ Σ_O1 ∪ Σ_O2 ⇒ i_j ∈ E then
55:     E_1 := E − {i_j} ∪_nodup {o1, o2}
56:     I' := update(I, s');
57:     if (N:=isConv(s', I', FS_AX, H_1, E_1)) ≠ FALSE_NODE then
58:       curr.addEdge(o1, o2/i1, i2); curr.addChild(N);
59:     end if
60:   end if
61: end for
62: if !curr.hasChild() then
63:   return FALSE_NODE
64: else
65:   return curr
66: end if
```

**Complexity.** The complexity of the algorithm can be obtained from the number of recursive calls. It is linear to the number of states in the composite protocols times the total number of counter partitions and exponential to the size of the ACTL formulas. The exponential factor comes in specifically because all the conjuncts in the conjunctive formula expression are aggregated (Line 30) and considered in converter existence. This is different from on-the-fly model checking [BCG95] where each conjunct are considered separately resulting in a complexity linear to the size of the formula. It is worth mentioning that in the current problem, unlike model checking, it is required to identify a converter which must behave identically for ensuring protocol conformance to each conjuncts and as such the conjuncts are aggregated.

The theorem follows from the above discussion.

**Theorem 1 (Sound and Complete)** *Given a protocol composition $P_1||P_2 = \langle AP_{1||2}, S_{1||2}, s_{0_{1||2}}, \Sigma_{I1||2}, \Sigma_{O1||2}, R_{1||2}, L_{1||2}, clk\rangle$ and properties represented as a set of formulas in FS, a control-consistent converter exists if and only if $isConv(s_{0_{1||2}}, FS, \emptyset, \emptyset)$ returns a non false node.*

To extract a control-consistent converter, the graph generated by the `isConv` function is traversed and the procedure is presented in Algorithm 2. The exploration is performed in a depth-first fashion and returns the converter state corresponding to the node `N` of the graph (argument of `extract`). Lines 1–3 state that if `N` is a false node, then the corresponding converter state is also a false state, denoting absence of converter. If the node `N` is revisited in the DFS exploration, then the converter state is one that is already associated with `N` during the first visit (Lines 4–6). If `N` is generated due to the presence of disjunctive obligation (Lines 31–45 in Algorithm. 1) a candidate controller can be extracted from any of the children of `N` (Lines 7–11). Finally, if `N` is an `AX_NODE` then a new converter state is generated and associated with `N` and for each edge-child of `N`, the corresponding transition-next states of the converter state are extracted.

**Converter for Reader-Writer Example.** For the reader-writer protocol pair example in Figure. 2 and the control and data requirements as described in Sections 2.3 and 2.4 the result from `isConv` is illustrated in Figure. 5. Note that, the extracted converter will have a state corresponding to each node in the graph (there is no `OR_NODE` in the graph). We will use the converter state and node identifiers of the graph interchangeably. The transitions in the extracted converter can be explained as follows: The converter state `Node 1` moves to `Node 2` via `req,ack/next,req`. Essentially, this means that the converter provides `next` signal to the reader protocol which produces `req` signal that is relayed by the converter to writer which in turn outputs `ack`. These operations happen in on clock tick. On the other hand, the converter moves from `Node 2` `Node 3` without consuming any output from the protocols; it just provides appropriate triggers to the protocols–`ack` signal to the reader and ∼`reset` (i.e. does not provide `reset`) signal to the write. The other transitions can be read likewise.

The converter not only allows the composite protocol behavior that conforms to control and data-mismatch requirements defined as ACTL formulas but it also resolves the incompatibilities between the protocols:

**Algorithm 2** CSTATE extract(N)

```
1:  if N = FALSE_NODE then
2:     return FALSE_STATE
3:  end if
4:  if N.visited = true then
5:     return getAssocState(N)
6:  end if
7:  if N.type = OR_NODE then
8:     N.visited := true
9:     pick any one child using N.getChild(i)
10:    return extract(N.getChild(i)
11: end if
12: if N.type = AX_NODE then
13:    N.visited := true
14:    CSTATE cstate := new CSTATE
15:    associateState(cstate, N);
16:    for each child of N do
17:       cstate.addTrans(N.getEdge(i))
18:       cstate.addNextState(extract(N.getChild(i)))
19:    end for
20:    return cstate
21: end if
```

1. *Synthesis of missing input signals*: The converter artificially generates inputs like `more` and `next` for the reader protocol and the `reset` signal for the writer as they are not emitted by any protocols.
2. *Buffering of the acknowledge signal*: From the initial state $(s_0, t_0)$ of $P_1||P_2$, once the transition to state $(s_1, t_1)$ is allowed, the writer generates the acknowledge signal immediately. The converter buffers this signal and presents it to the reader in the next clock-tick (represented by the enabled transitions from state $(s_1, t_2)$ to states $(s_2, t_2)$ and $(s_2, t_0)$ in $P_1||P_2$).
3. *Handling data-width mismatch*: The converted system satisfies the $\varphi_d$ property, which requires the counter associated with labels $\mathtt{DIn}_8$ and $\mathtt{DOut}_{16}$ to remain within the range 0 to 2. The converter synthesized guarantees the satisfaction of this specification and it can be seen that no reachable state exceeds the bounds on the counter.
4. *Preventing error states*: The converter synthesizes the `reset` signal for the writer to prevent it from entering the error state $t_3$.

# 4 Results

Our technique has been implemented by extending the NuSMV model checker [CCK$^+$06]. Table. 1 shows the results obtained from classical protocol conversion examples presented in existing literature on protocol conversion [RM91, PdAHSV02, CL90]. The first three columns describe the serial number and name of the protocols used, and the state size of their composition $(P_1||P_2)$. The fourth column informally describes the temporal logic specifications used for the converter synthesis. The final two columns present the result (success/failure in synthesizing a converter) and the size of the converter synthesized. Problems 1,2,3 and 4 are classical control mismatch problems with one-way communication between protocols. The handshake-serial problem for example requires that protocols do not read signals before they are emitted $(\varphi_{1_1})$ and that no two identical outputs appear

| No. | Name | No. of states in $P_1\|\|P_2$ | Properties | Result | No. of states in $\mathcal{C}$ |
|---|---|---|---|---|---|
| **1** | Handshake-serial [PdAHSV02] | 4 | ⇒No read before corr. write ($\varphi_{1_1}$).<br>⇒Outputs $a$ and $b$ alternate along every path ($\varphi_{1_2}$). | Success | 3 |
| 1.1 | Handshake-serial<br>(2-way communication) | 12 | ⇒$\varphi_{1_1}$,$\varphi_{1_2}$ | Success | 8 |
| 1.2 | Handshake-serial<br>(data-mismatch) | 12 | ⇒$\varphi_{1_1}$,$\varphi_{1_2}$<br>⇒Data is eventually consumed before more is written. | Success | 8 |
| **2** | ABP sender (8-bit data)-<br>NS receiver (8-bit data)[CL90] | 18 | ⇒Each output is eventually read ($\varphi_{2_1}$).<br>⇒Another output allowed only after an input ($\varphi_{2_2}$). | Success | 8 |
| 2.1 | ABP sender (16-bit data)-<br>NS receiver (8-bit data) | 18 | ⇒$\varphi_{2_1}$, $\varphi_{2_2}$ | Success | 12 |
| 2.2 | ABP sender (8-bit data)-<br>NS receiver (16-bit data) | 18 | ⇒$\varphi_{2_1}$, $\varphi_{2_2}$ | Success | 10 |
| **3** | ABP receiver (8-bit data)-<br>NS sender (8-bit data) [CL90] | 24 | ⇒Each output is eventually read ($\varphi_{3_1}$).<br>⇒Another output allowed only after an input ($\varphi_{3_2}$). | Success | 8 |
| 3.1 | ABP receiver (16-bit data)-<br>NS sender (8-bit data) | 24 | ⇒$\varphi_{3_1}$, $\varphi_{3_2}$ | Success | 10 |
| 3.3 | ABP receiver (8-bit data)-<br>NS sender (16-bit data) | 24 | ⇒$\varphi_{3_1}$, $\varphi_{3_2}$ | Success | 12 |
| **4** | Poll-End receiver (8-bit data)-<br>Ack-Nack sender (8-bit data)[RM91] | 6 | ⇒No overflow or underflow during<br>data communication ($\varphi_{4_1}$). | Success | 6 |
| 4.1 | Poll-End receiver (8-bit data)-<br>Ack-Nack sender(16-bit data) | 9 | ⇒ $\varphi_{4_1}$ | Success | 7 |

Table 1: Results from classical protocol conversion examples

consecutively along any path ($\varphi_{1_2}$). Problems 2 and 3 present the mismatch between alternating-bit and non-sequenced protocols where ABP protocols attach 1-bit tags with each output while the latter do not have any such sequencing. The aim of conversion in these cases was to ensure each output in one protocol is eventually read by the other ($\varphi_{2_1}, \varphi_{3_1}$) and that subsequent outputs happen only after previous outputs have been read ($\varphi_{2_2}, \varphi_{3_2}$). Problem 4 is a similar example which presents the mismatch between a Poll-end receiver and an Ack-Nack sender and conversion requires that outputs generated by the sender are always read successfully by the receiver. We found that for these classical examples, our approach generates converters similar in size and functionality to the ones described in the source literature from where the examples were obtained. However, when these examples are varied slightly to include bidirectional communication and/or data-width mismatches (1.1, 1.2, 2.1, 2.2, 3.1, 3.2 and 4.1), only our technique is able to generate converters.

Table. 2 shows the results obtained from synthetic benchmarks. Some benchmarks, namely Mutex, MCP-missionaries and 4-bit ABP sender-receiver, were obtained by modifying (introducing mismatches) the existing protocols in the NuSMV collection [CCK$^+$06], while the others are modeled on typical SoC protocols. Each example is selected to highlight the unique features of the protocol conversion approach presented in this paper. For example, while the Mutex problem (no. 2) can be handled by other conversion techniques like [PdAHSV02, Lam88], they involve writing a complex automaton to describe the mutual exclusion property. On the other hand, our approach creates the same converter using a much simpler mutual exclusion property (which requires that along all states, both processes never enter their critical sections simultaneously). Data mismatches and bidirectional communication presented in master-slave, reader-writer, MCP and ABP problems problems are handled successfully by our approach whereas other approaches failed.

The above benchmarking results showcase the capabilities of the presented conversion ap-

| No. | Name | No. of states in $P_1 \| \| P_2$ | Properties | Result | No. of states in $\mathcal{C}$ |
|---|---|---|---|---|---|
| 1 | Master-slave | 9 | ⇒No input before corr. output.<br>⇒Each output is read before another output. | Success | 6 |
| 2 | Mutex | 16 | ⇒Mutual exclusion always achieved | Success | 7 |
| 3 | Reader-writer | 12 | ⇒Writer never enters error state.<br>⇒No data written before request is made.<br>⇒All written data is read before transaction completes.<br>⇒Both protocols reset after each transaction.<br>⇒No read before data is written.<br>⇒Data-communication stays within bounds. | Success | 5 |
| 4 | MCP missionaries-cannibals | 30 | ⇒All missionary-cannibal pairs are transported without loss. | Success | 22 |
| 5 | 4-bit ABP sender-modified receiver | 166432 | ⇒Sender can always eventually read data. | Success | 14312 |

Table 2: Results from synthetic benchmarks

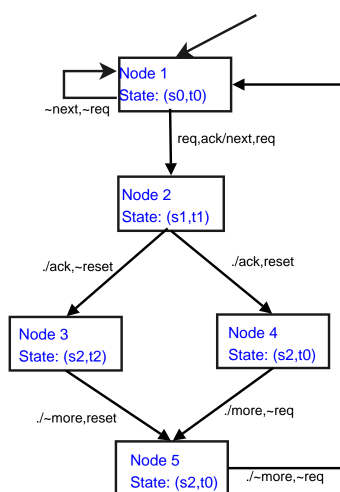


Figure 5: $\texttt{isConv}(s_{0_{1||2}}, \{\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5, \varphi_d\}, \emptyset, \emptyset)$

proach. While common control-mismatch problems can be tackled in a similar manner to other techniques, more complex specifications can be described in a succinct fashion and we allow conversion for protocols with data-width mismatches and/or bidirectional communication. More details on all above experiments appear in http://www.cs.iastate.edu/∼sbasu/research/emsoft07res.pdf

## 5 Conclusions

Protocol conversion to resolve protocol mismatches is an active research area and a number of solutions have been proposed. Some approaches require significant user effort, while some only partly address the protocol conversion problem. Most formal approaches work on protocols that have unidirectional communication and use finite state machines to describe specifications. In this paper we

propose a uniform formal approach to protocol conversion which can precisely represent protocols' input/output behavior and can handle bi-directional communication. The converters synthesized by our approach are capable of relaying and buffering of signals and address several protocol mismatch problems. We also presented comprehensive experimental results to show the practical applicability of our approach.

Some of the future avenues include investigating more powerful logic for property specification and handling of clock mismatches. We claim that the same algorithm can be extended with little effort to handle extensions to the logic while clock mismatches can be represented and addressed by using multi-clock Kripke structure.

# References

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[Ant95] M. Antoniotti. *Synthesis and verification of discrete controllers for robotics and manufacturing devices with temporal logic and the Control-D system*. PhD thesis, New York University, New York, 1995.

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64– 83, 2003.

[BCG95] Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for CTL*. In *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 388–397, June 1995.

[BI89] F M Burg and N D Iorio. Networking of networks: Interworking according to osi. *IEEE Journal on Selected Areas in Communications*, 7(7):1131–1142, September 1989.

[BK87] G Borriello and R H Katz. Synthesis and optimization of interface transducer logic. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 274–277, 1987.

[Boc90] G V Bochmann. Deriving protocol converters for communication gateways. *IEEE Transactions on Communications*, 38(9):1298–1300, September 1990.

[CCK+06] Roberto Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistore, and Marco Roveri. *NuSMV 2.1 User Manual*, 2006.

[CL90] Kenneth L Calvert and Simon S Lam. Formal methods for protocol conversion. *IEEE Journal on Selected Areas in Communication*, 8(1):127–142, 1990.

[DRS04] Vijay D'Silva, S Ramesh, and Arcot Sowmya. Synchronous protocol automata : A framework for modelling and verification of soc communication architectures. In *DATE*, pages 390–395, 2004.

[dRS05] 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*, 152(1):20–27, 2005.

[GBB+06] S. Gorai, S. Biswas, L. Bhatia, P. Tiwari, and R. S. Mishra. Directed-simulation assisted formal verification of serial protocol and bridge. In *Proceedings of the 43rd annual conference on Design automation DAC '06*, pages 731 – 736, 2006.

[Gre86] P. Green. Protocol conversion. *IEEE Transactions on Communications*, 34(3):257–268, March 1986.

[JK06] S. Jiang and R. Kumar. Supervisory control of discrete event systems with ctl* temporal logic specifications. *SIAM Journal on Control and Optimization*, 44(6):2079–2103, 2006.

[KG95] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems*. KLUWER, 1995.

[KVW01] O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164:322–344, 2001.

[Lam88] S Lam. Protocol conversion. *IEEE Transactions on Software Engineering*, 14(3):353–362, 1988.

[Lef05] J Lefebvre. *Esterel v7 Reference Manual-Initial Standardization Proposal*, 2005.

[NG95] S. Narayan and D. Gajski. Interfacing incompatible protocols using interface process generation. In *Design Automation Conference*, pages 468–473, 1995.

[PdAHSV02] 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 *International Conference on Computer Aided Design ICCAD*, 2002.

[RM91] Murali Rajagopal and Raymond E. Miller. Synthesizing a protocol converter from executable protocol traces. *IEEE Transactions on Computers*, 40(4):487–499, 1991.

[RMK03] Abhik Roychoudhury, Tulika Mitra, and S. R. Karri. Using formal techniques to debug the amba system-on-chip bus protocol. In *Proceedings of the conference on Design, Automation and Test in Europe DATE '03*, volume 1, 2003.

[RW89] Peter J. G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[SL89] J. C. Shu and Ming T. Liu. A synchronization model for protocol conversion. *Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies. Technology: Emerging or Converging? INFOCOM '89*, pages 276–284, 1989.