A Model Checking Approach to Protocol Conversion

Technical Report
No.0000482

Roopak Sinha, Partha Roop
Department of Electrical and Computer Engineering
University of Auckland
and
Samik Basu
Department of Computer Science
Iowa State University

Iowa State University
Department of Computer Science

November, 2006

Abstract

Protocol conversion for mismatched protocols has been addressed in a number of formal and informal settings. However, existing solutions address this problem only partially. This paper develops the first on-the-fly local approach to protocol conversion based on temporal logic model checking. The tableau-based approach verifies the existence of a converter, and if a converter exists, it is automatically synthesized. Our approach handles control and data mismatches under a single unifying framework. A NuSMV-based implementation has been developed and we provide results for some non-trivial protocol mismatch examples.

1 Introduction

A system-on-a-chip (SoC) is built by reusing components connected using a central bus such as AMBA [3]. A major problem with this reuse is the inherent *mismatch* between protocols of components, an active area of research for about two decades [7]. Mismatches occur because components are developed independently without any intention of eventual integration, and can result from control signal mismatches [6], inconsistent naming conventions [11], different clock speeds and difference in data-widths [3]. Mismatches are corrected, if possible, by synthesizing extra glue-logic, called a *converter*[10] to control communication between given protocols in order to satisfy given specifications.

A number of techniques address the problem of protocol conversion in a wide range of formal and informal settings with varying degrees of automation—projection approach [7], conversion seeds [9] and synchronization [11]. Some approaches, like conversion seeds [9] and protocol projections [7], require significant user expertise and guidance. While this problem has been studied in a number of formal settings [6, 7, 9, 11], only recently have some formal verification based solutions been proposed [3, 5, 10].

In [5], a hybrid simulation/verification approach to protocol conversion in SoC designs is proposed. [10] proposes an approach towards protocol conversion employing a game-theoretic framework to generate a converter. This solution is restricted only to protocols with half-duplex communication between them, with specifications represented as finite state machines. D'Silva et al [3] present synchronous protocol automata to allow formal protocol specification and matching, as well as converter synthesis. The technique addresses the problem of limited communication medium capacity, but data-communication between protocols cannot be constrained any further.

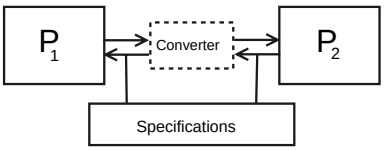

Figure 1: Protocol conversion

In contrast to the above techniques, we present a technique using model checking for automatic converter synthesis. Protocols, in our setting, are represented using *Kripke Structures* (KS) and specifications (and their negations) are expressed in the temporal logic ACTL. ACTL, a branching time temporal logic with universal path quantifiers, is particularly relevant for protocol conversion as mismatches in protocols must be addressed for *every* path of their KS descriptions. Given two KSs P_1 and P_2 and a set Ψ of desired ACTL properties, the protocol conversion via converter synthesis problem (illustrated in Fig. 1) may be stated as:

Can a converter $\mathcal{C}$ be synthesized for P_1 and P_2 such that all formulas in Ψ can be satisfied?

The proposed approach involves the local and on-the-fly construction of a tableau [2] where satisfaction of ACTL formulas in Ψ is defined in terms of the satisfaction of their subformulas by the states of the protocols. The tableau construction results in the automatic synthesis of the converter, if one exists. If no converter is found, failures can be identified without exploring states irrelevant for failure inference. Not only are temporal logic specifications succinct and more intuitive to write, additional constraints such as fairness can also be specified. Fairness constraints ensure that converters allow meaningful communication to take place between protocols. We use invariants to specify bounds on data-widths [1] so that data-width mismatches are addressed.

The main contributions of this paper are:

- We propose the first model checking based solution to protocol matching guaranteed to produce a converter if one exists—no further proof of correctness is required, unlike some other approaches [3].
- We present a novel tableau-based algorithm to address data and control mismatches in a unifying manner, unlike earlier solutions that deal with them separately or in an ad hoc manner.
- We present an on-the-fly algorithm for converter synthesis, one where protocol states are explored only when needed. The algorithm is polynomial in the size of the protocols and specifications.

The rest of this paper is organized as follows. Section 2 presents the automatic converter synthesis approach based on tableau generation. Section 3 provides some implementation results with concluding remarks in section 4.

2 Methodology

The proposed protocol conversion algorithm takes as input the KS descriptions of two protocols and a set of ACTL properties. It then employs a local, on-the-fly tableau construction algorithm to verify the existence of a converter.

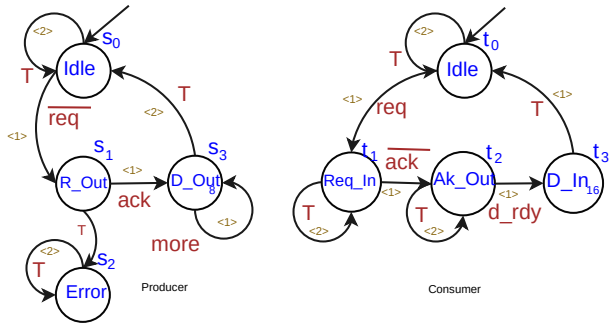

Figure 2: Producer-consumer protocol pair

2.1 Input description

The conversion algorithm takes as input two protocols represented as Kripke structures:

Definition 1: A Kripke structure (KS) is a *finite state machine* represented as a sextuple $\langle AP, S, s_0, \Sigma, R, L, \rangle$ where AP is a set of atomic propositions; S is a finite set of states; $s_0 \in S$ is the initial state; Σ is a finite set of input and output events; $R \subseteq S \times \Sigma \times S$ is the transition relation; and $L : S \rightarrow 2^{AP}$ is the state labelling function.

States in a protocol are labelled by unique identifiers. Transitions between states, each labelled with a priority, trigger with respect to a clock. At each clock cycle, the KS checks for the presence of input/output events that can trigger a transition from the current state. If multiple input/output triggers are present, the transition using the highest priority is taken. An event a represents an input whereas $\overline{a}$ represents an output. The relations $(s, a, s') \in R$ are represented as $s \xrightarrow{a} s'$.

Fig. 2 shows the protocols of two devices, a producer and a consumer represented as Kripke structures. The producer protocol P_1, sends sends a request before producing any data and in the next cycle awaits the input signal *ack*. If *ack* is absent, the protocol enters an error state (s_2). Else, it goes on to write 8-bits on to the data channel D_out_8 and is capable of writing multiple 8-bit data if the signal *more* is present in subsequent cycles. The 16-bit consumer, P_2, reads the request from the producer and then emits an acknowledgement (*ack*). It then waits for the input *d_rdy* to read *one* 16-bit data from the data channel.

The protocols have a number of incompatibilities. Although P_1 and P_2 can share the input/output channels *req* and *ack*, but there are no outputs corresponding to the control inputs *d_rdy* and *more* (*control incompatibility*). Furthermore, there is also a *data incompatibility* as P_1 writes 8-bit data whereas P_2 can only read 16-bit data.

In addition to addressing the above issues, the intended communication between the producer-consumer protocols can be further described using addi-

tional ACTL formulas. ACTL is a fragment of the branching time temporal logic CTL and only allows universal path quantifiers. Semantics of an ACTL formula, φ denoted by $[\![\varphi]\!]_M$ are given in terms of set of states in a KS, M, which satisfies the formula. A state $s \in S$ is said to satisfy a ACTL formula φ, denoted by $s \models \varphi$, if $s \in [\![\varphi]\!]_M$. We also say that $M \models \varphi$ to indicate that the initial state s_0 of the model M satisfies φ. We restrict ourselves to formulas where negations are applied to propositions only. For the producer-consumer example in Fig. 2, the following properties are needed:

1. $\mathtt{A}(\neg Req\_In\ \mathtt{U}\ R\_Out)$ ($S1$): A request cannot be read before one is made.
2. $\mathtt{A}(\neg D\_Out_8\ \mathtt{U}\ Req\_In)$ ($S2$): No data is written data before a request has been received.
3. $\mathtt{AG}(\neg Error)$ ($S3$): The communication never enters a state labelled by $Error$.

2.1.1 Describing data-width mismatches

The producer-consumer protocol pair has a data-width mismatch as P_1 writes 8-bit numbers to the data channel whereas P_2 reads 16-bit numbers. We formally describe the desired data-communication behaviour as follows. Given the data-widths N and M of the outputs and inputs respectively, we first compute the minimum width needed for the communication medium 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 $N \times f > M$; otherwise the minimum capacity is N. This 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 $\mathtt{LCM}(N, M)$. 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, *decremented* by x for every input, and verify that the counter always remains between 0 and $x \times y$ using the *invariant* $0 \leq counter \leq (x \times y)$.

In addition, we can also force that maximum number of outputs are done before inputs start and vice versa—expressed by the following ACTL formulas:

$$\mathtt{AG}(counter = 0 \ \Rightarrow\ \mathtt{A}(\neg \mathtt{input}\ \mathtt{U}\ counter = x \times y)\ )$$
$$\mathtt{AG}(counter = x \times y \ \Rightarrow\ \mathtt{A}(\neg \mathtt{output}\ \mathtt{U}\ counter = 0)\ )$$

For the producer-consumer example in Fig. 2, we use the invariant $0 \leq counter \leq 2$ as N=1 and M=2.

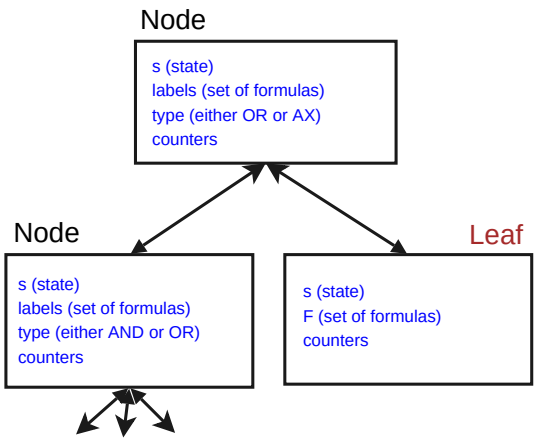

Figure 3: The structure of nodes and leaves

2.1.2 Fairness

In order to ensure that converters always allow meaningful communication between protocols, it may be desirable that additional fairness conditions are satisfied, which can be again defined using ACTL formulas. For converter synthesis, the goal is to ignore as well as *disable* all unfair behaviours of the protocols. For the producer-consumer example, we use the following fairness conditions:

- $\texttt{AGAF}(D\_Out_8)$ $(F1)$: The producer can always eventually write data.
- $\texttt{AG}(D\_Out_8 \Rightarrow \texttt{AXA}(\neg Req\_Out\ \texttt{U}\ Data\_In_{16})$ $(F2)$: Once some data is written, no further requests are allowed before a read operation is performed.
- $\texttt{AGAF}(R\_Out)$ $(F3)$: The producer can always eventually emit requests.

2.2 Tableau Construction Algorithm

The proposed technique is based on model checking and involves on-the-fly tableau construction, similar to [2]. Given the inputs P_1 and P_2 describing participating protocols, some invariants over data counters, and the set Ψ of desired properties (including fairness properties), the tableau construction algorithm proceeds as follows. We illustrate the working of the tableau construction algorithm using the producer-consumer example given in Fig. 2.

The first step is the computation of the unrestricted combined behaviour $P_1||P_2$ of P_1 and P_2, also called the parallel composition (similar to the synchronous parallel in Argos [8]), defined as follows.

Definition 2: Parallel Composition.

Given two Kripke structures $P_1 = \langle AP_1,\ S_1,\ s_{0_1},\ \Sigma_1,\ R_1,\ L_1\rangle$ and $P_2 = \langle AP_2,\ S_2,\ s_{0_2},\ \Sigma_2,\ R_2,\ L_2,\rangle$, their parallel composition, denoted by $P_1||P_2$ is

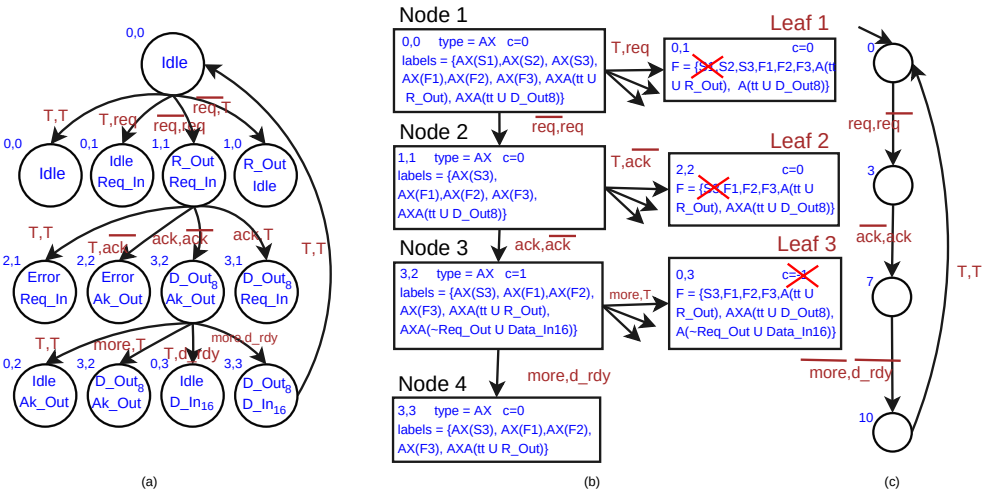

Figure 4: Tableau construction procedure. (a) states of $P_1||P_2$, (b) tableau nodes, (c) synthesized converter.

$\langle AP_{1||2}, S_{1||2}, s_{0_{1||2}}, \Sigma_{1||2}, R_{1||2}, L_{1||2}\rangle$ 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 $\Sigma_{1||2} \subseteq \Sigma_1 \times \Sigma_2$. $R_{1||2} \subseteq S_{1||2} \times \Sigma_{1||2} \times S_{1||2}$ such that

$$(s_1 \xrightarrow{\sigma_1} s_1') \wedge (s_2 \xrightarrow{\sigma_2} s_2') \Rightarrow ((s_1, s_2) \xrightarrow{(\sigma_1,\sigma_2)} (s_1', s_2'))$$

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

Some states of the parallel composition of the producer-consumer example are shown in Fig. 4(a).

Algorithm 1 *check*(s_0, Ψ)

```
1: Initialize counters, create a set A of leaves.
2: call create_leaf(s, counter_vals, Ψ, Nil)
3: while A is non-empty do
4:    remove one leaf t from A
5:    result = solve_leaf(t)
6:    if result = SUCCESS or FAILURE then
7:       result = notify_parent(t, result)
8:       if globalresult = SUCCESS then
9:          return SUCCESS
10:      end if
11:   end if
12: end while
13: return FAILURE
```

The proposed algorithm constructs a tableau which has *nodes* and *leaves* (Fig. 3). Nodes are parents that have one or more children. Each node corresponds to a state in $P_1||P_2$ and a specific valuation of all data counters, and is labelled with a set of formulas *labels*. Leaves, on the other hand, do not

have children, and have no labels. A leaf contains a set of formulas F that its corresponding state must satisfy. The tableau can be extended only when a leaf is expanded into a node.

Given the states of $P_1||P_2$ for the producer-consumer example (Fig. 4(a)), the algorithm proceeds as follows. The initial state (s_0, s_0) of $P_1||P_2$ and the set Ψ ($\{S1, S2, S3, F1, F2, F3\}$) are passed to the top-level tableau construction procedure *check* which initializes data counters and creates a set A, which contains all leaves created during tableau construction. *check* then contains the initial top-level tableau leaf corresponding to (s_0, s_0) and the set of properties Ψ. This is the *root* of the tableau which contains the initial state of $P_1||P_2$ and the original set of properties Ψ. The method $create\_leaf$ is used to create new leaves. If the creation of a leaf results in the violation of any invariants, the leaf is deemed invalid. After creating the root leaf, *check* calls the procedure $solve\_leaf$ for this newly created leaf.

Algorithm 2 $create\_leaf(s, prev_counter_vals, \Psi, parent)$

1: Update counters based on labels of s.
2: **return** if counters violate invariants.
3: **if** there exists a leaf t with $t.state = s$ and $t.counter\_status$ = current counter status **then**
4: $t.formulas = t.formulas \bigcup \Psi$
5: **else**
6: *create* a new leaf t with $t.state = s$
7: $t.formulas = \Psi$, $t.parent = parent$, $t.type = LEAF$.
8: $t.counter\_status$ = updated counter status.
9: **end if**

The $solve\_leaf$ method receives a tableau leaf t as input and proceeds to individually break down formulas contained in $t.F$ into current-state and future commitments (lines 4-18). This decomposition of formulas is based on a set of tableau rules (Fig. 5. For example, for root leaf of the producer-consumer example, the formula $S3$ (`AG`($\neg Error$)) is broken down into the subformulas $Error$ and `AXAG`($\neg Error$). The sub-formula $Error$, being a current-state commitment (a proposition), is checked against the leaf state (s_0, s_0). The future commitment `AXAG`($\neg Error$) is stored in the set $label$. Once all current-state commitments have been checked, the root leaf is expanded to become a node (Node 1 in Fig. 4(b)) and a leaf is created for each successor of the state (s_0, s_0). All future commitments (including `AXAG`($\neg Error$)) are added to the set $labels$ of the newly expanded node, and (after removing `AX` from the formulas) to the set F of each of the newly created leaves. Node 1 becomes an AX_NODE as only those successors of the state $t.state$ can be enabled that satisfy these future commitments. Another case when a node may be expanded is when an OR formula is encountered and the success of the parent node depends on the success of any of its children. When a leaf is expanded (as in the case of the root leaf for the producer-consumer example), the keyword $EXPANDED$ is

$$\text{emp}\ \ \frac{(s,\overrightarrow{c})//c \models \{\}}{\bullet} \qquad \text{prop}\ \ \frac{(s,\overrightarrow{c}//c \models [p \cup \Psi]}{s//c \models \Psi}\ p \in L(s)\ \vee\ \overrightarrow{c} \models p$$

$$\wedge\ \frac{(s,\overrightarrow{c})//c \models [\varphi_1 \wedge \varphi_2 \cup \Psi]}{(s,\overrightarrow{c})//c \models [\{\varphi_1,\varphi_2\} \cup \Psi]}$$

$$\vee_1\ \frac{(s,\overrightarrow{c})//c \models [\varphi_1 \vee \varphi_2 \cup \Psi]}{(s,\overrightarrow{c})//c \models [\{\varphi_1\} \cup \Psi]} \qquad \vee_2\ \frac{(s,\overrightarrow{c})//c \models [\varphi_1 \vee \varphi_2 \cup \Psi]}{(s,\overrightarrow{c})//c \models [\{\varphi_2\} \cup \Psi]}$$

$$\mathtt{unr}_{au}\ \frac{(s,\overrightarrow{c})//c \models [\mathtt{A}(\varphi\ \mathtt{U}\ \psi) \cup \Psi]}{(s,\overrightarrow{c})//c \models [(\psi \vee (\varphi \wedge \mathtt{AXA}(\varphi\ \mathtt{U}\ \psi))) \cup \Psi]}$$

$$\mathtt{unr}_{ag}\ \frac{(s,\overrightarrow{c})//c \models [\mathtt{AG}\varphi \cup \Psi]}{(s,\overrightarrow{c})//c \models [(\varphi \wedge \mathtt{AXAG}\varphi) \cup \Psi]}$$

$$\mathtt{unr}_{s}\ \frac{(s,\overrightarrow{c})//c \models \Psi}{\exists \pi \subseteq \Pi.\ (\forall \sigma \in \pi.\ (s_\sigma,\overrightarrow{c_\sigma})//c_\sigma \models \Psi_{AX})} \begin{cases} \Psi_{\mathtt{AX}} = \{\varphi_k \mid \mathtt{AX}\varphi_k\ \in\ \Psi\} \\ \Pi = \{\sigma \mid (s,\overrightarrow{c}) \xrightarrow{\sigma} (s_\sigma,\overrightarrow{c_\sigma})\} \\ c_\sigma = c' : c \xrightarrow{\sigma'} c'\ \wedge\ \mathcal{D}(\sigma,\sigma') \end{cases}$$

Figure 5: Tableau Rules for converter generation

returned by *solve_leaf*.

For the producer-consumer example, root node 1 (Fig. 4(b)) has 4 children. *check* iteratively calls *solve_leaf* on all newly created leaves. When *solve_leaf* operates on leaf 1 (a child of node 1), it returns a $FAILURE$. This happens because the violation of the property $S1$. $S1$ is broken down to the current-state commitment $R\_Out \vee (\neg R\_In \wedge \mathtt{AXA}(\neg R\_In\ \mathtt{U}\ R\_Out)$ which is not satisfied by the state (s_0, s_1) (the state corresponding to leaf 1). The *check* procedure then passes the returned value to the method *notify_parent*, which effectively results in the transition from (s_0, s_0) to (s_0, s_1) to be disabled in the tableau. Another child leaf of node 1 corresponding to (s_1, s_1) satisfies all its current-state commitments and hence is further expanded into node 2 with 4 children. Its child leaf 2, corresponding to the state (s_2, s_2) in $P_1||P_2$ returns a $FAILURE$ as it is labelled by $Error$ (violates $S3$).

The child leaf corresponding to the state (s_3, s_2) is expanded (node 3) with only 3 children even though the state (s_3, s_2) has 4 successors. This is so because the child leaf 3 corresponding to the state (s_0, s_3) is invalid as the data read (D_In_{16}) in this state results in the counter c taking a value (-1) which is outside the allowed invariant range ($0 \leq c \leq 2$).

One of the children of node 3, corresponding to the state (s_3, s_3) is further expanded into node 4 with 2 child leaves (one for each successor (s_3, s_0) and (s_0, s_0)). The leaf corresponding to (s_3, s_0) returns $FAILURE$ (counter out of bounds), whereas (s_0, s_0) returns $SUCCESS$. This happens because all formulas passed to this child leaf are contained in the labels of the ancestor node 1 (corresponding to the same state (s_0, s_0) and counter valuation ($c = 0$). Due to this, *solve_leaf* does not store any further future commitments (all of the type `AXAG` (lines 24-33) and as all current-state commitments are met, returns

Algorithm 3 *solve_leaf*(t)

1: s = t.state, $\Psi = t.F$
2: Initialize Ψ_{AX}
3: **while** Ψ is non-empty **do**
4: remove one formula f from Ψ
5: **if** $f =$ `TRUE` **then**
6: continue
7: **else if** $f =$ `FALSE` **then**
8: **return** `FAILURE`
9: **else if** $f = p \in AP$ **then**
10: return `FAILURE` if **not** $f \in L(s)$
11: **else if** $f = \neg p$ $(p \in AP)$ **then**
12: return `FAILURE` if $f \in L(s)$
13: **else if** $f =$ `AG`P **then**
14: insert $P \wedge$ `AXAG`P in Ψ
15: **else if** $f =$ `A`$(P$ `U` $Q)$ **then**
16: insert $Q \vee (P \wedge$ `AXA`$(P$ `U` $Q))$ in Ψ
17: **else if** $f = P \wedge Q$ **then**
18: insert P, Q in Ψ
19: **else if** $f = P \vee Q$ **then**
20: $t.type = OR\_NODE$
21: $create\_leaf(s, \Psi \bigcup \{P\} \bigcup \Psi_{AX}, t)$
22: $create\_leaf(s, \Psi \bigcup \{Q\} \bigcup \Psi_{AX}, t)$
23: **return** `EXPANDED`
24: **else if** $f =$ `AX`P **then**
25: **if** $P =$ `A`$(Q$ `U` $R)$ and P occurs earlier in tableau at s **then**
26: **return** `FAILURE`
27: **else**
28: add f to the set Ψ_{AX}
29: **end if**
30: **if** $P =$ `AG`Q and no ancestor of t contains P at state s **then**
31: add f to the set Ψ_AX
32: **end if**
33: **end if**
34: **end while**
35: **if** Ψ_{AX} is non-empty **then**
36: $\Psi'_{AX} = \{p | \texttt{AX}p \in \Psi_{AX}\}$, $t.type = AX\_NODE$
37: $t.labels = label$
38: **for** each successor s' of s **do**
39: $create\_trace(s', \Psi'_{AX}, t)$
40: **end for**
41: **return** `EXPANDED`
42: **end if**
43: **return** `SUCCESS`

$SUCCESS$. This results in the tableau being folded back with a reference to node 1. A similar leaf may return $FAILURE$ if the future commitments are of the type `AU`.

The return values $SUCCESS$ or $FAILURE$ returned by *solve_leaf* for a leaf t is passed to the $notify\_parent$ method which recursively passes the result to its parent (ancestors if needed). A node returns $SUCCESS$ if at least on of its children returns $SUCCESS$. An OR_NODE does not disable any transitions (as only one child is needed to satisfy commitments), whereas an AX_NODE enables transitions to only those children that satisfy the future commitments passed to them. The algorithm finishes when the root leaf (corresponding to (s_0, s_0)) returns $SUCCESS$.

Algorithm 4 $notify\_parent(t, result)$

```
1: if t is a top-level trace then
2:    globalresult = result
3:    return
4: end if
5: parent = t.parent
6: if parent.type = OR_NODE then
7:    if result = FAILURE then
8:       call notify_parent(parent, FAILURE) if no unchecked child remains.
9:    else
10:      call notify_parent(parent, SUCCESS)
11:   end if
12: else if parent.type = AX_NODE then
13:   if result = FAILURE then
14:      remove the link from parent to t.
15:      if child-set of parent is empty then
16:         call notify_parent(parent, FAILURE) if successor set of parent.state
            is empty
17:         else call notify_parent(parent, SUCCESS)
18:      end if
19:   end if
20: end if
```

2.3 Converter Synthesis

The converter is synthesized automatically during tableau construction if it completes successfully. We traverse the nodes (starting from the root node that returned $SUCCESS$) in the tableau and select the AX nodes encountered to form states in the converter. The converter states synthesized for the states of $P_1||P_2$ for the producer-consumer example shown in Fig. 4(a), are shown in Fig. 4(c). Note how each converter state corresponds to an AX node in the tableau (Fig. 4(b)). The full converter $\mathcal{C}$ is shown in Fig. 7.

The converter resolves incompatibilities between P_1 and P_2 as follows. The input (output) events in $P_1||P_2$ are output (input) events for the converter. The converter can read an output generated by either protocol and may pass it to the other. Using this underlying control, the controller can perform *inhibition*, *buffering* and *synthesis*. The converter *inhibits* an event when the transition resulting from an event is disabled in $\mathcal{C}$ itself. In this case, the converter disallows (or absorbs) the input instead of passing it on. The converter may *buffer* an event, where it holds an input for a while before passing it on. Finally, in case a control input expected by one protocol is not provided by the other, the converter *synthesizes* it as an output itself in order to avoid blocking states. The converter $\mathcal{C}$ (Fig. 7) for the producer-consumer example resolves the control incompatibility between the two protocols by synthesizing the signals *more* and *d_rdy* artificially when required. The data incompatibility is resolved as all paths leading to states where a mismatch happens (the counter variable in the tableau exceeds invariant range) are disabled using inhibition. Furthermore, buffering and inhibition are used such that only those paths that satisfy the given $ACTL$ specifications and fairness properties are also satisfied.

Formally, the control of a converter $\mathcal{C}$ over given protocols P_1 and P_2, called the lock-step composition $\mathcal{C}//(P_1||P_2)$, is defined as follows

Definition 3: Lock-step Converter Composition. Given the KS $P_{1||2} = \langle AP_{1||2}, S_{1||2}, s_{0_{1||2}}, \Sigma_{1||2}, R_{1||2}, L_{1||2}\rangle$ and a converter $\mathcal{C} = \langle AP_{\mathcal{C}}, S_{\mathcal{C}}, s_{\mathcal{C}0}, \Sigma_{\mathcal{C}}, R_{\mathcal{C}}, L_{\mathcal{C}}\rangle$, the lock-step composition $\mathcal{C}//(P_1||P_2) = \langle AP_{1||2}, S_{\mathcal{C}//(1||2)}, s_{0_{\mathcal{C}//1||2}}, \Sigma_{1||2}, R_{\mathcal{C}//(1||2)}, L_{\mathcal{C}//(1||2)}\rangle$ such that:

$S_{\mathcal{C}//(1||2)} \subseteq S_{\mathcal{C}} \times S_{1||2}$; and $s_{0_{\mathcal{C}//1||2}}(s_{0_{\mathcal{C}}}, s_{0_{(1||2)}})$. $R_{\mathcal{C}//(1||2)}$ is defined as:

$$s_{\mathcal{C}} \overset{\sigma_1^c,\sigma_2^c}{\rightarrow} s'_{\mathcal{C}} \wedge s_{1||2} \overset{(\sigma_1,\sigma_2)}{\rightarrow} s'_{1||2} \wedge \mathcal{D}(\sigma_1^c,\sigma_1) \wedge \mathcal{D}(\sigma_2^c,\sigma_2) \Rightarrow s_{\mathcal{C}//(1||2)} \overset{(\sigma_1,\sigma_2)}{\rightarrow} s'_{\mathcal{C}//(1||2)}$$

Finally, $L_{\mathcal{C}//(1||2)}(s_P, s_{\mathcal{C}}) = L_{1||2}(s_{1||2})$.

The transition relation of the protocols composed with a converter ensures that protocols move only when the converter allows that move. As such the lock-step composition // is different from unrestricted composition (Definition 2). For the producer-consumer example, the corresponding controlled system $\mathcal{C}//(P_1||P_2)$ is given in 6

The following theorem follows from the above:

Theorem 3: Sound and Complete. Two protocols P_1 and P_2, with n defined data counters $c_1, c_2, \ldots c_n$ with the constraints $min_i \leq c_i \leq max_i$, can be made compatible *wrt* to a set Ψ of `ACTL` formulas by using an automatically generated converter $\mathcal{C}$ *iff* *check* returns $SUCCESS$.

Proof.

The first step in proving the above theorem is to realize the size of the state space to be traversed during tableau construction.

Given P_1 with total number of states $|S_1|$, and P_2 with total number of states $|S_2|$, the input to the *check* method $(P_1||P_2)$ would contain a maximum of $|S_1| \times |S_2|$ states.

Observation 1. For each counter c_i $(0 \leq i \leq n)$, the values of interest lie in $K_i + 2$ partitions, where $K_i = max_i - min_i + 1$. K_i partitions are singleton

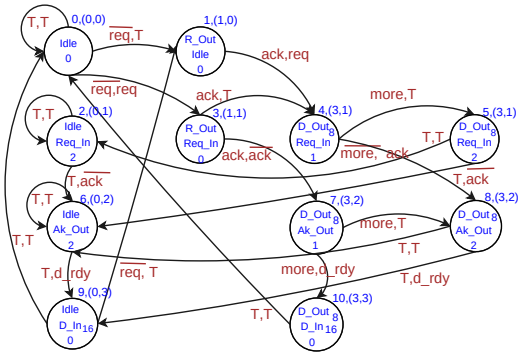

Figure 6: The resulting system $\mathcal{C}//(P_1||P_2)$

sets containing one element each from the range max_i to min_i, one partition containing all data points in $< min_i$ and another containing all data points $> max_i$. In other words, if the valuation of the counter goes beyond the limits min_i or max_i, it is not required to record the exact valuation. This stems from the fact that if c_i takes values outside the allowed range, the invariant $(min_i \leq c_i \leq max_i)$ is violated and leads to a failed path in the tableau.

It follows from the above observation that for each state s in $P_1||P_2$, there may be several (exactly K_i+2) valuations for each counter variable c_i, resulting in multiple states corresponding to the same state in $P_1||P_2$ (each with a different valuation of counters). Therefore the total number of states $|S|$ that can be traversed by the algorithm in the worst case is:

$$|S| = |S_1 \times S_2| \times (K_1 + 2) \times (K_2 + 2) \times \ldots \times (K_n + 2)$$

We use $\overrightarrow{c}$ to denote a valuation of all counters and use S to denote the expanded state space.

Having defined the maximum size of the state-space to be traversed, we first note that for any state $(s, \overrightarrow{c})$ in S, if $\overrightarrow{c}$ refers to a valuation of the counters where any one counter is outside its allowed range, the tableau leaf automatically results in a failed path (function $create\_leaf$). $solve\_leaf$ is only called for leaves which are associated states in S that have valid valuations for all counters.

The proof then proceeds by realizing the soundness and completeness of each of the tableau rules. For brevity, we present here the proof-sketch for $\texttt{unr}_s$, proofs for the other rules are straightforward.

Recall that, $(s, \overrightarrow{c})|c \models \Psi$ $((s, \overrightarrow{c}) \in S)$, where Ψ is the set of formula expressions with temporal operators AX, is satisfiable if the next states proof obligations are satisfied by destination states reachable via transitions enabled by the converter $\mathcal{C}$. The converter can enable any subset (barring $\emptyset$) of transitions. The tableau rule, therefore, considers all possible subsets of destination states of enabled transitions.

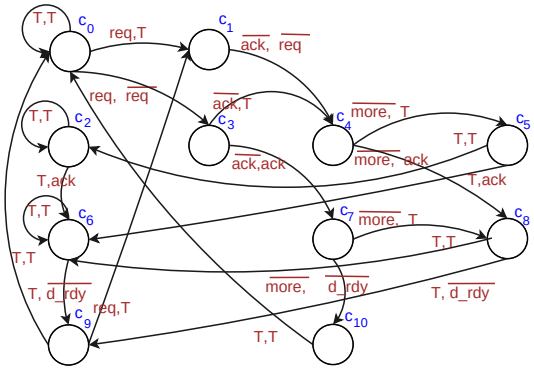

Figure 7: The converter $\mathcal{C}$

As each transition is annotated by an event σ, we construct Π, the set of events of out-going transitions. In other words, $\sigma \in \Pi \Rightarrow (s_\sigma, \overrightarrow{c}_\sigma)$ is reachable via the transition with event σ. We are required to identify one possible subset of Π which represents the enabled transitions whose destinations conform to the obligations in the consequent (see $\exists \pi \subseteq \Pi$ in the consequent). Let $\pi_s = \{(s_\sigma, \overrightarrow{c}_\sigma) \mid \sigma \in \pi\}$ be the next states reachable via (selected) enabled transitions.

The consequent of the tableau rule has the following obligations. All elements of π_s in parallel composition with the converter must satisfy the expressions in Ψ_{ax}. This ensures that the converter constructed is consistent, i.e., c is constructed such that $(s, \overrightarrow{c})|c$ satisfies all obligations (Ψ_{ax}). Therefore, if we can generate an environment for s corresponding to rule $\mathtt{unr}_s$, then $s \models \psi$ where ψ is the conjunction of the elements of the set $\{AX\varphi \mid \varphi \in \Psi_{ax}\}$. The other direction can be proved likewise.

3 Results

A protocol conversion tool employing the tableau construction approach has been implemented by extending the NuSMV model checker [4]. The results table (Table 1) contains four columns. The first two columns contain the description and size (number of states) of the participating protocols . The ACTL properties used are shown in the third column with the size of the converter shown in column 4. The first five problems are well-known protocol conversion problems with control mismatches [10, 7]. The next problem is the multi-write producer and single-read consumer protocol pair used as the motivating example in this paper. The size of the output of the producer was kept constant at 8-bits and the input-size of the consumer were varied in multiples of 8-bits and the size of the converter is noted in the fourth column. Note that when the input size was not a direct multiple of the output size (9-bits and 2-bits respectively), no converter could be generated. This was due to the fact that the consumer

$P_1(\|S_{P_1}\|)$	$P_1(\|S_{P_2}\|)$	ACTL Properties	$\mathcal{C}(\|S_C\|)$
Master (3)	Slave (3)	Event sequencing (one grant per request), (requests precede grants)	6
ABP sender(6)	NP receiver(4)[7]	Control Signal matching	8
ABP receiver(8)	NP sender(3)[7]	Control signal matching	8
Poll-End Receiver(2)	Ack-Nack Sender(3)	Data communication resolution (one data in per data out)	6
Handshake (2)	Serial(2)[10]	Control signal matching and event sequencing (Alternating A and B labels)	3
Multi-write Producer protocol(3)	Single-read Consumer protocol(4)	$\texttt{AG}(\neg Error)$,$\texttt{A}(\neg D\_Out\ \texttt{U}\ Req\_In)$ $\texttt{A}(\neg Req\_In\ \texttt{U}\ R\_Out)$	
8-bit Write	8-bit Read	(Error is never encountered,	8
8-bit Write	16-bit Read	no data written before requests,	11
8-bit Write	24-bit Read	and no requests read before	13
8-bit Write	32-bit Read	any requests are made)	15
2-bit Write	9-bit Read		Failed
Multi-write Producer protocol(3)	Multi-read Consumer protocol(4)	$\texttt{AG}(\neg Error)$,$\texttt{A}(\neg D\_Out\ \texttt{U}\ Req\_In)$ $\texttt{A}(\neg Req\_In\ \texttt{U}\ R\_Out)$	
2-bit Write	9-bit Read	(Error is never encountered,	25
2-bit Write	11-bit Read	no data written before requests,	29
3-bit Write	10-bit Read	and no requests read before	28
3-bit Write	11-bit Read	any requests are made)	30
9-bit Write	2-bit Read		11
7-bit Write	64-bit Read		140
11-bit Write	256-bit Read		528
Mutex Process 1 (3)	Mutex Process 2(3)[4]	Mutual exclusion	7
MCP missionaries	MCP cannibals (30)[4]	$Num_{missionaries} \geq Num_{cannibals}$	22
4-bit ABP Sender	Modified Receiver (166432)[4]	Liveness checking based on control signal matching	14312

Table 1: Implementation Results

allowed only a single read after each handshake with the producer. The next set of results were obtained when the consumer allowed multiple reads with each handshake. This allowed handling arbitrary read-write pairs. The final three results are well-known NuSMV examples modified to create mismatches. The mutex example was modified such that a violation of the mutual exclusion property occurred. The missionaries and cannibals problem, an abstraction of data-communication between two protocols was also handled successfully. It involved constraining data-communication between protocols such that size of the communication medium (boat) was not exceeded, and at the same time, further restrictions on data-variables (number of cannibals never exceeds the number of missionaries) were also handled. The 4-bit alternating-bit protocol example, was modified to create a control mismatch between the sender and the

receiver, which introduced many faulty paths in the combined system. Using the tableau construction algorithm, a converter that effectively disabled such faulty communication paths was realized. Note that size entry in the second column for the final two results is the combined size of the system (size of $P_1||P_2$).

4 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 formal approach to protocol conversion which alleviates the above problems. Specifications are described in temporal logic and bidirectional communication is allowed. A tableau-based approach using the model checking framework is used to generate converters in polynomial time. We use invariants to handle data-width issues. Fairness properties are used to generate fair converters. We prove that the approach is sound and complete and provide implementation results.

References

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

[2] 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.

[3] 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.

[4] R. Cavada et al. *NuSMV 2.1 User Manual*, June 2003.

[5] Saurav Gorai et al. Session 42: simulation assisted formal verification: 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.

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

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

[8] F. Maraninchi and Y. Remond. Argos: an automaton-based synchronous language. *Computer Languages*, 27:61–92, 2001.

[9] K. Okumura. A formal protocol conversion method. In *ACM SIGCOMM 86 Symposium*, pages 30–37, 1986.

[10] R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *International Conference on Computer Aided Design ICCAD*, 2002.

[11] 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.

[12] R Sinha, P Roop, and S Basu. Automatic synthesis of converters for protocol matching using model checking. Technical Report 00000482, Iowa State University, 2006.