Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 296206, 19 pages
doi:10.1155/2008/296206

Research Article

SoC Design Approach Using Convertibility Verification

Roopak Sinha,[1] Partha S. Roop,[1] and Samik Basu[2]

[1] *Department of Electrical and Computer Engineering, The University of Auckland, Auckland 1142, New Zealand*
[2] *Department of Computer Science, Iowa State University, Ames, Iowa 50011, USA*

Correspondence should be addressed to Roopak Sinha, roopak.sinha@gmail.com

Received 13 April 2008; Accepted 23 October 2008

Recommended by Eric Rutten

Compositional design of systems on chip from preverified components helps to achieve shorter design cycles and time to market. However, the design process is affected by the issue of protocol mismatches, where two components fail to communicate with each other due to protocol differences. Convertibility verification, which involves the automatic generation of a *converter* to facilitate communication between two mismatched components, is a collection of techniques to address protocol mismatches. We present an approach to convertibility verification using module checking. We use Kripke structures to represent protocols and the temporal logic `ACTL` to describe desired system behavior. A tableau-based converter generation algorithm is presented which is shown to be sound and complete. We have developed a prototype implementation of the proposed algorithm and have used it to verify that it can handle many classical protocol mismatch problems along with SoC problems. The initial idea for `ACTL`-based convertibility verification was presented at SLA++P '07 as presented in the work by Roopak Sinha et al. 2008.

1. INTRODUCTION

Systems on chip (SoC) are complex embedded systems built using preverified components (called intellectual property blocks or IPs) chosen from available IP libraries. The various IPs of an SoC may be interconnected via a central system bus on a single chip.

The integration of IPs into an SoC involves addressing key compatibility and communication issues. One such important issue is that of *protocol mismatches* which arise when two or more IPs in an SoC have incompatible protocols. Protocol mismatches may prevent meaningful communication between IPs as they may not be able to correctly exchange control signals (due to *control mismatches*), exchange data (due to *data-width mismatches*), and/or connect to each other at all (due to *interface mismatches*). For example, two IPs that have different handshaking sequences have inherent control mismatches. If their word-sizes (the sizes of their data read/write operations) differ, they suffer from data-width mismatches. Interface mismatches occur when two IPs use different naming conventions for the same control/data signals, disallowing straightforward integration of the IPs into an SoC. Unless any mismatches between IPs are resolved, it is impossible to build an SoC that is consistent with the intended system-level behavior.

Protocol mismatches may be resolved if one or all of the mismatched IPs are modified. However, this process of manual modification is usually ineffective because firstly, it requires significant time and effort to modify complex IPs, and secondly, if requirements change later in the design cycle, further repetitions of manual modification might be required. Protocol mismatches may also be resolved by using *convertibility verification* (or protocol conversion) techniques, which involve the generation of a *converter*, some additional glue, that guides mismatched components to satisfy system-level behavioral requirements while bridging the mismatches as well.

A possible solution to convertibility verification comes from the verification of open systems using module checking [1]. An embedded system may behave differently under different environments, and verification of an embedded system under the influence of different environments was studied in [2]. In [3], local module checking, an approach to build an environment under which a system satisfies a given specification, was presented. In this paper, we adapt local module checking for convertibility verification to build a converter under which IPs satisfy given system-level specifications.

The main features of the proposed solution are as follows. Firstly, IPs are represented using Kripke structures and communicate synchronously with each other through input/output control signals. The desired behavior of the interaction between IPs is described using the temporal logic ACTL. ACTL is the universal fragment of CTL which allows only universal path quantification. ACTL is used for two main reasons. Firstly, the intended behavior of the interaction between mismatched protocols is usually required to be exhibited over all paths. Hence, universally quantified specifications are usually sufficient to describe such intended behavior. Secondly, the handling of existentially quantified formulas (EU and EG) results in the high (exponential) complexity of module checking [2].

Given two mismatched IPs and a set of ACTL specifications to be satisfied by their interaction, an automatic converter generation algorithm is employed to generate a converter if possible. We prove that the algorithm is sound and complete and can be used to resolve many commonly encountered control mismatches.

The rest of this paper is organized as follows. Section 2 discusses literature related to the proposed approach. An illustrative example is presented in Section 3. Section 4 presents the formal description of protocols and their interaction. Section 5 shows how specifications are described in our setting. Section 6 defines converters and shows how they control a given pair of mismatched protocols. The converter generation and extraction algorithm is presented in Section 7. Implementation results are given in Section 8 followed by concluding remarks in Section 9.

2. RELATED WORK

A number of techniques have been developed to address the problem of protocol conversion using a wide range of formal and informal settings with varying degrees of automation—projection approach [4], quotienting [5], conversion seeds [6], synchronization [7], and supervisory control theory [8], just to name a few. Some techniques, like converters for protocol gateways [9] and interworking networks [10], rely on ad hoc solutions. Some other approaches, like protocol conversion based on conversion seeds [6] and protocol projections [4], require significant user expertise and guidance. While this problem has been studied in a number of formal settings [4, 6, 7, 11], only recently some formal verification-based solutions have been proposed [8, 12–14].

The closest ones to our approach are [12, 13]. In [12], the authors present an approach using finite state machines to represent both the protocols and the desired specifications. A game-theoretic framework is used to generate a converter. This solution is restricted only to protocols with half-duplex communication between them. D'silva et al. [13, 15] present synchronous protocol automata to allow formal protocol specification and matching, as well as converter synthesis. The matching criteria between protocols are based on whether events are blocking or nonblocking and no additional specifications can be used. The approach allows model checking only as an auxiliary verification step to ensure that the generated converter is correct.

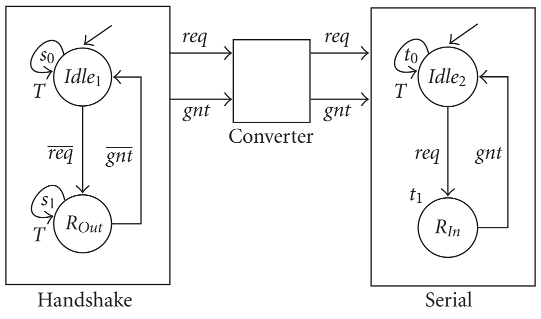

FIGURE 1: The handshake-serial protocol pair.

In contrast to the above techniques, we use temporal logic to represent desired functionality of the combined protocols. Being based on temporal logic, our technique can define desired properties succinctly and with a higher level of granularity. For example, a desired behavior of the combination may be sequencing of events such that event a in protocol P_1 always happens before event b in P_2. Also, as our technique is based on the (tableau-based) module checking algorithm, the converter synthesized is correct by construction, unlike [13].

The presented approach, based on extending the local module checking algorithm [3], is similar to the synthesis of discrete controllers using temporal logic specifications [16, 17]. In [17], a controller synthesis paper using temporal logic (ctl*) specifications, the authors note that the problems of module checking and supervisory control are *duals* of each other. The problem of conversion can therefore be handled by extending any of these dual approaches. However, it must be noted that the conversion problem addressed in this paper is not a straight forward implementation of either module checking or supervisory control. It requires nontrivial extensions including the handling of multiple protocols, input-output signals, exchange of signals between protocols, and special states such as output-only states or delayed-output states (described later in Section 4).

3. ILLUSTRATIVE EXAMPLE

We motivate our protocol conversion technique using the following example of two mismatched IPs. Figure 1 shows the protocols of two IPs, handshake and serial, that need to communicate with each other. This example was presented originally in [12] and has been adapted by adding state labels to each protocol. The handshake protocol emits the outputs *req* and *gnt* which can be read by the serial protocol. The IPs are expected to communicate by exchanging these I/O signals infinitely often.

The protocols of the two devices are described as follows. In its initial state s_0, the handshake protocol emits the signal $\overline{req}$ and makes a transition to state s_1. In s_1, it can wait for an indefinite number of clock ticks (shown as the self-loop on s_1) before writing the signal $\overline{gnt}$ and moving back to its initial state s_0. Outputs are distinguished from inputs by placing a bar over them.

The serial protocol operates as follows. In its initial state t_0, it waits indefinitely (using the self-loop on t_0) for the signal *req* and makes a transition to state t_1 when *req* is available. In t_1, it immediately requires the input *gnt* to make a transition back to its initial state t_0.

Intuitively, the mismatch between the two protocols can be described as follows. The handshake protocol can wait indefinitely before writing the signal $\overline{gnt}$ once it has emitted the signal $\overline{req}$. On the other hand, the serial protocol needs *gnt* to be available immediately following the reception of *req*. Given this basic difference in their protocols, it is possible that their interaction results in the violation of system-level specifications, such as

(1) a protocol must never attempt to read a signal before it is emitted by the other;

(2) each emission of a signal by a protocol must be read successfully by the other protocol before it attempts to emit more instances of the same signal.

Due to different protocols, the IPs cannot guarantee satisfaction of the above specifications. To enable these IPs to interact in the desired fashion, a converter to bridge inconsistencies between their protocols is generated. The converter acts as a communication medium between the protocols, as shown in Figure 1, and guides the interaction between the two protocols by controlling the exchange of control signals between them. The resulting system, in the presence of the converter, guarantees the satisfaction of the given system-level properties.

We use the above handshake-serial example throughout the the rest of this paper to illustrate our approach.

4. MODEL OF PROTOCOLS

4.1. Kripke structures

Protocols are formally represented using Kripke structures, defined as follows.

Definition 1 (Kripke structure). A Kripke structure (KS) is a finite state machine represented by a tuple $\langle AP, S, s_0, \Sigma, R, L, clk\rangle$, where

(i) AP is a set of atomic propositions,

(ii) S is a finite set of states,

(iii) $s_0 \in S$ is the initial state,

(iv) $\Sigma = \Sigma_I \cup \Sigma_O \cup \{T\}$ is a finite set of events, where Σ_I is the set of all input events, Σ_O is the set of all output events, and T is the tick event of the clock *clk*,

(v) $R : S \times \Sigma \rightarrow S$ is a *total* (with respect to S) and *deterministic* transition function,

(vi) $L : S \rightarrow 2^{AP}$ is the state labelling function,

(vii) *clk* is the system clock event.

Transitions of a Kripke structure can be divided into three categories: *input transitions*, *output transitions*, and *tick transitions*. All three types of transitions trigger with respect to the tick T of *clk* which represents the rising edge of the clock. An input transition from a system's current state triggers when the input trigger of the transition is present at the next tick of the clock. Similarly, an output transition triggers when the system, at the next clock tick, emits the output corresponding to a given transition. Finally, in case of a tick transition, a transition simply triggers on the next clock tick without the Kripke structure reading any inputs or emitting any outputs. A tick transition is used to implement delays. In all three cases, the Kripke structure moves from a current state to a destination state.

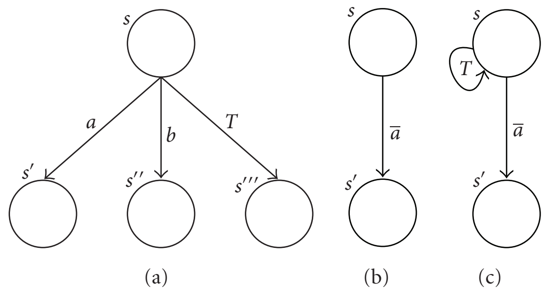

FIGURE 2: Possible types of states in a well-formed Kripke structure. (a) An input state. (b) An output-only state. (c) A delayed-output state.

The presence of an event a as an input is denoted as a ($a \in \Sigma$) over a transition whereas the emission of a signal b as an output is denoted as $\overline{b}$ ($\overline{b} \in \Sigma$). In case no input/output triggers are present, the transition with respect to solely the clock's tick event T is taken ($T \in \Sigma$). Note that R is total with respect to S, implying that each reachable state in a KS must have at least one outgoing transition. Furthermore, R is deterministic, implying that each reachable state can have only one transition to a unique successor state for any particular input or output event. The shorthand $s \xrightarrow{a} s'$ is used to represent the transitions of a Kripke structure ($s' = R(s, a)$).

Restrictions

The following restrictions are placed on Kripke structures representing IP protocols.

(1) Well-formedness. A well-formed KS has only the following types of states (Figure 2 shows the different types of states in a well-formed Kripke structure).

(i) *Input states*. A state $s \in S$ is an input state if none of its transitions results in the emission of an output. In other words, all of its transitions are triggered by input events or T (see Figure 2(a)). Whenever the KS reaches s, if no input triggers are present in the next tick, the tick transition is taken. In case there is no tick transition, s must be provided with an input that enables one of its transition.

For example, consider the state t_1 of the serial protocol presented in Figure 1. The state must be provided with the input *gnt* in the next tick, otherwise its behavior is not defined.

(ii) *Output-only states*. A state $s \in S$ is an output-only state if it has only one transition that is triggered by an output signal (see Figure 2(b)). Whenever a KS reaches an output-only state, in the next tick, the lone output transition is taken (and the corresponding output is emitted). We introduce the function *OutputOnly*, where *OutputOnly*(s) returns true when the state s is an output-only state.

(iii) *Delayed-output states.* A state $s \in S$ is a delayed-output state when it has exactly two transitions: one triggered by an output signal and the other triggered by T. The tick transition must be a self-loop ($s \xrightarrow{T} s$) and must model an arbitrary delay before the output transition is taken (see Figure 2(c)). We introduce the function *DelayedOutput*, where *DelayedOutput*(s) returns true when the state s is a delayed-output state.

We restrict states in a well-formed KS to have at most one output transition to ensure determinism. A state with two output transitions is nondeterministic because whenever the KS reaches such a state, it is not known which output will be emitted (and which transition will be taken) in the next tick.

For example, state s_0 of the handshake protocol presented in Figure 1 is a delayed-output state with a tick transition modelling an arbitrary delay. Hence, *DelayedOutput* $(s_0) = tt$.

(2) Shared clock. Each KS must execute using the same clock, allowing multiple protocols to only make synchronous transitions. This restriction assumes that there are no clock mismatches between protocols.

Consider the protocol for the handshake IP presented in Figure 1. The protocol can be described as the Kripke structure $P_1 = \langle AP_1, S_1, s_{0_1}, \Sigma_1, R_1, L_1, clk_1\rangle$, where $AP_1 = \{Idle_1, R_{Out}\}$, $S_1 = \{s_0, s_1\}$, $s_{0_1} = s_0$, $\Sigma_1 = \{\overline{req}, \overline{gnt}, T\}$, $R_1 = \{s_0 \xrightarrow{T} s_0,\ s_0 \xrightarrow{\overline{req}} s_1,\ s_1 \xrightarrow{T} s_1,\ s_1 \xrightarrow{\overline{gnt}} s_0\}$, $L_1(s_0) = \{Idle_1\}$, and $L_1(s_1) = \{R_{Out}\}$.

In our setting, all Kripke structures execute synchronously using the same clock. Hence, for the protocols P_1 and P_2 described above, $clk_1 = clk_2 = clk$.

4.2. Composition of Kripke structures

The parallel composition defines the unrestricted composite behavior of two protocols when they are physically connected (without a converter).

Definition 2 (parallel composition). Given Kripke structures $P_1 = \langle AP_1, S_1, s_{0_1}, \Sigma_1, R_1, L_1, clk\rangle$ and $P_2 = \langle AP_2, S_2, s_{0_2}, \Sigma_2, R_2, L_2, clk\rangle$, their parallel composition, denoted by $P_1\|P_2$, is $\langle AP_{1\|2}, S_{1\|2}, s_{0_{1\|2}}, \Sigma_{1\|2}, R_{1\|2}, L_{1\|2}, clk\rangle$, where

(i) $AP_{1\|2} = AP_1 \cup AP_2$,

(ii) $S_{1\|2} \subseteq S_1 \times S_2$,

(iii) $s_{0_{1\|2}} = (s_{0_1}, s_{0_2})$,

(iv) $\Sigma_{1\|2} \subseteq \Sigma_1 \times \Sigma_2$,

(v) $R_{1\|2} : S_{1\|2} \times \Sigma_{1\|2} \to S_{1\|2}$ is the transition function such that

$$[s_1 \xrightarrow{\sigma_1} s_1'] \wedge [s_2 \xrightarrow{\sigma_2} s_2'] \Longrightarrow [(s_1, s_2) \xrightarrow{(\sigma_1,\sigma_2)} (s_1', s_2')], \quad (1)$$

(vi) finally, $L_{1\|2}[(s_1, s_2)] = L_1(s_1) \cup L_2(s_2)$.

Each state s of the parallel composition corresponds to unique individual states s_1 and s_2 in the protocols, and its labels contain every proposition contained as a label of any of its constituent states. The initial state of the composition is (s_{0_1}, s_{0_2}). Each state $s = (s_1, s_2)$ of the composition makes a transition to a successor state $s' = (s_1', s_2')$ when both protocol states s_1 and s_2 make individual transitions to s_1' and s_2', respectively. The transition trigger is obtained by combining the transition triggers of the participating protocols. Note that the set of states $S_{1\|2}$ is a subset of the cartesian product of the sets of states S_1 and S_2. This is because $S_{1\|2}$ contains only those states that are reached by both protocols making simultaneous transitions. For the same reason, the event set $\Sigma_{1\|2}$ does not contain all elements of the cartesian product of Σ_1 and Σ_2.

Semantics of the transitions of states in the parallel composition

The parallel composition of two well-formed KS can contain different types of states. The state (s, t) resulting from the parallel composition of two states s and t can be of 4 different types, depending on types of s and t, as shown in Figure 1.

An output state (see Figure 3(a)) in the parallel composition results if each of the states s and t is either an output-only or a delayed-output state. If both s and t are output-only states with (sole) transitions to states s' and t' triggered by the outputs $\overline{a}$ and $\overline{b}$, the composite state (s, t) has only one transition to (s', t') which results in the emission of the outputs $\overline{a}$ and $\overline{b}$. If s and/or t have tick transitions (in case one or both are delayed-output states), the composite state (s, t) may have additional transitions triggered by a combination of T and the outputs a and b, as shown in Figure 3(a).

An input-output state (see Figure 3(b)) results when one of the states s or t is an input state and the other is an output-only state. Note that for input-output states, all transitions trigger with respect to one KS emitting an output and the other reading an input (or the tick event).

An input-delayed-output state (see Figure 3(c)) is obtained when one of the states s or t is an input state and the other is a delayed-output state. In this case, half of the transitions of the input-delayed-output state corresponds to one KS emitting an output and the other reading an input (or making a T-transition). For example, in Figure 3(c), the transitions $(s, t) \xrightarrow{a,\,\overline{c}} (s', t')$ and $(s, t) \xrightarrow{b,\,\overline{c}} (s', t'')$ result in the output $\overline{c}$ being emitted. We introduce a function $Out : S_{1\|2} \to 2^{\Sigma_{1\|2}}$, where for any input-delayed-output state $s \in S_{1\|2}$, $Out(s)$ returns the set of transition triggers that result in the emission of an output (e.g., in Figure 3(c), $Out((s, t)) = \{(a, \overline{c}), (b, \overline{c})\}$). The remaining transitions of an input-delayed-output state correspond to one protocol taking a tick transition and the other reading an input (hence delaying the emission of the output). We introduce a function $Delay : S_{1\|2} \to 2^{\Sigma_{1\|2}}$, where for any input-delayed-output state, $s \in S_{1\|2}$, $Delay(s)$ returns the set of transition triggers that do not result in the emission of an output (e.g., in Figure 3(c), $Delay((s, t)) = \{(a, T), (b, T)\}$). Note that taking the delay transitions merely delays the emission of an output. For example, if the transition $(s, t) \xrightarrow{b,\,T} (s, t'')$

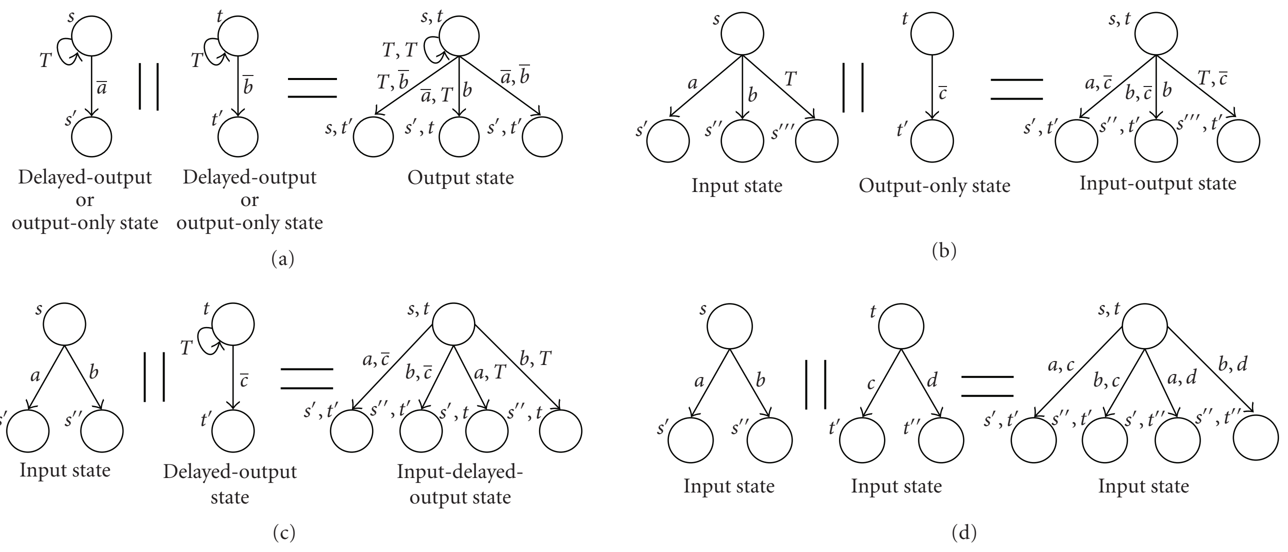

FIGURE 3: Possible types of states in the composition of two well-formed Kripke structures. (a) An output state. (b) An input-output state. (c) An input-delayed-output state. (d) An input-only state.

is taken by the state (s, t) in Figure 3(c), the state (s, t'') still refers to the delayed-output state s which is capable of emitting the output $\overline{c}$ in the next tick after it is reached.

Finally, an input state (see Figure 3(d)) results when both the states s and t are input states. Each transition in the input state is triggered when a pair of inputs (or T) is read.

Illustration

The parallel composition $P_1 \| P_2$ of P_1 and P_2 in Figure 1 is shown in Figure 4. The initial state (s_0, t_0) corresponds to the initial states s_0 and t_0 of the two protocols and is labelled by $Idle_1$ and $Idle_2$.

Each transition from the state (s_0, t_0) is triggered when both s_0 and t_0 take their respective transitions. For example, the transition from state (s_0, t_0) to state (s_1, t_1) is triggered when s_0 makes a transition to s_1 (emitting $\overline{gnt}$) and t_0 makes a simultaneous transition to t_1 (reading gnt).

All states in the parallel composition presented in Figure 4 are input-delayed-output states as each corresponds to a delayed-output state from P_1 and an input state from P_2 (P_1 has only delayed-output states whilst P_2 has only input states). For the input-delayed-output state (s_0, t_0), $Out((s_0, t_0)) = \{(\overline{req}, T), (\overline{req}, req)\}$ while $Delay((s_0, t_0)) = \{(T, T), (T, req)\}$.

5. SPECIFICATIONS

Specifications of correct interaction between protocols may be formally described using temporal logic. In our approach, ACTL ($\forall$CTL), the universal fragment of CTL is used. ACTL is a branching time temporal logic with universal path quantifiers. As described earlier, ACTL is used because it can describe common protocol conversion specifications and at the same time results in a lower worst-case complexity of the conversion algorithm than CTL.

ACTL is defined over a set of propositions using temporal and boolean operators as follows:

$$\phi \longrightarrow P|\neg P|tt|ff|\phi \wedge \phi|\phi \vee \phi|\mathtt{AX}\phi|\mathtt{A}(\phi \,\mathtt{U}\, \phi)|\mathtt{AG}\phi|. \quad (2)$$

Note that, in the above, negation is not applied on temporal and boolean operators. This restriction is due to the fact that the converter generation algorithm uses tableau generation (similar to the local module checking algorithm presented in [3]), where tableau rules can operate only on formulas where negations are applied over propositions.

Semantics of ACTL formula, φ denoted by $[\![\varphi]\!]_M$, is given in terms of the set of states in Kripke structure, M, which satisfies the formula (see (3)):

$$\begin{aligned}
[\![p]\!]_M &= \{s \mid p \in L(s)\},\\
[\![tt]\!]_M &= S,\\
[\![ff]\!]_M &= \varnothing,\\
[\![\varphi \wedge \psi]\!]_M &= [\![\varphi]\!]_M \cap [\![\psi]\!]_M,\\
[\![\varphi \vee \psi]\!]_M &= [\![\varphi]\!]_M \cup [\![\psi]\!]_M,\\
[\![\mathtt{AX}\varphi]\!]_M &= \{s \mid \forall s \longrightarrow s' \wedge s' \in [\![\varphi]\!]_M\},\\
[\![\mathtt{A}(\varphi \,\mathtt{U}\, \psi)]\!]_M &= \{s \mid \forall (s = s_1 \longrightarrow s_2 \longrightarrow \cdots),\\
&\qquad \exists j \cdot (s_j \vDash \psi) \wedge \forall i < j \cdot (s_i \vDash \varphi)[i, j \geq 1]\},\\
[\![\mathtt{AG}\varphi]\!]_M &= \{s \mid \forall s = s_1 \longrightarrow s_2 \longrightarrow \cdots \wedge\\
&\qquad \forall i \cdot s_i \vDash \varphi (i \geq 1)\}.
\end{aligned} \quad (3)$$

A state $s \in S$ is said to satisfy an ACTL formula expression φ, denoted by $M,\ s \vDash \varphi$, if $s \in [\![\varphi]\!]_M$. We will omit M from the $[\![\,]\!]$ and the $\vDash$ relation if the model is evident in the context. The short hand $M \vDash \varphi$ is used to indicate $M,\ s_0 \vDash \varphi$.

The desired behavior of the interaction of the handshake and serial protocols presented in Section 3 is described formally using the following ACTL formulas.

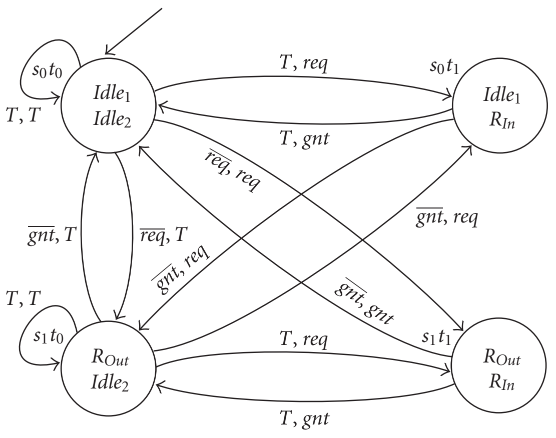

Figure 4: $P_1 \| P_2$ for the handshake-serial protocol pair.

[φ_1] $\mathtt{AG}(Idle_1 \wedge Idle_2 \Rightarrow \mathtt{AX}(R_{Out} \vee \neg R_{In}))$. Whenever a state in the parallel composition is reached where both protocols are in their initial states, in all its successors, the serial protocol does not advance to a state labelled R_{In} unless the handshake protocol simultaneously moves to a state labelled R_{Out}. In other words, the serial protocol does not read the *req* signal unless the handshake emits it simultaneously.

[φ_2] $\mathtt{AG}(R_{In} \wedge R_{Out} \Rightarrow \mathtt{AX}(Idle_1 \vee \neg Idle_2))$. Whenever a state in the parallel composition is reached where both protocols are in states labelled by R_{Out} and R_{In}, respectively, in all its successors, the serial protocol does not move to a state labelled $Idle_2$ unless the handshake protocol simultaneously moves to a state labelled $Idle_1$. In other words, the serial protocol does not read the *gnt* signal unless handshake emits it simultaneously.

[φ_3] $\mathtt{AG}(R_{Out} \wedge Idle_2 \Rightarrow \mathtt{AX}(R_{In} \vee \neg Idle_1))$. Whenever a state in the parallel composition is reached where the handshake protocol is in a state labelled by R_{Out} (*req* emitted) and the serial protocol is in its initial state, in all successors, the handshake protocol must not move to its initial state unless the serial protocol moves to a state labelled R_{In}. In other words, once *req* has been emitted by the handshake protocol, it must be read by the serial protocol before handshake moves back to its initial state and emits *gnt*.

[φ_4] $\mathtt{AG}(Idle_1 \wedge R_{In} \Rightarrow \mathtt{AX}(\neg R_{Out} \vee \neg Idle_2))$. Whenever a state in the parallel composition is reached where the handshake protocol is in its initial state and the serial protocol is in the state labelled by R_{In} (*req* read), in all successors, the handshake protocol must not move to a state labelled by R_{Out} unless the serial protocol moves to its initial state. In other words, once *gnt* has been emitted by the handshake protocol (by moving to its initial state), it must be read by the serial protocol before handshake moves back to its initial state and emits *req*.

In Figure 4, it can be seen that certain paths are inconsistent with the specifications described in Section 3. For example, the transition from the initial state (s_0, t_0) to (s_0, t_1) results in the serial protocol reading the input *req* before handshake protocol has emitted it, hence violating the property φ_1.

6. PROTOCOL CONVERTERS

The composition $P_1 \| P_2$ (see Figure 4) represents the unconstrained behavior of the protocols including undesirable paths introduced due to mismatches. A converter is needed to bridge the mismatches appropriately. This section formally introduces converters and also the control exerted by a converter over participating protocols.

6.1. I/O relationships between converters and protocols

Firstly, we describe the relationship between the input/output signals of a converter and the participating protocols. Inputs to the converter are outputs from the protocols and vice versa. For example, Figure 1 shows the handshake and serial protocols connected via a converter. The converter reads the outputs $\overline{req}$ and $\overline{gnt}$ of the handshake protocol and emits the signals *gnt* and *req* to be read by the serial protocol. This concept of *duality* of I/O signals is formalized as follows.

Definition 3 (duality). Given a set of input signals Σ_I, a set of output signals Σ_O, and two signals a and b, such that $\mathcal{D}(a,b) = tt$ if and only if:

(i) $a \in \Sigma_I$ and $b \in \Sigma_O$ such that $b = \overline{a}$, or

(ii) $a \in \Sigma_O$ and $b \in \Sigma_I$ such that $a = \overline{b}$, or

(iii) a is the tick event T and $b = a = T$.

Generalizing to pairs of signals, given two pairs $A = (a_1, a_2)$ and $B = (b_1, b_2)$ of signals $\mathcal{D}(A,B) = tt$ if and only if $\mathcal{D}(a_1, b_1)$ and $\mathcal{D}(a_2, b_2)$.

Based on the above definition, each input to the converter is a *dual* of an output of participating protocols, and vice versa. Also, the tick event T is its own dual. This is so because a converter does not need to provide any inputs/outputs when protocols make tick-only transitions.

6.2. The role of a converter

A converter $\mathcal{C}$ for the parallel composition $P_1 \| P_2$ of the two protocols is represented as the Kripke structure $\langle AP_{\mathcal{C}}, S_{\mathcal{C}}, c_0, \Sigma_{\mathcal{C}}, R_{\mathcal{C}}, L_{\mathcal{C}}, clk \rangle$. Many converters that can disable transitions in $P_1 \| P_2$ to ensure the satisfaction of the given ACTL formulas can be envisaged. However, some of these may be incorrect. For example, an output transition in one of the protocols is *uncontrollable* because a converter cannot prevent it from happening. A converter that attempts to block or disable such a transition is not realizable and hence, deemed incorrect. In our setting, we want to synthesize *correct* converters. This notion of correctness is formalized by defining a *conversion refinement relation* between converters and participating protocols. We first provide the intuition behind the converter refinement relation by describing the role of a correct converter in controlling a given protocol pair.

Given a converter $\mathcal{C} = \langle AP_{\mathcal{C}}, S_{\mathcal{C}}, c_0, \Sigma_{\mathcal{C}}, R_{\mathcal{C}}, L_{\mathcal{C}}, clk \rangle$, every state $c \in S_{\mathcal{C}}$ controls exactly one state $s \in S_{1\|2}$ and we say that c is *matched* to s. The following restrictions must be satisfied by the matched states c and s.

(1) Every transition out of c must correspond to a dual transition out of s. A transition $s \xrightarrow{(a,b)} s'$ is a dual of a transition $c \xrightarrow{(c,d)} c'$ if $\mathcal{D}(a,c)$and $\mathcal{D}(b,d)$ (transitions have dual I/O). Further, the destinations states of the dual transitions must also match. This restriction forms the basis of the control of a converter state over the matched state in the given parallel composition. The converter allows a transition in s only if c has a transition that is a dual of the s transition. In case neither has a dual transition for a given transition in s, the transition is *disabled* by the converter, nor does the converter disable all transitions of s.

(2) Whenever s is an output state (see Figure 3(a)), the converter is not permitted to disable any transitions of s. In other words, corresponding to each transition of s, c must have a dual transition. This restriction comes from the fact that each transition of an output state corresponds to the protocols emitting outputs or choosing to remain in the current state. These transitions cannot be disabled by the converter as the protocols do not read any inputs during any transition in an output-state.

(3) If s is an input-delayed-output state (see Figure 3(c)), the converter has to enable precisely one output transition and one delay transition of s. An input-delayed-output state corresponds to a delayed-output state in one protocol and an input state in the other and has two types of transitions. A delayed-output protocol state has two types of transitions: output transitions (returned by the set *Out*) and delay transitions (returned by the set *Delay*). Output transitions (triggered by the elements of $Out(s)$) result in an output being emitted by the protocol while the delay transitions (triggered by the elements of $Delay(s)$) result in the protocols delaying the emission of the output to at least the next tick. As the converter cannot force the protocols from emitting an output or taking a delay transition, it must enable one transition each from the sets $Out(s)$ and $Delay(s)$.

We now define the conversion refinement relation.

Definition 4 (conversion refinement relation). Let $P_1\|P_2$ be the parallel composition of two protocols P_1 and P_2. A state $s \in S_{1\|2}$ can be represented as the state (s_1, s_2), where $s_1 \in S_1$ and $s_2 \in S_2$.

Given a converter $\mathcal{C} = \langle AP_{\mathcal{C}}, S_{\mathcal{C}}, c_0, \Sigma_{\mathcal{C}}, R_{\mathcal{C}}, L_{\mathcal{C}}, clk \rangle$, a relation $\mathcal{B} \subseteq S_{\mathcal{C}} \times S_{1\|2}$ is a *conversion refinement relation* if for any $(c,s) \in \mathcal{B}$, the following conditions hold.

(1) *Enabled transitions*. For all $c \xrightarrow{\sigma'} c'$, there exists $s \xrightarrow{\sigma} s'$, for some $s, s' \in S_{1\|2}$, such that $\mathcal{D}(\sigma, \sigma')$ and $(c', s') \in \mathcal{B}$.

(2) *Output states restriction*. If s is an output state, then there must exist a $c \xrightarrow{\sigma'} c'$ for every $s \xrightarrow{\sigma} s'$ such that $\mathcal{D}(\sigma, \sigma')$ and $(c', s') \in \mathcal{B}$.

(3) *Input-delayed-output states restriction*. If s is an input-delayed-output state, then c must have precisely two transitions $c \xrightarrow{\sigma_c'} c'$ and $c \xrightarrow{\sigma_c''} c''$ that match transitions $s \xrightarrow{\sigma_s'} s'$ and $s \xrightarrow{\sigma_s''} s''$ of s, respectively, such that $\mathcal{D}(\sigma_c', \sigma_s')$, $\mathcal{D}(\sigma_c'', \sigma_s'')$, $\sigma_s' \in Out(s)$, $\sigma_s'' \in Delay(s)$, $(c', s') \in \mathcal{B}$ and $(c'', s'') \in \mathcal{B}$.

A conversion refinement relation between a converter KS and a given parallel composition states the constraints on a converter state c that controls a state s in the composition.

Illustration

Figure 5 presents a converter $\mathcal{C}$ for the composition of handshake-serial protocol pair presented in Figure 4. There exists a conversion refinement relation $\mathcal{B}$ between the converter and the protocols, where

(1) $\mathcal{B}(c_0, (s_0, t_0))$: each transition of c_0 is dual to some transition of (s_0, t_0) (rule 1) and c_0 has at least one transition (rule 2). State (s_0, t_0), being an input-delayed-output state, requires c_0 to enable one output transition and another tick transition (rule 3). c_0 satisfies this restriction by having the transition $c_0 \xrightarrow{(req,T)} c_1$ that reads *req* if it is emitted by the protocols and the transition $c_0 \xrightarrow{(T,T)} c_0$ allowing (s_0, t_0) to wait;

(2) $\mathcal{B}(c_1, (s_1, t_0))$: each transition of c_1 is dual to some transition of (s_1, t_0) (rule 1) and c_1 has at least one transition (rule 2). State (s_1, t_0), being an input-delayed-output state, requires c_1 to one output transition and another tick transition (rule 3). c_1 satisfies this restriction by having the transition $c_1 \xrightarrow{(gnt,\overline{req})} c_2$ that reads *gnt* if it is emitted by the protocols and the transition $c_1 \xrightarrow{(T,T)} c_1$ allowing (s_1, t_0) to wait;

(3) $\mathcal{B}(c_2, (s_0, t_1))$: each transition of c_2 is dual to some transition of (s_0, t_1) (rule 1) and c_2 has at least one transition (rule 2). State (s_0, t_1), being an input-delayed-output state, requires c_2 to enable one output transition and another tick transition (rule 4). c_2 satisfies this restriction by having the transition $c_2 \xrightarrow{(req,\overline{gnt})} c_1$ that reads *req* if it is emitted by the protocols and the transition $c_2 \xrightarrow{(T,\overline{gnt})} c_0$ allowing (s_0, t_1) to wait.

We use the conversion refinement relation to define correct converters.

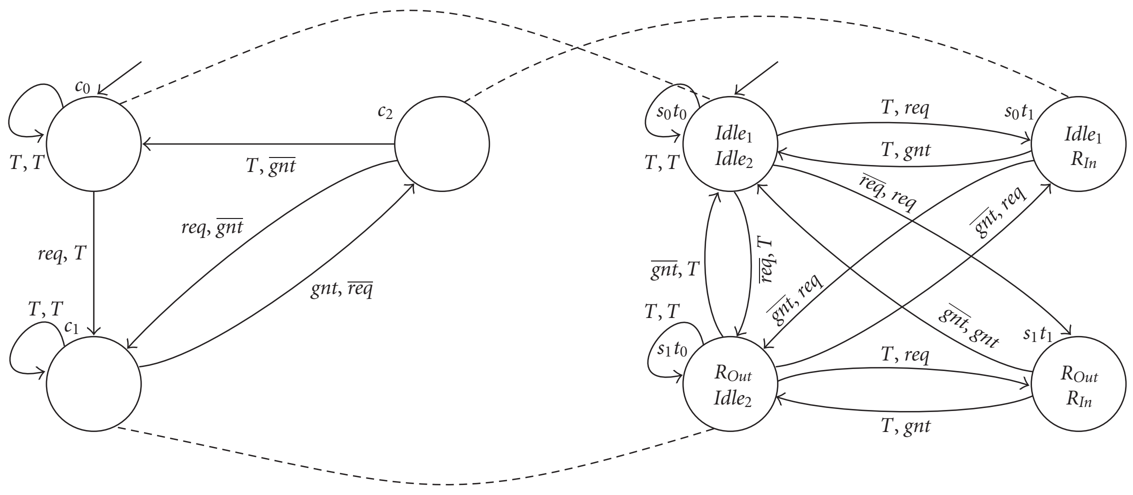

FIGURE 5: The converter $\mathcal{C}$ for the handshake-serial protocol pair.

6.3. Definition of converters

Having defined the relationship between the states of a converter and the states of participating protocols, we now define correct converters as follows.

Definition 5 (converter). A correct converter $\mathcal{C}$ for two protocols P_1 and P_2 with parallel composition $P_1\|P_2$ is a Kripke structure $\langle AP_{\mathcal{C}}, S_{\mathcal{C}}, c_0, \Sigma_{\mathcal{C}}, R_{\mathcal{C}}, L_{\mathcal{C}}, clk\rangle$, where

(1) $AP_{\mathcal{C}} = \varnothing$;

(2) $S_{\mathcal{C}}$ is a finite set of states and there exists a conversion refinement relation $\mathcal{B}$ such that for every state $c \in S_{\mathcal{C}}$, there is a state $s \in S_{1\|2}$ such that $\mathcal{B}(c,s)$;

(3) c_0 is the initial state such that $\mathcal{B}(c_0, s_{0_{1\|2}})$;

(4) $\Sigma_{\mathcal{C}} \subseteq \{a : b \in \Sigma_1 \wedge \mathcal{D}(a,b)\} \times \{a : b \in \Sigma_2 \wedge \mathcal{D}(a,b)\}$;

(5) $R_{\mathcal{C}} : S_{\mathcal{C}} \times \Sigma_{\mathcal{C}} \rightarrow S_{\mathcal{C}}$ is a *total* (with respect to $S_{\mathcal{C}}$) transition function;

(6) $L(c) = \varnothing$ for any state $c \in S_{\mathcal{C}}$.

A converter is a KS whose states are related by a conversion refinement relation to the states of the given parallel composition. Its inputs are the outputs from the parallel compositions and its outputs are the inputs to the parallel composition. Converter states do not have any state labels. The converter also operates on the same clock *clk* as the protocols. Note that converters are required to have a transition function that is *total* with respect to $S_{\mathcal{C}}$. This means that for every state $c \in S_{\mathcal{C}}$, there must be at least one transition $c \xrightarrow{\sigma_c} c'$ for some $\sigma_c \in \Sigma_{\mathcal{C}}$ and $c' \in S_{\mathcal{C}}$.

Illustration

The converter (see Figure 5) for the handshake-serial pair has the following elements.

(i) $S_{\mathcal{C}} = \{c_0, c_1, c_2\}$ with c_0 as the initial state.

(ii) $AP_{\mathcal{C}} = \varnothing$, and for any state c in the converter, $L(c) = \varnothing$.

(iii) $\Sigma_{\mathcal{C}} = [\{\overline{req}, \overline{gnt}, T\} \times \{req, gnt, T\}]$.

(iv) As noted earlier, there exists a conversion refinement relation between the states of $\mathcal{C}$ and $P_1\|P_2$.

6.4. Lock-step composition

The control of a converter over a given parallel composition is defined using the // operator as follows.

Definition 6 (lock-step converter composition). Given the KS $P_1\|P_2 = \langle AP_{1\|2}, S_{1\|2}, s_{0_{1\|2}}, \Sigma_{1\|2}, R_{1\|2}, L_{1\|2}, clk\rangle$ and a converter $\mathcal{C} = \langle AP_{\mathcal{C}}, S_{\mathcal{C}}, s_{\mathcal{C}0}, \Sigma_{\mathcal{C}}, R_{\mathcal{C}}, L_{\mathcal{C}}, clk\rangle$, such that there exists a conversion refinement relation $\mathcal{B}$ between the states of the $\mathcal{C}$ and $P_1\|P_2$, the lock-step composition $\mathcal{C}//(P_1\|P_2) = \langle AP_{1\|2}, S_{\mathcal{C}//1\|2}, s_{0_{\mathcal{C}//(1\|2)}}, \Sigma_{1\|2}, R_{\mathcal{C}//(1\|2)}, L_{\mathcal{C}//(1\|2)}, clk\rangle$, where

(1) $S_{\mathcal{C}//1\|2} = \{(c,s) : c \in S_{\mathcal{C}} \wedge s \in S_{1\|2} \wedge \mathcal{B}(c,s)\}$;

(2) $s_{0_{\mathcal{C}//(1\|2)}} \in S_{\mathcal{C}//1\|2}$ is the initial state. $s_{0_{\mathcal{C}//(1\|2)}} = (c_0, s_{0_{1\|2}})$;

(3) $R_{\mathcal{C}//(1\|2)} : S_{\mathcal{C}//(1\|2)} \times \Sigma_{1\|2} \rightarrow S_{\mathcal{C}//(1\|2)}$ is the transition function, where for each state $s_{\mathcal{C}//(1\|2)} \in S_{\mathcal{C}//(1\|2)}$ such that $s_{\mathcal{C}//(1\|2)} = (c,s)$, that has the following transitions:

$$\begin{bmatrix} c \xrightarrow{\sigma_c} c' \wedge s \xrightarrow{\sigma_s} s' \\ \bigwedge \\ \mathcal{D}(\sigma_c, \sigma_s) \end{bmatrix} \Longrightarrow (c,s) \xrightarrow{\sigma_s} (c', s'). \tag{4}$$

(4) $L_{\mathcal{C}//(1\|2)}(c,s) = L_{1\|2}(s_{1\|2})$.

The lock-step composition ensures that states in the protocols take only those transitions that are allowed by the converter. Each state in the lock-step composition corresponds to a state in the converter and its corresponding state in the parallel composition of the given participating protocols. For example, the initial state of the lock-step composition corresponds to the initial state of the converter and the initial state of the given parallel composition. For any state in the lock-step composition, a transition is allowed when its constituent converter state has a transition which is dual to a transition in its corresponding state in the given parallel composition. In other words, when a transition in the converter provides any outputs needed by the participating protocols to trigger a transition in their composition, and the outputs emitted by the protocols are read as inputs by the converter during the same transition, a transition in the lock-step composition triggers. Similarly, if the protocols emit any outputs in a transition or do a tick transition, they are read by the converter in a dual transition. The presence of dual transitions is guaranteed due to the presence of a conversion refinement relation between the states of the converter and the protocols.

Illustration

Figure 6 presents the lock-step composition $\mathcal{C}//(P_1\|P_2)$ for handshake-serial protocol pair presented in Figure 4 and the converter $\mathcal{C}$ presented in Figure 5. The key features of the composition are as follows.

(i) $S_{\mathcal{C}//1\|2} = \{(c_0,(s_0,t_0)),(c_1,(s_1,t_0)),(c_2,(s_0,t_1))\}$ with $(c_0,(s_0,t_0))$ as the initial state.

(ii) For any state $(c,s) \in S_{\mathcal{C}//1\|2}, c, s) \in \mathcal{B}$.

(iii) For any state $(c,s) \in S_{\mathcal{C}//1\|2}, L_{\mathcal{C}//(1\|2)}((c,s)) = L_{1\|2}(s)$.

(iv) Each transition of any state (c,s) in the lock-step is a result of individual dual transitions of c and s. For example, the transition $(c_0,(s_0,t_0)) \xrightarrow{\overline{req},T} (c_0,(s_1,t_0))$ is possible only because the transition $c_0 \xrightarrow{req,T} c_1$ is dual to the transition $(s_0,t_0) \xrightarrow{(\overline{req},T)} (s_1,t_0)$.

It is important to note that the lock-step composition operator // is different from parallel composition operator $\|$ (Definition 2). // provides state-based control to a converter over participating protocols whereas $\|$ describes all possible behaviors of the interaction between two given protocols.

The converter presented in Figure 5 can drive the handshake-serial protocol pair to satisfy the system-level properties given in Section 5, or $\mathcal{C}//(P_1\|P_2) \models \varphi_1 \wedge \cdots \wedge \varphi_3$. The next section details the automatic algorithm that is used to generate the above converter for the handshake-serial protocol pair.

The resulting system

The lock-step composition of a given converter and a composition of protocols is a *closed* system. The protocols read all inputs from the converter whereas the converter reads all its inputs from the protocols. Once composed with a converter, the system does not interact with the external environment.

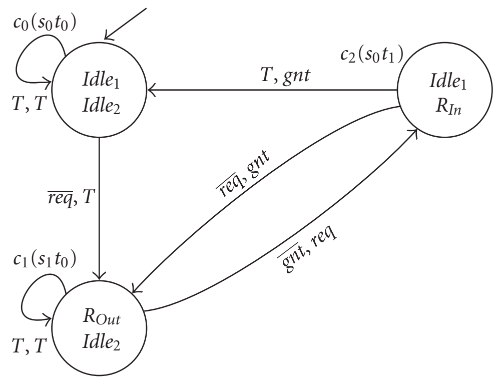

FIGURE 6: The lock-step composition of the converter $\mathcal{C}$ and the handshake-serial protocol pair.

7. CONVERTER GENERATION USING TABLEAU-BASED MODEL CHECKING

7.1. Overview

The proposed algorithm attempts to automatically generate a converter from a given pair of protocols and a set Ψ of ACTL properties describing the system-level behavior of their interaction. Given protocols P_1 and P_2, and a set of ACTL properties Ψ_0, the converter generation problem is formalized as follows.

$$\overset{?}{\exists}\ \mathcal{C} : \forall \varphi \in \Psi_0 : \mathcal{C}//(P_1\|P_2) \models \varphi. \qquad (5)$$

In other words, is there a converter $\mathcal{C}$ for the given protocols P_1 and P_2 such that in the presence of $\mathcal{C}$, the protocols satisfy all the properties contained in Ψ_0?

The proposed approach is based on the local module checking algorithm presented in [3] with nontrivial extensions for use in the convertibility verification domain. Convertibility verification using ACTL specifications is carried out using tableau construction. The conversion methodology can be summarized as follows.

(1) Identify the protocols of given IPs and extract their KS representation.

(2) Describe system-level properties using the temporal logic ACTL.

(3) Employ tableau construction to generate (if possible) a successful tableau given the inputs identified in steps 1 and 2.

(4) If no successful tableau can be generated, return to steps 1 or 2 (or both) to modify inputs (weaken ACTL properties or use modified protocols), then repeat step 3.

(5) If a successful tableau is generated in step 3, a converter is extracted automatically.

The convertibility verification algorithm is broken into two major parts: tableau construction and converter extraction. Although both these steps are carried out simultaneously, they are provided separately to aid readability.

7.2. *Tableau construction*

7.2.1. *Inputs*

The tableau construction algorithm takes the following two inputs into consideration: $P_1 \| P_2$—the composition of participating protocols, and Ψ_0—the set of `ACTL` properties to be satisfied by the interacting protocols.

7.2.2. *Data structure and initialization*

The proposed algorithm is based on tableau construction where a proof structure, called a tableau, is constructed. A tableau is essentially a table of assertions. It is structured in a top-down manner such that an assertion, called a goal, that appears directly above some assertions, called subgoals, is true only when the subgoals are true. The initial goal (the top-most assertion) for our algorithm is $c_0//s_0 \models \Psi_0$ which requires the existence of a converter state c_0 that can guide the initial state s_0 of the protocols to satisfy Ψ. This goal is then successively resolved into subgoals using *tableau rules* (to be described later).

In our setting, like in [18], a tableau is represented as an acyclic directed graph where goals are represented as *nodes* and the edges represent the top-down hierarchy between a goal and its subgoals. A tableau is defined as follows.

Definition 7 (tableau). Given $P_1 \| P_2$, and a set of `ACTL` properties Ψ_0, a tableau *Tab* is a labelled acyclic graph $\langle N, n_0, L\rangle$, where

(i) N is a finite set of nodes of the tableau. Each node $n \in N$ corresponds to a unique state $s \in S_{1\|2}$ and a unique state c in the converter (to be generated) and a set of formula Ψ where each formula $\varphi \in \Psi$ is a subformula of some formula in Ψ_0;

(ii) n_0 is the *root node* of the tableau, or the initial goal, which corresponds to the initial state $s_{0_{1\|2}}$ of $P_1 \| P_2$, the initial state c_0 of the converter to be generated, and the set of formulas Ψ_0;

(iii) $L \subseteq N \times N$ is the set of all links (edges) of the tableau.

Each node, that corresponds to states c and s of the converters and protocols, respectively, and a set of formulas Ψ, represents the *assertion* $c//s \models \Psi$. For example, the root node n_0, that corresponds to the states c_0 and s_0 (initial states of the converter and the protocols, resp.) and the set of formulas Ψ_0, represents the top-assertion $c_0//s_0 \models \Psi_0$.

A node n may have one or more *children* (nodes to which it has outgoing edges). Children nodes represent the subassertions that must be met for the assertion represented by n is satisfied. Nodes with one or more children are called *internal nodes*. Nodes with no children are called *leafnodes*. A leafnode could be a `TRUE_NODE` (represented by •), or a `FALSE_NODE`.

A node is *successful* when the assertion it represents is found to be true. A `TRUE_NODE` is implicitly successful as it represents the assertion $c//s \models \varnothing$. A `FALSE_NODE` is implicitly unsuccessful because it represents the assertion $c//s \models ff$. An internal node is successful when all its children nodes are successful. Finally, a tableau is successful when its root node is successful.

The aim of the proposed algorithm is to generate a successful tableau for the inputs $P_1 \| P_2$ and Ψ_0. During the initialization phase, only the root node n_0 of the tableau is created.

7.2.3. *Tableau construction*

After the tableau is initialized, the root node is processed using the tableau rules for converter generation presented in the following equation:

$$\begin{gathered}
\texttt{emp}\frac{c//s \models \{\}}{\bullet} \qquad \texttt{prop}\ \frac{c//s \models [\{p\} \cup \Psi]}{c//s \models \Psi}, \quad p \in L(s) \\
\wedge\ \frac{c//s \models [\{\varphi_1 \wedge \varphi_2\} \cup \Psi]}{c//s \models [\{\varphi_1, \varphi_2\} \cup \Psi]}, \\
\vee_1 \frac{c//s \models [\{\varphi_1 \vee \varphi_2\} \cup \Psi]}{c//s \models [\{\varphi_1\} \cup \Psi]}, \qquad \vee_2 \frac{c//s \models [\{\varphi_1 \vee \varphi_2\} \cup \Psi]}{c//s \models [\{\varphi_2\} \cup \Psi]}, \\
\texttt{unr}_{au} \frac{c//s \models [\{\texttt{A}(\varphi\, \texttt{U}\, \psi)\} \cup \Psi]}{c//s \models [(\psi \vee (\varphi \wedge \texttt{AXA}(\varphi\, \texttt{U}\, \psi))) \cup \Psi]}, \\
\texttt{unr}_{ag} \frac{c//s \models [\{\texttt{AG}\varphi\} \cup \Psi]}{c//s \models [(\varphi \wedge \texttt{AXAG}\varphi) \cup \Psi]}, \\
\texttt{unr}_{s} \frac{c//s \models \Psi}{\exists \pi \subseteq \Pi \cdot (\forall \sigma \in \pi \cdot c_\sigma//s_\sigma \models \Psi_{\texttt{AX}})} \\
\times \begin{cases} \Psi_{\texttt{AX}} = \{\varphi_k \mid \texttt{AX}\varphi_k \in \Psi\}, \\ \Pi = \{\sigma \mid (s \xrightarrow{\sigma} s_\sigma) \wedge (c \xrightarrow{\sigma} c_\sigma) \wedge D(\sigma, \sigma')\}. \end{cases}
\end{gathered} \tag{6}$$

The tableau rules are of the following form:

$$\frac{c//s \models \Psi}{c_1//s_1 \models \Psi_1 \cdots c_n//s_n \models \Psi_n}. \tag{7}$$

In the above, c, s, and Ψ are the constituent elements of the given node n, where Ψ is the set of formulas to be satisfied by s when it is guided (in lock-step fashion) by the converter c. s is a state in $P_1 \| P_2$ and $s_1, s_2, \ldots, s_n$ are some successor states of s, while $c_1, c_2, \ldots, c_n$ are the states of the converter to be generated. Similarly, Ψ is the set of formulas to be satisfied by s whereas $\Psi_1, \Psi_2, \ldots, \Psi_n$ are some derivatives of Ψ. The numerator represents the proof obligation (goal) that s in the presence of c must satisfy Ψ. To realize the proof, the denominator obligations (subgoals) must be satisfied.

The construction proceeds by matching the current tableau node (initially the root node) with the numerator of a tableau rule and obtaining the denominator which constitutes the next set of tableau nodes. Whenever a tableau rule is applied to a node (goal) to create new nodes (subgoals), the node has an edge to each such newly created node. Equation (6) presents the tableau rules for convertibility verification using `ACTL` specifications.

The rule `emp` corresponds to the case when there is no obligation to be satisfied by the current state of the protocols; any converter is possible in this case, that is, the converter allows all possible behavior of the parallel composition at state s. As this scenario describes a TRUE_NODE ($\bullet$), which is implicitly successful, no further processing is possible.

The prop rule states that a converter is synthesizable only when the proposition is contained in the labels of the parallel composition state s; otherwise there exists no converter. Once the propositional obligation is met, the subsequent obligation is to satisfy the rest of the formulas in the set Ψ. For example, if Ψ contains the formulas p, q, and r, and if the p is a proposition, the rule checks if the parallel composition state s is labelled by p. If this is indeed the case, the denominator then requires that the formulas q, and r are satisfied by the state.

The $\wedge$-rule states that the satisfaction of the conjunctive formula depends on the satisfaction of each of the conjuncts. The $\vee$-rules are the duals of $\wedge$-rule. The rule $\mathtt{unr}_{au}$ depends on the semantics of the temporal operator AU. A state is said to satisfy $\mathtt{A}(\varphi \,\mathtt{U}\, \psi)$ if and only if it either satisfies ψ or satisfies φ and evolves to new states each of which satisfies $\mathtt{A}(\varphi \,\mathtt{U}\, \psi)$. Similarly, $\mathtt{AG}\varphi$ is satisfied by states which satisfy φ and whose all next states satisfy $\mathtt{AG}\varphi$ (Rule $\mathtt{unr}_{ag}$).

Finally, $\mathtt{unr}_s$ is applied when the formula set in the numerator Ψ consists formulas of the form $\mathtt{AX}\varphi$ only. Satisfaction of these formulas demands that all successor states of the $c//s$ must satisfy every φ where $\mathtt{AX}\varphi \in \Psi$, that is, $c//s$ satisfies all elements of $\Psi_{\mathtt{AX}}$. A converter controls the parallel composition through implicit disabling induced by this rule. The unrestricted behavior of the protocols (where c allows all the transitions from s) may not be able to satisfy this obligation as one or more successors may fail to satisfy the commitments passed to them. However, we can check if a *subset* of the successors satisfies these commitments. If a subset of successors satisfies these future commitments, the converter can disable all other transitions of s that lead to states that are not contained in this subset.

In order to identify the subset of successors that satisfies all future commitments of the current state, we pass these commitments to each successor of the current state. For k possible successors, we need to perform k passes of this step. During each step, a new successor is chosen and the future commitments are passed to it. If it returns success, it is added to the subset. Once all successors have been checked, we return success if the subset is nonempty, and if it satisfies the well-formedness conditions described earlier in Section 6.2. In case the subset is empty, we note that there is no successor of the state s that can fulfil its future commitments, and we return failure (an unsuccessful tableau).

7.2.4. Termination: finitizing the tableau

It is important to note that the resulting tableau can be of infinite depth as each recursive formula expression AU or AG can be unfolded infinitely many times.

This problem due to the unbounded unfolding of the formula expressions can be addressed using the fixed-point semantics of the formulas $\mathtt{AG}\varphi$ and $\mathtt{A}(\varphi \,\mathtt{U}\, \psi)$. The former is a greatest fixed-point formula while the later is a least fixed-point formula,

$$\begin{aligned} \mathtt{AG}\varphi &\equiv Z_{\mathtt{AG}} =_\nu \varphi \wedge \mathtt{AX}Z_{\mathtt{AG}}, \\ \mathtt{A}(\varphi \,\mathtt{U}\, \psi) &\equiv Z_{\mathtt{AU}} =_\mu \psi \vee (\varphi \wedge \mathtt{AX}Z_{\mathtt{AU}}). \end{aligned} \tag{8}$$

The greatest (least) solution for $Z_{\mathtt{AG}}$ ($Z_{\mathtt{AU}}$) is the semantics of $\mathtt{AG}(\varphi)$. It can be shown (details are omitted) that satisfaction of the greatest fixed-point formula is realized via loops in the model. Least fixed-point formulas require that the paths satisfying the formulas must have finite length. For these formulas, if a tableau node is revisited, then it can be inferred that the LFP formula is not satisfied. As such, if a tableau node $c'//s \vDash \Psi$ is visited and there exists a prior node $c//s \vDash \Psi$ (the same tuple s paired with the same Ψ is seen in a tableau path), we check whether there exists a least fixed-point formula AU in Ψ. If such a formula is present, the tableau path is said to have resulted in an unsuccessful path (FALSE_NODE is returned). Otherwise, the tableau path is successfully terminated by equating c' with c (a loop in the converter is generated), and TRUE_NODE is returned.

We now look at how the proposed tableau construction algorithm is implemented.

7.3. Converter generation algorithm

Algorithm 1 shows the recursive converter generation procedure. Given a state s of the parallel composition, a set of subformulas FS, and a history of all previously visited tableau nodes H (all nodes visited on the path from the root node to the current node) used for finitizing the tableau (as discussed in Section 7.2), the algorithm first checks if there are no commitments to be met. In that case, it returns success ($\bullet$), otherwise it creates a new node with respect to the state s and the set of formulas FS. It then checks if a node with the same elements (state and formulas) has been visited before. If such a node is found, and FS contains an AU formula, the algorithm returns failure (FALSE_NODE). Otherwise it returns success (see notes on finitizing the tableau in Section 7.2) that results in a loop in the converter. If no matching node is found, the current node is added to the set of visited nodes. We remove a formula F from FS. Depending on the type of the formula, the corresponding tableau rule is applied by calling Algorithm 1 recursively. If the recursive call returns a non-FALSE_NODE, we return success (by adding the node returned by the recursive call as a child of the current node). Consider for example the handling of a disjunction $\varphi \vee \psi$. The algorithm first checks if the state satisfies φ along with any other commitments (subformulas) left after F was removed. If a successful tableau can be generated, the node returns success. Otherwise, we check if the state satisfies ψ along with the remaining subformulas and returns success if a successful tableau can be generated. If however neither tests returns success, the node returns failure.

If FS contains only future commitment (AX formulas), the algorithm proceeds as follows. Firstly, the node is marked as an X_NODE. An X_NODE refers to an internal node in the tableau that has been extended using the $\mathtt{unr}_s$ rule (see (6)). The future (AX) commitments are passed to each successor of s. If success is returned, we create a link from the current node to this newly created node (corresponding to the selected successor). If failure is returned, we check if the disabling of a transition to this successor would result in the breach of a rule of the converter refinement relation

```
(1) if FS = ∅ then
(2)     return TRUE_NODE
(3) end if
(4) curr = createNode(s, FS);
(5) if anc ∈ H = curr then
(6)     if FS contains AU formulas then
(7)         return FALSE_NODE
(8)     else
(9)         Remember link between curr and anc
(10)        curr.addChild(TRUE_NODE)
(11)        return curr
(12)    end if
(13) end if
(14) H_1 = H ∪ {curr};
(15) if FS contains a formula F which is not of type AX then
(16)    FS_1 := FS − F, Node ret := FALSE_NODE
(17)    if F = TRUE then
(18)        ret := isConv (s, FS_1, H_1)
(19)    else if F = $p$ ($p \in AP$) then
(20)        if $p$ is satisfied in s then
(21)            ret := isConv (s, FS_1, H_1)
(22)        end if
(23)    else if F = $\neg p$ ($p \in AP$) then
(24)        if $p$ is not satisfied in s then
(25)            ret := isConv (s, FS_1, H_1)
(26)        end if
(27)    else if F = $\varphi \wedge \psi$ then
(28)        ret := isConv (s, FS_1 ∪ {$\varphi, \psi$}, H_1)
(29)    else if F = $\varphi \vee \psi$ then
(30)        ret := isConv (s, FS_1 ∪ {$\varphi$}, H_1)
(31)        if ret = FALSE_NODE then
(32)            ret := isConv (s, FS_1 ∪ {$\psi$}, H_1)
(33)        end if
(34)    else if F = AG$\varphi$ then
(35)        ret := isConv (s, FS_1 ∪ {$\varphi \wedge$ AXAG$\varphi$}, H_1)
(36)    else if F = A($\varphi$ U $\psi$) then
(37)        ret := isConv (s, FS_1∪
                    {$\psi \vee (\varphi \wedge$ AXA($\varphi$ U $\psi$))}, H_1)
(38)    end if
(39)    if ret ! = FALSE_NODE then
(40)        curr.addChild(ret)
(41)    end if
(42)    return ret
(43) end if
(44) curr.type := X_NODE
(45) FS_AX = {$\varphi$ | AX$\varphi$ ∈ FS}
(46) for each successor s′ of s do
(47)    if (N := isConv (s′, FS_AX, H_1)) ≠ FALSE_NODE then
(48)        curr.addChild(N)
(49)    else if Transition to s′ can not be disabled (As
        disabling it would violate the conversion refinement
        rules) then
(50)        return FALSE_NODE
(51)    end if
(52) end for
(53) return curr
```

Algorithm 1: NODE isConv (s, FS, E).

```
(1) Create new map MAP
(2) Create new map PURE_MAP
(3) initialState = extractState($t$ .rootnode);
(4) return initialState
```

Algorithm 2: STATE extractConverter(Tableau t).

```
(1) if NODE is present in MAP then
(2)     return map.get(NODE)
(3) else if NODE is an internal node then
(4)     MAP.put(NODE, extract(NODE.child))
(5)     return MAP.get(NODE)
(6) else if NODE is TRUE_NODE then
(7)     if NODE is related to an ancestor anc then
(8)         {see line 9 of isConv}
(9)         MAP.put(NODE,getPureState(anc.s)
(10)        return MAP.get(NODE)
(11)    else
(12)        MAP.put(NODE,getPureState(NODE.s)
(13)        return MAP.get(NODE)
(14)    end if
(15) else if NODE is of type X_NODE then
(16)    create new converter state $c$
(17)    MAP.put(NODE, $c$)
(18)    for each linked NODE′ of NODE do
(19)        State $c'$ = extract(NODE′)
(20)        add transition $c \xrightarrow{(\sigma')} c'$ where
            NODE.s $\xrightarrow{\sigma}$ NODE′.s and $\mathcal{D}(\sigma, \sigma')$.
(21)    end for
(22)    return MAP.get(NODE)
(23) end if
```

Algorithm 3: NODE extract(NODE).

(see Definition 4). For example, if the current successor is reached via an output transition, the transition to it cannot be disabled. In such cases when a successor does not meet future commitments and we cannot disable transitions to them, we return failure. The algorithm returns success when one or more successors of s satisfy the future commitments (and the converter refinement relation holds if a converter state enables these transitions in s).

7.4. Converter extraction

If a successful tableau is constructed, the converter is extracted by traversing the nodes of the tableau as shown in extractConverter algorithm (Algorithm 2). Firstly, it creates a map MAP. MAP is essentially a lookup table that relates nodes in the tableau (keys) to states in the converter being generated (values). This map is initially empty. We then pass the tableau's root node to the recursive procedure extract (Algorithm 3) which processes a given node as follows.

(1) If NODE is already present in MAP as a key, return the converter state associated with it.

```
(1)  if s is present in PURE_MAP then
(2)      return PURE_MAP.get(s)
(3)  else
(4)      create new converter state $c$
(5)      PURE_MAP.put(s,$c$)
(6)      for each successor s′ of s do
(7)          State $c'$ = getPureState(s′)
(8)          add transition $c \xrightarrow{\mathcal{D}\sigma} c'$ where $\sigma$ is the label of
             the transition from s to s
(9)      end forreturn $c$
(10) end if
```

Algorithm 4: NODE getPureState(s).

(2) If NODE is an internal node, the converter state corresponding to this node is the converter state extracted with respect to its child. An internal node always has one child as it is expanded by the application of a tableau rule other than $\mathtt{unr}_s$.

(3) If NODE is of type TRUE_NODE, and it is related to an ancestor node (see line 9 of isConv), the converter state corresponding to the node is the same as the converter state corresponding to its linked ancestor.

(4) If NODEis of type TRUE_NODE but is not related to any ancestors, the converter allows all transitions in $P_1\|P_2$ from this state onwards. We return a converter state that allows all transitions in the state corresponding to NODE and any of its successors (see Algorithm 4).

(5) If NODE is of type X_NODE, we create a new converter state corresponding to the node. The created state c contains transitions to each state corresponding to each linked child of the X_NODE.

Illustration

This section presents the steps involved in the tableau construction and converter extraction for the handshake-serial example in Figure 1.

The tableau construction for the handshake-serial example starts at the construction of the *root* node of the tableau corresponding to the initial state (s_0, t_0) of $P_1\|P_2$ (see Figure 2), and the set FS (see Section 5) of system-level properties to be satisfied. The root node is created when these arguments are passed to the recursive algorithm isConv.

The processing of the root node is shown in Table 1. The algorithm proceeds as follows. Given the root node (node 0), the algorithm removes one formula $\mathtt{F} = \mathtt{AG}(Idle_1 \wedge Idle_2 \Rightarrow \mathtt{AX}(R_{Out} \vee \neg R_{In}))$ from the set NODE.FS. Then, F is broken down into simpler commitments $(Idle_1 \wedge Idle_2 \Rightarrow \mathtt{AX}(R_{Out} \vee \neg R_{In})) \wedge \mathtt{AXAG}(Idle_1 \wedge Idle_2 \Rightarrow \mathtt{AX}(R_{Out} \vee \neg R_{In}))$. These simpler commitments are then reinserted into the set FS of a new node (node 1 as shown in Table 1) along with all remaining commitments in node 1.FS, and a recursive call

Table 1: Nodes with the attributes $s = (s_0, t_0)$, $c = c_0$.

Node	Ψ
0	$\{\mathtt{AG}(Idle_1 \wedge Idle_2 \Rightarrow \mathtt{AX}(R_{Out} \vee \neg R_{In}))$, $\mathtt{AG}(R_{In} \wedge R_{Out} \Rightarrow \mathtt{AX}(Idle_1 \vee \neg Idle_2))$, $\mathtt{AG}(R_{Out} \wedge Idle_1 \Rightarrow \mathtt{AX}(\neg Idle_1 \vee \neg Idle_2))$, $\mathtt{AG}(Idle_1 \wedge R_{In} \Rightarrow \mathtt{AX}(\neg R_{Out} \vee \neg R_{In}))\}$
0	$\{\mathtt{AG}\psi_1, \mathtt{AG}\psi_2, \mathtt{AG}\psi_3, \mathtt{AG}\psi_4\}$
1	$\{\psi_1 \wedge \mathtt{AG}\psi_1, \mathtt{AG}\psi_2, \mathtt{AG}\psi_3, \mathtt{AG}\psi_4\}$
2	$\{\psi_1, \mathtt{AG}\psi_1, \mathtt{AG}\psi_2, \mathtt{AG}\psi_3, \mathtt{AG}\psi_4\}$
⋮	
9	$\{\mathtt{AX}(R_{Out} \vee \neg R_{In}), \mathtt{AXAG}\psi_1, \mathtt{AXAG}\psi_2, \mathtt{AXAG}\psi_3, \mathtt{AXAG}\psi_4\}$

to isConv is made. The newly created node is added as a child node of NODE if it returns success. Whenever F is a propositional formula, it can be checked against the labels of the state (s_0, t_0). The process of removing one formula and reinserting its subformulas stops when NODE contains no formulas or only AX-type formulas (node 9).

When only AX formulas are left (node 9 in Table 1), the node is labelled as an X_NODE (line 44). At this stage, for every successor of (s_0, t_0), the algorithm makes a recursive call to itself. During each call, the arguments to isConv are a unique successor of (s_0, t_0), and all the commitments to be satisfied by the successor (if it is enabled by the converter). For calls that return success, their corresponding nodes are added as children of node 9.

The above process continues until the recursive call made by the root node returns. For the given example, this call returns success, which means that a successful tableau has been constructed. Using the converter extraction algorithm described earlier, the tableau yields the converter presented in Figure 5.

7.5. Complexity

The tableau considers all possible subformulas of the given set of desired properties. In the worst case, each subformula can be paired with every possible state in the protocol pair. The complexity of the tableau construction is therefore $O(|S| \times 2^{|\varphi|})$, where S is the number of states in the protocol pairs and $|\varphi|$ is the size of the formula(s) used for conversion.

The complexity differs from model checking [18, 19]. The complexity for both CTL and ACTL model checking algorithms is $O(|S| \times |\varphi|)$, where φ is the size of the given formula. The reason for this difference arises from the handling of conjunctions of disjunctive formulas in model checking and our algorithm. In model checking, if a state is expected to satisfy a formula $(a \vee b) \wedge (c \vee d) \wedge (e \vee f)$, the algorithm first computes the states that satisfy the subformulas a, b, c, d, e, f, $(a \vee b)$, $(c \vee d)$, and $(e \vee f)$ of the given formula before computing the states that satisfy the given formula. Hence, each subformula is only checked once. However, in our approach, the increase in complexity occurs due to the fact that all possible subformulas (of the given set of formulas) are considered for every node.

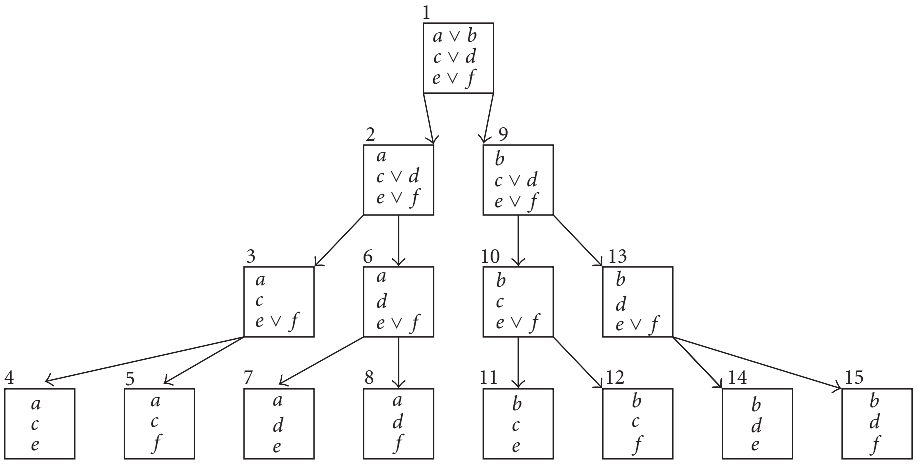

Figure 7: Handling of disjunctive formulas.

For example, Figure 7 shows a node which requires the conjunction of three disjunctive formulas to hold on a state in the protocols. As there is no particular order in which formulas are picked from the node's commitment set FS (containing the three formulas), we consider the order given in Figure 7. Formulas $a, \dots, f$ could be any type of ACTL formulas. Each node is numbered in the order visited. Initially, the disjunction $a \vee b$ is removed from ψ and a child node checking only a and the other formulas is created (node 2). In node 2, the formula $c \vee d$ is chosen which results in the creation of node 3 which checks a, c, and $e \vee f$. Similarly, node 3 leads to node 4 which checks a, c, and e. At this stage, as all remaining formulas are propositional, no further breaking up can be carried out. Assuming that a, c, and e are propositions, it is possible for the algorithm to check their satisfaction linearly. In case any one is not satisfied, we return failure which results in the creation of node 5 which checks a, c, and f. Given the order in Figure 7 and assuming that none of the propositional formulas are satisfied by the corresponding nodes, the algorithm terminates after it has checked all possible combination of disjuncts. It can be seen that the number of nodes created is exponential to the number of disjunctive formulas.

7.6. Soundness and completeness

The following theorem follows from the above discussion. The proof of the theorem is provided in the appendix.

Theorem 1 (sound and complete). *There exists a converter $\mathcal{C}$ to control two given KS P_1 and P_2, such that the resulting system $\mathcal{C}//(P_1 \| P_2)$ satisfies a set* FS *of* ACTL *formulas, if and only if* isConv $(s_{0_{1\|2}}, \text{FS}, \varnothing)$ *does not return* FALSE_NODE.

8. RESULTS

A convertibility verification tool using tableau construction has been implemented using Java. The implementation takes as input the Kripke structure representation of two protocols P_1 and P_2 and a set Ψ of ACTL properties from the user. The Kripke structure representation of a protocol is extracted automatically from a given NuSMV description of an IP. For this purpose, a modified version of the NuSMV model checking tool is employed [20]. Given these inputs, the algorithm constructs a tableau after computing the parallel composition $P_1 \| P_2$. If a successful tableau is created, a converter, represented as a Kripke structure, is automatically generated. This converter is guaranteed to bridge mismatches between the two protocols.

Table 2 shows the results obtained from the protocol conversion of various examples. Problems 1–6 are well-known protocol mismatch problems explored by earlier works such as [4, 12]. Problems 7–9 are derived from well-known NuSMV examples [20]. These examples were created by introducing commonly-encountered control mismatches, such as incorrect signal exchange sequences and lack of mutual exclusion, into the systems. Problems 10–14 are SoC examples that were chosen to show the applicability of our approach for SoC design. Each of these SoC problems modeled control mismatches between the AMBA bus and a peripheral. Different version of AMBA, namely the Advanced High-performance Bus (AHB), Advanced System Bus (ASB), and Advances Peripheral Bus (APB) were used. Problems 12–15 involve conversion between more than 2 IPs that is achieved by extending the proposed framework. The proposed extension to handle "n"-protocols is presented in [21]. This extension involves the formulation of composition rules for multiple Kripke structures, and new constraints for converters (conversion refinement relation) to extend their control to multiple protocols. The conversion algorithm is then extended such that its input KSs P_1 and P_2 are parallel compositions of multiple protocols.

The first three columns of Table 2 contain the ID, description, and size (number of states) of the participating protocols. The intuitive description of the ACTL properties used for each problem was shown in the fourth column. For most of the problems presented in Table 2, our algorithm was

Table 2: Implementation results.

No.	$P_1(\lvert S_{P_1}\rvert)$	$P_1(\lvert S_{P_2}\rvert)$	ACTL properties
1	Master (3)	Slave (3)	Correct sequencing of signal emission, (no read before a request, no duplicate requests)
2	ABP sender (6)	NP receiver (4)	Correct exchange of control signals
3	ABP receiver (8)	NP sender (3)	Correct exchange of control signals
4	Poll-end receiver (2)	Ack-nack sender (3)	Correct, loss-free reception of packets by the receiver
5	Handshake (2)	Serial (2)	Consistent sequencing [12]
6	Multiwrite master (3)	Single-read slave (4)	One read per write of master, master always writes before the slave reads
7	Mutex process (3)	Mutex process (3)	Mutual exclusion
8	MCP missionaries (6)	MCP cannibals (5)	Correct error-free sequencing of two processes
9	4-bit ABP sender (430)	Modified receiver (384)	Correction of the interaction such that desired behavior is achieved
10	AMBA AHB, ASB, or APB arbiter (3)	Producer master (4)	Correct arbitration (producer always gets access)
11	AMBA AHB, ASB or APB Arbiter (3)	Consumer master (4)	Correct arbitration (consumer always gets access)
12	AMBA AHB arbiter (3)	Producer and consumer masters (16)	Arbitration (both always eventually get to execute)
13	AMBA AHB arbiter (3)	Producer, consumer masters, and memory slave (144)	Correct sequencing and arbitration (packets from producer to memory to consumer)
14	AMBA AHB arbiter (3)	3 masters and 2 slaves (5184)	Correct sequencing and arbitration
15	7-process system (2187)	7-process system (2187)	Mutual exclusion and no-starvation

able to generate a converter to control the participation to satisfy the given ACTL properties.

There was one case (Problem 11) when the algorithm failed to generate a converter. It failed because of the inability of the AMBA APB to allow burst transfers (multiple operations per activation of a master). In cases where the algorithm fails (an unsuccessful tableau is returned), it is required that the inputs (IPs and/or specifications) be modified manually in order to bridge mismatches between these protocols. To carry out this manual modification, the unsuccessful tableau returned by the algorithm can be used (as a counterexample) to find the exact reason (state, subformula) for failure.

The significance of these results is explained as follows. For problems 1–6, the use of ACTL specifications resulted in converters that were similar in size than those generated when automata-based specifications were used (in [4, 12]). This shows that ACTL is powerful enough to describe most commonly-encountered specifications used for conversion. ACTL also has the additional benefit of being succinct and more intuitive to write than automata-based properties for many problems. For example, to guide a 14-process system to have mutual exclusion (Problem 15), the resulting automaton that describes such interaction will be very complex and difficult to write while one can write simple ACTL properties that describe such interaction. In addition, a user may provide multiple properties and IPs during conversion using the proposed algorithm, a feature that is not offered by earlier works [4, 6]. Furthermore, our approach is capable of handling bidirectional communication between IPs, which cannot be handled using approaches such as [4, 12].

Finally, specifications in our setting are described using temporal logic properties whereas other approaches like [4, 12] use automata-based specifications. The use of temporal logic allows us to write complex specifications succinctly by breaking them down into multiple properties. Furthermore, in addition to mismatch resolution, the conversion algorithm can be used to enforce additional properties in the converted system.

9. CONCLUSIONS

Protocol conversion to resolve mismatches between IPs is an active research area. A number of solutions have been proposed. Some approaches require significant user input and guidance, 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 the first temporal logic-based approach to protocol conversion that can automatically generate a converter given the protocols of mismatched IPs and specifications described as ACTL formulas. The proposed algorithm uses tableau generation and attempts to construct a tableau given the above inputs. If the algorithm succeeds (a successful tableau is constructed), a converter is automatically generated. This converter is guaranteed to control the given IPs such that the required ACTL properties are satisfied.

The approach is shown to be sound and complete and is capable of handling many common mismatches that existing conversion techniques can address. In addition, ACTL formulas are easier to write for complex specifications

```
(1) process FS till no conjunctions, AG and AU formulas
      remain.
(2) if FS contains no disjunctions then
(3)     return FS
(4) end if
(5) Pick a disjunction F = $\phi \vee \varphi$ from FS
(6) return extractSets(FS-F + $\phi$) ∪ extractSets(FS-F + $\varphi$)
```

Algorithm 5: FormulaSets extractSets (FS).

as compared to automata-based specifications. The proposed technique can handle bidirectional communication between protocols and can also ensure that the converted system satisfies additional temporal logic constraints. All these features are not provided by any existing approaches to protocol conversion.

The future direction for this work involves the extension of the approach to handle data and clock mismatches between IPs and to provide a correct-by-construction design technique for SoCs using protocol conversion. Another area of exploration is the automatic derivation of ACTL properties to describe the behavior of two or more communicating IPs.

APPENDIX

In order to prove Theorem 1, we first prove the following lemmas.

Lemma 1 (termination). *A call to* isConv $(s'$,H$'$,E$')$ *always terminates.*

Proof. The proof of this lemma follows from an analysis of recursion in isConv. The following observations can be made.

(i) Whenever isConv is called, if a node with the same attributes (s$'$ and FS$'$) exists in the history set H, then the call returns without further recursion. This observation can be verified by checking that the newly created node curr is only added to the tableau in line 13, after the history set H is checked for the presence of a node with the same attributes as curr. If the history set does contain such a node, we either return FALSE_NODE or curr but do not call isConv any further.

(ii) Whenever a recursive call to isConv is made, and no node with the same attributes to curr is found in the history set, the node created in the current call is always added to the history set H$'$ before a recursive call is made (see line 13).

(iii) The number of states in $P_1\|P_2$ and the maximum number of valuations for the sets FS$'$ are finite in the following manner.

(a) $P_1\|P_2$ has a finite number of states by definition.

(b) FS$'$ cannot have a valuation outside the $2^{|\mathrm{FS}|}$ possible subsets of subformulas of FS.

(iv) As each node corresponds to a combination of the following: a state of $P_1\|P_2$, and a subset FS$'$ of subformulas of FS, both of which have finite valuations, the number of nodes that can be created in isConv is finite.

As described above, the number of nodes that can be created by isConv is finite. Furthermore, isConv stops recursing if a newly created node is already present in the history set, preventing duplicating nodes along all paths. Hence, it is proved that a call to isConv always terminates. □

Lemma 2 (sufficient condition). *If* isConv$(s_{0_{1\|2}}$,FS,$\varnothing)$ *returns a non-*FALSE_NODE*, there exists a correct converter* $\mathfrak{C}$ *such that* $\mathfrak{C}//P_1\|P_2 \models$ FS.

Proof. In order to prove the above lemma, we assume that we have been given a tableau with root node n_0 corresponding to the initial state $s_{0_{1\|2}}$, and we generate a converter from the tableau and show that the converter helps satisfy the property.

We first note the following characteristics of a successful tableau return by isConv.

(1) A node can never have a false node as a child. A leafnode results when the algorithm finds the same node in the history set. Hence, no further recursion is done and the leafnode itself has no children. An internal node makes a recursive call to isConv, and if that call returns a false node, the calling node does not add the false node to its children list.

(2) A node is present in the tableau if and only if it returned a nonfalse node: if a call to isConv returns false node, the node is not added to the tableau. Hence, all nodes present in the tableau reachable from n_0 returned a nonfalse node.

(3) All leafnodes and true nodes have no children: these nodes are formed when there are no subformulas to satisfy (FS $=$ $\varnothing$) or when a node with the same attributes is found in history. In both these cases, no further calls to isConv are made.

(4) All internal non-X_NODEs have one child only.

(5) An X_NODE may have more than one child.

(6) All internal nodes lead to an X_NODE, true node, or a leafnode: given a set of formulas FS, the child of curr, if curr is an internal node, will have the same commitments as curr except that one of the formulas is broken into smaller commitments. If this process is repeated infinitely often, we reach a fixpoint where no formulas in curr can be broken any further. At this stage, we can only have AX formulas in FS which results in the formation of an X_NODE. If FS contains no formulas, we form a true node. It is also possible that while breaking down formulas from a parent node to a child node (where both are internal nodes), we may find that a matching node is already present in the history set H. In that case, we finitize

the tableau by returning the current node `curr` and marking it as an internal node.

(7) All leafnodes lead to an `X_NODE` before they are visited again: a leafnode points to a node in history, which is an ancestor of the leafnode. That ancestor node cannot be a leafnode because otherwise there would have been no path from the ancestor to the current leafnodes. Furthermore, starting from the ancestor node, none of its descendent nodes can be leafnodes (except the current one) because otherwise a path from that descendent to the leafnode would not have been possible. The commitments in the leafnode are the same as those of the matching ancestor. Now, the child of the ancestor must contain simpler commitments in its formula set than the ancestor. It can be seen that the commitments of an internal node are never repeated by their descendent nodes because of the successive breaking down of commitments until only `AX` commitments remain. Hence, the ancestor must eventually lead to an `X_NODE` (fix point where all formulas except `AX` commitments). It cannot come to the leafnode without an `X_NODE` in between because that would require that the ancestor and leafnodes have different attributes (formula sets) which is not possible.

Given the above observations, we now extract a converter that satisfies `FS` starting from the root node using the algorithms `extractConverter`, `extract`, and `getPureState` given earlier.

`extractConverter` creates two global variables `MAP` and `PURE_MAP` that store all converter states that are generated. It then calls the recursive algorithm `extract` and passes the root node of the successful tableau obtained by `isConv`.

In algorithm `extract`, starting from the root node n_0, we recurse until we reach an `X_NODE` or a $\bullet$. We cannot come to a leafnode before reaching an `X_NODE` from the root node as per the above observations. Furthermore, we can only reach a single `X_NODE` as all internal nodes starting from n_0 can only have one child each.

Corresponding to the `X_NODE`, we create the initial node c_0 of the converter. To c_0, we now add all states obtained by recursively calling `extractConverter` on each of the children of the `X_NODE`.

In case we reach a true node, all commitments in `FS` have been satisfied and there are no future commitments to be satisfied. Hence, we now use `getPureState` to allow the converter to enable all transitions in the parallel composition from this point onwards.

Once the above converter is extracted, we can prove that it indeed satisfies all formulas `FS` as follows. From any node `curr` in the tableau, we may eventually reach an `X_NODE` $\texttt{curr}_{ax}$ (possibly through a leafnode) or a true node. In case we reach a true node, there are no future commitments to satisfy. Furthermore, all present state commitments must be satisfied in the current node otherwise we could not have reached a true node (each present state commitment is removed from `FS` as it is satisfied by the current node).

If we instead reach an `X_NODE`, it can only contain `AX` commitments. We can see that all the present state commitments in `FS` of `curr` are satisfied by the state `curr`.state.

Now, we create a converter state c for each `X_NODE`. Furthermore, c contains one node corresponding to each child of the `X_NODE`. An `X_NODE` contains a child only if all `AX` commitments present in the present node are satisfied by the successor corresponding to the child. Each child can only be present in the tableau because it returned success. For each of this children, we create a further successor of c and repeat this process until all nodes have been processed.

We can see that each state in the converted system satisfies all its present commitments and its enabled successors satisfy all their future commitments. □

Lemma 3 (correct converters). *Any converter $\mathcal{C}$ obtained from the tableau returned by the call* `isConv` $(s_{0_{1\|2}}, \texttt{FS}, \varnothing)$ *is a correct converter.*

Proof. The proof of the above lemma follows from the proof of the previous lemma which shows that a converter state extracted from an `X_NODE` always ensures conversion refinement between itself and the state corresponding to the `X_NODE`. □

Lemma 4 (necessary condition). *If there exists a converter $\mathcal{C}$ such that $\mathcal{C}//P_1\|P_2 \models \texttt{FS}$,* `isConv` $(s_{0_{1\|2}}, \texttt{FS}, \varnothing)$ *returns a non-*`FALSE_NODE`*.*

Proof. We assume that we have been given a converter $\mathcal{C}$ under which $P_1\|P_2$ satisfies all commitments `FS`. Furthermore, we assume that `isConv` $(s_{0_{1\|2}}, \texttt{FS}, \varnothing)$ returns a `FALSE_NODE`.

We now show that above statements are contradictory.

We first process the set `FS` successively in the following manner. Each formula `F` in `FS` is processed as follows.

(i) If `F` is tt, ff, proposition, negated proposition, disjunction, or `AX` formula, it is not processed further.

(ii) If $\texttt{F} = \phi \wedge \varphi$: after processing, $\texttt{FS} = \texttt{FS} - f \cup \{\phi, \varphi\}$.

(iii) If $\texttt{F} = \texttt{AG}\phi$: after processing, $\texttt{FS} = \texttt{FS} \cup \{\phi \wedge \texttt{AXAG}\phi\}$. Again, ϕ can be processed further depending on its type whilst $\texttt{AXAG}\phi$ is an `AX` formula.

(iv) If $\texttt{F} = \texttt{A}(\phi\, \texttt{U}\, \varphi)$: after processing, $\texttt{FS} = \texttt{FS} \cup \{\varphi \vee (\phi \wedge \texttt{AXA}(\phi\, \texttt{U}\, \varphi)\}$, which is a disjunction.

Once the above processing is completed, `FS` will contain only propositions (including tt and ff), negated propositions, disjunctions, or `AX` formulas.

Using the algorithm `extractSets`, we remove all disjunctions by creating multiple copies of `FS` (Algorithm 5). Given a formula $\phi \vee \varphi$ in `FS`, we create two copies of `FS`, each containing all formulas in `FS` except the disjunction plus one of the operands of the disjunction. We process `FS` until all disjunctive formulas are removed, and a set `SetOfFormulaSets` containing multiple formula sets. Furthermore, we remove from `SetOfFormulaSets` all sets that contain ff, because such a set cannot be satisfied by any state in a Kripke structure. Furthermore, we remove from all sets in `SetOfFormulaSets` the formula *tt* because it is implicitly satisfied by all states in any given Kripke structure.

Each set in `SetOfFormulaSets` contains only propositions, negated propositions, and disjunctions. For a state in a Kripke structure to satisfy the original formula set `FS`, it must satisfy at least one of the sets contained in `SetOfFormulaSets`. The state must satisfy all propositions and negated propositions, and its successors must satisfy all `AX` formulas. A state does not satisfy `FS` if it does not satisfy any of the sets contained in `SetOfFormulaSets`.

We now show that given a formula set `FS`, a call to `isConv` returns failure (an unsuccessful tableau) only after checking all possible sets that can be contained in `SetOfFormulaSets`.

A call to `isConv` processes formulas as follows.

(i) A propositional formula $p \in$ `FS` is always checked against the labels of the given state s in the parallel composition.

(ii) A negated proposition $\neg p \in$ `FS` is always checked against the labels of the given state s in the parallel composition.

(iii) A conjunction is processed by adding both conjuncts to `FS`, which are further processed depending on their type.

(iv) `AG` formulas are replaced by conjunctions which are processed further.

(v) `AU` formulas are replaced by disjunctions.

(vi) A disjunction $\phi \vee \varphi$ is processed as follows. First, `isConv` checks whether the current node can satisfy ϕ by making a recursive call to `isConv` and passing `FS` in which the disjunction is replaced by the first disjunct ϕ. A failure (an unsuccessful tableau) is returned if an eventual subformula of ϕ cannot be satisfied. During this call, ϕ may be further broken down to subformulas propositions, negated propositions, disjunctions, and/or conjunctions which are handled depending on their respective types.

If the first recursive call returns failure, another call to `isConv` is made where `FS` is passed after the disjunction is replaced by φ. This call now checks all subformulas of φ along with other formulas in `FS`.

`isConv` returns failure when both recursive calls return failure, which happens after all different sets of commitments resulting from the disjunction have been checked. Note that during this process, it checks all possible sets of commitments resulting from the disjunction (along with other formulas in `FS`).

(vii) When `FS` contains only `AX` formulas (all present state commitments have been checked), `isConv` tries to find a subset of the set of successors of the current parallel composition state, such that each state in the subset satisfies all `AX` formulas. Note that `isConv` also ensures that the enabled subset ensures that the rules of the conversion refinement relation are satisfied. `isConv` returns failure when no conforming subset that satisfies the above constraints could be found.

Given a set `FS` and the initial state $s_{0_{1\|2}}$ of the parallel composition, the call `isConv` $(s_{0_{1\|2}}, \mathtt{FS}, \varnothing)$ results in each formula of `FS` being processed as discussed above. All different sets of formulas resulting from the presence of disjunctions are checked, including all future `AX` commitments that must be satisfied by the successors of the initial state.

For the call to `isConv` to return failure, no set of commitments present in the set of formula sets `extractSets` (`FS`) must be satisfied. In other words, there must be no possible converter states under which $s_{0_{1\|2}}$ satisfies `FS`. However, as we are given that there exists a correct converter $\mathcal{C}$ with the initial state c_0 such that $c_0 // s_{0_{1\|2}} \models$ `FS`, the statement that `isConv` may return `FALSE_NODE` cannot be true.

Hence, if there exists a converter that can guide a given parallel composition to satisfy a given set of commitments, `isConv` will never return `FALSE_NODE`. In other words, `isConv` will be always able to find a converter. □

Proof. The proof of Theorem 1 follows from the above lemmas. □

ACKNOWLEDGMENTS

The authors acknowledge Dr. Alain Girault and Professor Zoran Salcic for their help in formulating this approach. The authors would also like to thank the anonymous reviewers of SLA++p'07 and Eurasip JES (Special Issue: SLA++p'07 and SLA++P'08) for their invaluable feedback.

REFERENCES

[1] O. Kupferman and M. Y. Vardi, "Module checking [model checking of open systems]," in *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)*, pp. 75–86, New Brunswick, NJ, USA, July-August 1996.

[2] O. Kupferman, M. Y. Vardi, and P. Wolper, "Module checking," *Information and Computation*, vol. 164, no. 2, pp. 322–344, 2001.

[3] S. Basu, P. S. Roop, and R. Sinha, "Local module checking for CTL specifications," in *Proceedings of Formal Foundations of Embedded Software and IP-Based Software Architectures (FESCA '06)*, pp. 1–19, Vienna, Austria, March-April 2006.

[4] S. S. Lam, "Convertibility verification," *IEEE Transactions on Software Engineering*, vol. 14, no. 3, pp. 353–362, 1988.

[5] K. L. Calvert and S. S. Lam, "Formal methods for convertibility verification," *IEEE Journal on Selected Areas in Communication*, vol. 8, no. 1, pp. 127–142, 1990.

[6] K. Okumura, "A formal protocol conversion method," *SIGCOMM Computer Communication Review*, vol. 16, no. 3, pp. 30–37, 1986.

[7] J. C. Shu and M. T. Liu, "A synchronization model for convertibility verification," in *Proceedings of the 8th Annual Joint Conference on the IEEE Computer and Communications Societies (INFOCOM '89)*, vol. 1, pp. 276–284, Ottawa, Canada, April 1989.

[8] R. Kumar, S. S. Nelvagal, and S. I. Marcus, "Protocol conversion using supervisory control techniques," in *Proceedings of the IEEE International Symposium on Computer-Aided Control*

System Design (CACSD '96), pp. 32–37, Dearborn, Mich, USA, September 1996.

[9] G. V. Bochmann, "Deriving protocol converters for communications gateways," *IEEE Transactions on Communications*, vol. 38, no. 9, pp. 1298–1300, 1990.

[10] F. M. Burg and N. Di Iorio, "Networking of networks: interworking according to OSI," *IEEE Journal on Selected Areas in Communications*, vol. 7, no. 7, pp. 1131–1142, 1989.

[11] P. Green Jr., "Protocol conversion," *IEEE Transactions on Communications*, vol. 34, no. 3, pp. 257–268, 1986.

[12] R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli, "Convertibility verification and converter synthesis: two faces of the same coin [IP block interfaces]," in *Proceedings of IEEE International Conference on Computer-Aided Design (ICCAD '02)*, pp. 132–139, San Jose, Calif, USA, November 2002.

[13] V. D'silva, S. Ramesh, and A. Sowmya, "Bridge over troubled wrappers: automated interface synthesis," in *Proceedings of the 17th International Conference on VLSI Design (VLSID '04)*, pp. 189–194, Mumbai, India, January 2004.

[14] 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)*, pp. 731–736, San Francisco, Calif, USA, July 2006.

[15] V. D'silva, S. Ramesh, and A. Sowmya, "Synchronous protocol automata: a framework for modelling and verification of SoC communication architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '04)*, vol. 1, pp. 390–395, Washington, DC, USA, February 2004.

[16] M. Antoniotti, *Synthesis and verification of discrete controllers for robotics and manufacturing devices with temporal logic and the control-D system*, Ph.D. thesis, New York University, New York, NY, USA, 1995.

[17] S. Jiang and R. Kumar, "Supervisory control of discrete event systems with CTL* temporal logic specifications," *SIAM Journal on Control and Optimization*, vol. 44, no. 6, pp. 2079–2103, 2006.

[18] G. Bhat, R. Cleaveland, and O. Grumberg, "Efficient on-the-fly model checking for CTL," in *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS '95)*, pp. 388–397, San Diego, Calif, USA, June 1995.

[19] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[20] R. Cavada, A. Cimatt, E. Olivetti, M. Pistore, and M. Roveri, *NuSMV 2.1 User Manual*, June 2003.

[21] R. Sinha, P. S. Roop, and S. Basu, "A model checking approach to protocol conversion," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 4, pp. 81–94, 2008.