Transducer-Based Algorithmic Verification of Retransmission Protocols over Noisy Channels

Rajeev Alur, Jay Thakkar, Aditya Kanade

Transducer-based Algorithmic Verification of Retransmission Protocols over Noisy Channels

Jay Thakkar[1], Aditya Kanade[1], and Rajeev Alur[2]

[1] Indian Institute of Science
[2] University of Pennsylvania

Abstract. Unreliable communication channels are a practical reality. They add to the complexity of protocol design and verification. In this paper, we consider *noisy channels* which can corrupt messages. We present an approach to model and verify protocols which combine error detection and error control to provide reliable communication over noisy channels. We call these protocols *retransmission protocols* as they achieve reliable communication through repeated retransmissions of messages. These protocols typically use cyclic redundancy checks and sliding window protocols for error detection and control respectively. We propose models of these protocols as regular transducers operating on bit strings. Streaming string transducers provide a natural way of modeling these protocols and formalizing correctness requirements. The verification problem is posed as functional equivalence between the protocol transducer and the specification transducer. Functional equivalence checking is decidable for this class of transducers and this makes the transducer models amenable to algorithmic verification. We present case studies based on TinyOS serial communication and the HDLC retransmission protocol.

1 Introduction

Communication protocols play a foundational role in the design of distributed systems. Model checking approaches (e.g. [25]) analyze protocols for concurrency bugs. The traditional approach here is to build a finite-state model which abstracts message contents. Whether the communication channels between distributed components of a protocol deliver messages correctly or not is modeled as a nondeterministic choice. In practice, the physical channels can give rise to multiple types of faults, including message corruption, loss, and reordering. The real-world protocols, that provide reliable communication over unreliable channels, examine message contents to determine validity and semantics of messages [35]. An approach where message contents are abstracted by symbolic constants may capture semantics of such protocols only partially.

A common approach to ensure reliable communication is to combine error detection and error control mechanisms. The sender adds redundancy (checksum bits) to the messages which is used by the receiver to detect errors in the received messages. Upon receiving a corrupted message, the receiver requests the sender for retransmission of the message. We call protocols which follow this scheme as

retransmission protocols for noisy channels. These protocols typically use cyclic redundancy checks (CRC) [29] and sliding window protocols [14, 34] for error detection and control respectively. By a *noisy channel*, we mean a channel that can corrupt messages but not drop, re-order, or duplicate them. TinyOS serial communication [1], high-level data link control (HDLC) [26], and transmission control protocol (TCP) [2] are examples of widely used retransmission protocols over noisy channels.

The objective of this paper is to devise a technique to model and verify such protocols. We observe that, for accurate modeling of the protocols, we have to encode messages as sequences of bits that are exchanged over channels. (In Section 2, we show that a protocol model, in which messages are abstracted as symbolic constants, can deliver a wrong sequence of messages.) However, verification of finite-state machines communicating asynchronously over unbounded FIFO channels is known to be undecidable [12].

In this paper, we present an algorithmic technique for verification of retransmission protocols where messages, checksums, and acknowledgements are treated as *bit strings*. In our models, the message strings and the number of (re)transmission rounds can be unbounded. Our approach is based on the observation that the protocol components can be viewed as *transductions over bit strings*. As an example, consider a sender which gets a message M from its client. The sender transmits it and in return, gets an acknowledgement a from the receiver. The combined input to the sender can be modeled as a string $M\sharp a$ where $\sharp$ is the end-marker attached to the message. The semantics of the sender is that if a is 0 (a negative acknowledgement) then it should retransmit the message. This can be formalized as a transduction f defined as $f(M\sharp 1) = M\sharp$ and $f(M\sharp 0) = M\sharp M\sharp$. We give such transducer-based semantics to protocol components (sender and receiver) and obtain the protocol model by *sequential composition* of its components. Our modeling approach thus differs from the modeling of protocol components as asynchronously communicating finite-state machines [12]. The correctness requirement here is that, in spite of message corruption, the receiver delivers only correct messages to its client and in the same order as the sender's client intended. We show that the specification can also be modeled as a transducer. The verification problem is then reduced to checking the functional equivalence of the protocol and the specification transducers.

In many cases, the transductions defined by the sender, the receiver, and the specification of retransmission protocols are *regular*. Regular transducers are closed under sequential composition [16] and equivalence checking is decidable for them [22]. This makes these transducer models amenable to algorithmic verification. Even though sequential transducers also enjoy similar properties, they are not expressive enough to model components of retransmission protocols. For example, they cannot model the sender transduction f.

In this work, we use *deterministic streaming string transducers* (SSTs) [5, 6] as finite-state descriptions of regular transductions. An SST is equipped with a finite set of string variables to store strings over the output alphabet. The output of an SST can be defined in terms of the string variables. As compared to equally

expressive models of two-way transducers and MSO-definable transductions, SSTs provide a more natural way of modeling retransmission protocols. In particular, the buffers used by the sliding window protocols for storing messages for retransmission can be modeled directly as string variables of SSTs.

We have designed transducer models of TinyOS serial communication protocol and HDLC with different CRC polynomials and have implemented a prototype tool for their verification. The sliding window protocols are fairly complex and are challenging even for manual proofs [24, 23]. Our approach proposes transducers for modeling only those aspects of these protocols that are relevant for reliable communication over noisy channels and gives an algorithmic verification technique. Some features of these protocols, such as timers and dynamic window sizes, are neither relevant nor amenable to our approach.

In Section 2, we motivate transducer models of retransmission protocols with an example. The modeling approach is presented in Section 3 and the verification algorithm is discussed in Section 4. Section 5 describes case studies. The related work is surveyed in Section 6. In Section 7, we sketch future work and conclude.

2 Motivating Example

Suppose we want to transmit two messages (bit strings) M_0 and M_1 over a noisy channel. Following the usual procedure of abstraction, let us denote them respectively by symbolic constants m_0 and m_1. In this example, we use the stop-and-wait protocol for error control. In this protocol, the sender prepends one bit sequence numbers to messages. This defines the space of valid encodings as $\{0,1\} \times \{m_0, m_1\}$. Let ack_0 and ack_1 denote acknowledgements indicating that the receiver expects a message with sequence number 0 or 1 next.

Consider a simple model of a noisy channel depicted in Fig. 1(a)–1(b). The sender-to-receiver channel is called the *forward channel* and the opposite channel is called the *backward channel*. The forward channel may corrupt the sequence bit of the encoded message as shown in Fig. 1(a). The messages to the left of the arrows are the original messages and those on the right indicate their possible incarnations at the other end of the channel. A dashed arrow indicates message corruption. The backward channel may corrupt ack_0 to ack_1 and vice versa.

The noisy channel can give rise to a sequence of message corruptions inducing the receiver to deliver an incorrect sequence of messages to its client. Fig. 1(c) shows such an example. The strings with Gray background indicate contents of the sliding window buffers. The dashed arrows annotating message exchanges indicate corruptions. In the first message transfer, the receiver cannot detect that $(1, m_0)$ is corrupt since $(1, m_0)$ does belong to the space of valid messages. It nevertheless discards it, as it is awaiting a message with sequence number 0. As ack_1 belongs the space of valid acknowledgements, the sender too cannot detect corruption and transmits the next message. Even after corruption, $(0, m_1)$ is accepted by the receiver since it has the expected sequence number. Overall, the sender believes that it has sent $\langle m_0, m_1 \rangle$ but the receiver accepts $\langle m_1, m_1 \rangle$.

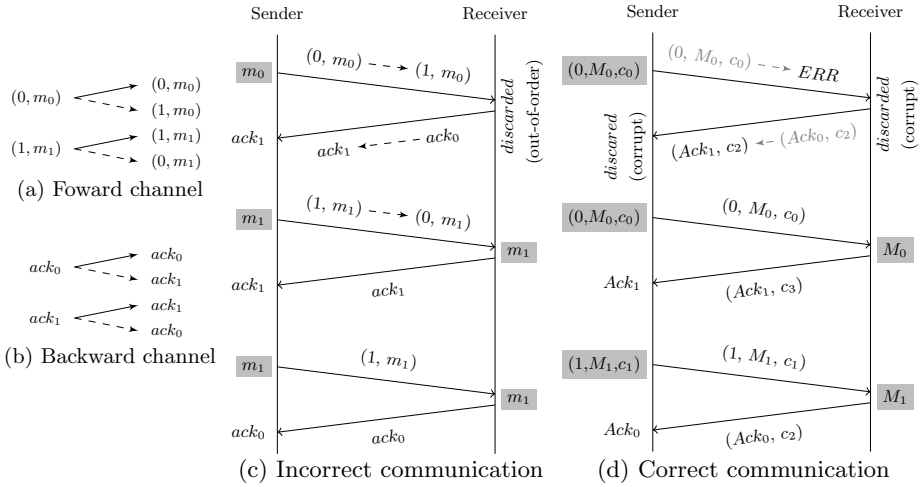


Fig. 1. A noisy channel model and scenarios of correct and incorrect communication

The problem with this protocol model is that it is unable to detect corruption. Our solution is to make the modeling more precise by treating messages, acknowledgements, and checksums as bit strings. We represent a *message encoding* as a triple (n, M, c) where n is the (fixed-length) bit encoding of a sequence number, M is a message string (of an arbitrary length), and c is the (fixed-length) checksum over strings n and M. It is possible to build finite-state protocol models by bounding the length of message strings. This could be useful for finding bugs quickly. We are however interested in verification of retransmission protocols without bounding the message length artificially.

We propose a transducer-based model that does not exhibit the incorrect communication scenario discussed above. We view the input to the sender as a sequence of messages and acknowledgements (protected by checksum). Consider the following string of inputs to the sender according to our scenario:

$$M_0 \ (Ack_1, c_2) \ (Ack_1, c_3) \ M_1 \ (Ack_0, c_2)$$

The acknowledgement strings corresponding to ack_0 and ack_1 are denoted by Ack_0 and Ack_1 respectively. Let the checksum be an even parity bit with c_2 and c_3 as the correct checksums for strings Ack_0 and Ack_1 respectively.

Consider a sender transducer which scans the input string in a single left-to-right pass. Here, it reads the message string M_0, generates an encoding $(0, M_0, c_0)$, and stores it in a string variable corresponding to the sliding window buffer. It then looks ahead at the acknowledgement to determine its output in the first round of transmission. It detects that (Ack_1, c_2) is an invalid/corrupt acknowledgement string. It does not know whether the message was received correctly at the receiver or not. It conservatively assumes that it was not delivered correctly. The sender sets its output to some string, say ERR, with an incorrect checksum – modeling the effect of corruption to M_0. Since (Ack_1, c_3) is a valid

encoding, the sender sets its output in the second round to $(0, M_0, c_0)$. Finally, it reads and encodes M_1. The encoding of M_1 is appended to the output since (Ack_0, c_2) indicates its correct delivery. The output string of the sender is thus:

$$ERR\ (0, M_0, c_0)\ (1, M_1, c_1)$$

We depict the message exchanges according the input/output of the sender transducer in Fig. 1(d). It differs from the corresponding scenario in Fig. 1(c) in the output of the sender in the second round. The sender outputs (an encoding of) message M_0 instead of M_1. Thus, this sender cannot be tricked into making an incorrect transition on receiving a corrupt acknowledgement. The receiver too can be modeled as a transducer. It takes the message triples, verifies the checksum, and accepts a message if the checksum agrees. In this example, it accepts the correct message strings M_0 and M_1 and in the same order as that used by the sender, in spite of corruptions of messages and acknowledgements.

3 Transducer Models of Retransmission Protocols

We use streaming string transducers for protocol modeling and start with a brief introduction to them before giving the transducer constructions.

3.1 Streaming String Transducers

A *deterministic streaming string transducer* (SST) [5] makes a *single* left-to-right pass over an input string to produce an output string. It uses a finite set of string variables to store strings over the output alphabet. Upon reading an input symbol, it may move to a new state and update all the string variables using a parallel (simultaneous) *copyless* assignment where the right-hand side expressions are concatenations of string variables and output symbols. A parallel assignment is copyless if no string variable appears more than once in any of the right-hand side expressions. For example, let a be an output symbol and $X = \{x, y\}$ be the set of variables. Then, $[x = x.y, y = a]$ is a copyless assignment, whereas, $[x = x.y, y = y]$ is not because y occurs twice on the right-hand side.

Formally, an SST is an 8-tuple $(Q, \Sigma_1, \Sigma_2, X, F, \delta, \gamma, q_0)$ machine, where Q is a finite set of states, Σ_1 and Σ_2 are finite sets of input and output symbols respectively, X is a finite set of string variables, F is a partial output function from Q to $(\Sigma_2 \cup X)^*$ with the constraint of copyless assignment, δ is a state transition function from $(Q \times \Sigma_1)$ to Q, γ is a variable update function from $(Q \times \Sigma_1 \times X)$ to $(\Sigma_2 \cup X)^*$ using copyless assignments and $q_0 \in Q$ is the initial state. The semantics of an SST is defined in terms of the summaries of a computation of the SST. Summaries are of the form (q, s) where q is a state and s is a valuation from X to Σ_2^* which represents the effect of a sequence of copyless assignments to the string variables. The second component can be extended to a valuation from $(\Sigma_2 \cup X)^*$ to Σ_2^*. In the initial configuration (q_0, s_0), s_0 maps each variable to the empty string. The transition function is defined by $\psi((q, s), a) = (\delta(q, a), s')$

where for each variable $x \in X$, $s'(x) = s(\gamma(q, a, x))$. For an input string $w \in \Sigma_1^*$, if $\psi^*((q_0, s_0), w) = (q, s)$, then if $F(q)$ is defined, the output string is $s(F(q))$ otherwise it is undefined.

Example Let us consider the sender transduction f described in Section 1. This can be implemented by an SST using four states, $Q = \{p_0, p_1, p_2, p_3\}$ where p_0 is the initial state, as shown in Fig. 2. Here, $\Sigma_1 = \Sigma_2 = \{0, 1, \sharp\}$. A message, M is a string over $\{0, 1\}$ and $\sharp$ is the message end marker. The SST for this example requires two string variables, $X = \{x_1, x_2\}$, both of which store a copy of the input message M. On reading 0 (resp. 1) in p_0, the SST remains in state p_0 and appends 0 (resp. 1) to both x_1 and x_2. The state transitions and variable updates are shown in Fig. 2. On reading the end-marker $\sharp$ in state p_0, the SST moves to state p_1 and appends $\sharp$ to both x_1 and x_2. In state p_1, there can be two input symbols, 0 (a negative acknowledgement) and 1 (a positive acknowledgement). Upon seeing 1, the SST moves from p_1 to p_2 and frees x_2. The output function in state p_2 is defined to be x_1, i.e., $F(p_2) = x_1$. The contents of x_1 here is $M\sharp$. On reading 0, it goes from state p_1 to state p_3 whose output is $x_1.x_2$, i.e., $F(p_3) = x_1.x_2$. This holds the output string $M\sharp M\sharp$. The output function F is undefined for other states.

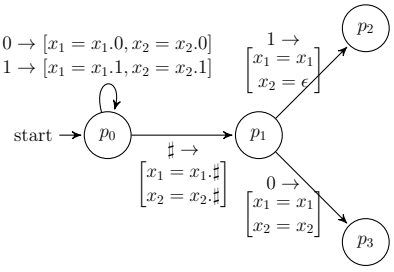


Fig. 2. An SST for the sender transduction f described in Section 1

3.2 Construction of Sender and Receiver Transducers

We present the transducer constructions abstractly without fixing the window sizes and CRC polynomials. An SST for a specific protocol configuration can be obtained by fixing values of these parameters. Due to the space constraints, for the sliding window logic, we consider only the *go-back-n* protocol. It is easy to extend the construction to stop-and-wait and selective-repeat protocols.

Sender SST Let W be the size of the sliding window of the sender. The sender can store up to W outstanding messages in a set of buffers. As more than one message can be in transit, to distinguish between them, the sender associates a sequence number with each message. Let the set of sequence numbers that the sender can use be $0, \ldots, N-1$. As a noisy channel may corrupt a message, the sender attaches a checksum with each of its outstanding messages.

In our setting, a sender receives a sequence of strings over $\{msg, ack_i, b\_ack\}$ where msg is a bit string that is provided to the sender by its client for transmission, ack_i acknowledges the outstanding messages up to sequence number $i-1$, and b_ack is an acknowledgement string with incorrect checksum – modeling corruption in the backward channel. Each acknowledgement corresponds to a *(re)transmission round*. If the receiver sees a corrupt message, it drops it and resends ack_i where i is one plus the sequence number of the last message received successfully. We call such repeated acknowledgements as *retransmission*

requests. The corruptions in the forward channel are thus identified by presence of retransmission requests in the sender's input string. The sender treats a *b_ack* as a retransmission request for all outstanding messages.

The behavior of a noisy channel, that is, message corruption, is modeled in the output function of the sender. The output of the sender is thus the sequence of messages that the receiver sees at its end of the noisy channel. It is a sequence of strings over $\{ERR, Emsg\}$ where ERR is a string with incorrect checksum, modeling a corrupt message. A string $Emsg$ is an *encoded message* consisting of concatenations of a sequence number n (represented by a bit string), the message msg being transmitted, and the checksum crc. We require some meta-symbols to separate the substrings of $Emsg$ and to separate consecutive messages and acknowledgements. For brevity, we omit them in this discussion.

For an outstanding message msg with sequence number i, the sender must decide whether the receiver sees ERR or its valid encoding $Emsg$. The encoding $Emsg$ is emitted only if the subsequent acknowledgement is ack_j where j is the sequence number that follows i. Otherwise, ERR is emitted. Thus, the sender must look ahead at the suffix of the string to determine its output. Since there are only a finite number of acknowledgement strings, they form a regular language. Thus, the sender can determine its output with a *regular look-ahead*. The output function concatenates the encoded messages and ERR strings. An encoded message is present in the output at most once, only when it is positively acknowledged. For every retransmission request for a message, the fixed string ERR is added to the output instead of the encoded message. Thus, the size of the output string is a constant multiple of the size of the input string. This ensures that the resulting transducer is regular.

Presently, we model the behavior of a noisy (forward) channel in the output of the sender. The channel can also be modeled independently as a deterministic SST. The sender can output an additional bit with a message to indicate whether it is to be delivered uncorrupted or not, based on the regular look-ahead at the acknowledgements. The channel SST could simply inspect this bit to determine its output.

Sliding Window Management We model each buffer of the sender by a string variable. Let the set X of string variables of the sender SST be $\{x_0, \ldots, x_{W-1}\}$. We remember the sliding window configuration and the next unused sequence number S in the states of the SST. A *sliding window configuration* is a pair of numbers F and L representing the first and the last occupied buffers respectively. Fig. 3 shows a snapshot of the abstract model of the sender SST. As a convention, if a variable is unmodified, we do not show an assignment to it.

Given F, L, and S, it is straightforward to identify sequence numbers of the outstanding messages. If the sender receives ack_i then the buffers containing the messages with sequence numbers up to $i-1$ are freed (by assigning ϵ to them). The pointer F is updated to reflect this as indicated in the transition t_3 in Fig. 3, where OM is the set of sequence numbers of the outstanding messages and R is the number of buffers that are to be freed. If ack_S arrives then all outstanding messages are delivered. The sliding window becomes empty ($F = 0, L = -1$).

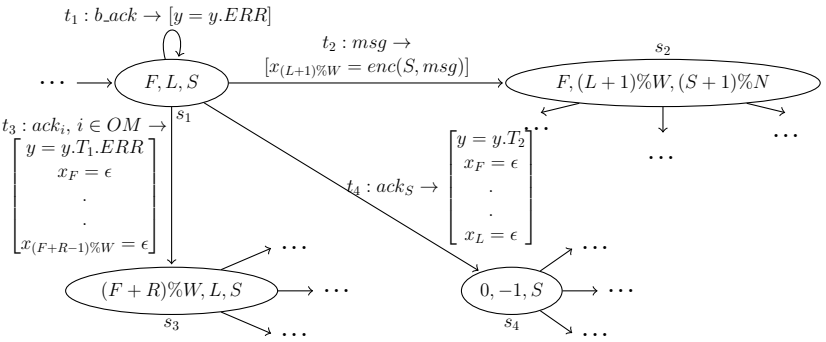


Fig. 3. A snapshot of the abstract model of the sender transducer where $T_1 = x_F.x_{(F+1)\%W} \ldots x_{(F+R-1)\%W}$ and $T_2 = x_F.x_{(F+1)\%W} \ldots x_L$

The transition t_4 shows this behavior. If the sliding window is not full, then the SST may read a message msg. It computes encoding of msg augmented with the sequence number S and stores the encoded message in the next free buffer. The transition t_2 illustrates this. The function enc denotes a CRC computation. We discuss it later in this section. A bad acknowledgement b_ack does not affect the state and variables of the SST as indicated by the transition t_1.

Handling Retransmission Requests The protocol runs in potentially unbounded number of (re)transmission rounds. We wish to define the output of the sender across all the rounds as a string. We accumulate the output in a string variable y. The updates to y on every type of acknowledgement are shown in Fig. 3. If ack_S is received, all the outstanding messages are appended to y in the order of their sequence numbers. The string T_2 used in transition t_4 is defined in the caption of the figure. If ack_i is received then the messages that are acknowledged positively are appended to y. For the messages that were transmitted but not received correctly, we append ERR to y, as shown in both t_1 and t_3. The output of the sender SST is set to y for every state in which message/acknowledgement is read completely. In each of these transitions, a variable x_k is used at most once on the right-hand side of variable update γ. It is either unmodified (not shown in the figure), or assigned to y and at the same time, is reset to ϵ. Thus, the resulting assignments conform to the copyless restriction of SSTs. It is easy to see that the sender SST is deterministic.

CRC Computation We now explain the modeling of the CRC computation, denoted by enc function in transition t_2. A CRC computation is parameterized with a *generator polynomial* p. If the degree of the polynomial is r, then it appends an r-bit checksum to the input string. We model all possible values of an r-bit checksum into states of the SST with the initial state corresponding to checksum of zero. We require only a single string variable x to store a copy of the input string. For a state q, representing a checksum value c, we define the

output to be $x.\#.c$ where $\#$ is a separator. We construct the transitions of the SST in such a way that if the SST ends up in a state q after reading the input string w, then the checksum value c, represented by q, is the checksum of w under polynomial p. The logic for constructing the state transitions follows from the semantics of linear feedback shift register (LFSR) circuits used for implementing CRC computations [20]. For want of space, we omit the details here.

The sender needs to *verify* checksum of the acknowledgement strings to classify them into ack_i or b_ack. The construction of SST for CRC verification is similar to that of CRC computation except for the variable update γ. From an input string, only the bits corresponding to the message content are copied.

Receiver SST The output from the sender (contents of variable y) forms the input to the receiver. The input to the receiver is thus a sequence of strings over $\{ERR, Emsg\}$. The output of the receiver is a sequence of strings over $\{msg\}$. In the go-back-n logic, the receiver accepts messages only in order.

We observe that the output of the sender contains the valid encoding of a message exactly once in its output. That is, there is no message duplication. This is because whenever a variable x_k is appended to y, it is also freed. Second, the messages are always transmitted in the same order as the order in which they are obtained from the client. Thus, there is no message reordering. These two observations simplify the design of the receiver SST. In particular, the receiver SST must only be able to distinguish between corrupt and correct messages.

The receiver SST therefore consists of only two string variables: a variable x to store the current message contents and a variable y to accumulate the output across all (re)transmission rounds. Upon receiving a message encoding, the message content msg is extracted and stored in x. If the CRC verifies then x is appended to y and freed. The output for any state of the SST is the variable y. Clearly, the receiver SST is deterministic.

3.3 Sequential Composition of Sender and Receiver Transducers

The sender and receiver SSTs represent distributed components of the protocol such that the output of the sender is the input to the receiver. These SSTs model the checksum computations, message and acknowledgement encodings, and the sliding window logic of a retransmission protocol. These are however internal details of the protocol. Our goal is to analyze the end-to-end input/output relation implemented by the protocol where the input is a sequence of strings over $\{msg, ack_i, b\_ack\}$ (same as the sender) and the output is a sequence of strings over $\{msg\}$ (same as the receiver).

SSTs are known to be closed under sequential composition [5]. Thus, the input/output relation implemented by the protocol can be represented as a *protocol* SST. Further, the protocol SST can be obtained algorithmically by the sequential composition of the sender and the receiver SSTs. Consider an SST S_2 to be composed with an SST S_1. The number of string variables in the composed SST is $2.|X_2|.|Q_2|.|X_1|$, where Q_2 is the set of states in S_2 and X_1, X_2 are the sets of string variables in S_1, S_2 respectively. It is beyond the scope of this paper

to discuss the algorithm for sequential composition. We refer the reader to [5]. The sender and receiver SSTs are much simpler to define than the protocol SSTs. Thus, the approach of modeling them individually and then composing is easier than constructing the SST for a protocol directly.

4 Verification of Transducer Models

In our models, each acknowledgement b_ack or ack_i corresponds to a separate (re)transmission round. Since the input to the sender can contain an unbounded number of acknowledgement strings, the number of (re)transmission rounds encoded in our models is unbounded. The number of messages received by the sender from its client too is not bounded. Similarly, there is no bound on the length of an individual message msg. Even with these sources of unboundedness, the verification problem for our models is decidable. In this section, we present the specification mechanism and the verification approach.

Specification SST The key property of a retransmission protocol is that the messages acknowledged by the receiver (across all rounds) are delivered to the receiver's client correctly and in the same order in which the client of the sender handed them to the sender. This property can be specified as an SST. Similar to the protocol SST, the input to the specification SST is a sequence of strings over $\{msg, ack_i, b\_ack\}$ and the output is a sequence of strings over $\{msg\}$. The *specification* SST does not encode sequence numbers and checksums. It also does not corrupt or retransmit messages. It mainly encodes the sliding window logic to interpret the acknowledgement strings and determine which strings are supposed to be delivered by the receiver to its client.

The specification SST has a similar set of states, transitions, string variables, and output function as the sender transducer. The main difference between the sender and the specification SSTs is in the variable update γ. We refer to Fig. 3 to describe the specification SST for the go-back-n protocol. The specification transducer stores a message msg as it is in a string variable along transition t_2. It neither attaches a sequence number nor a checksum with it. Since the output of the specification SST is the output of the receiver (and not that of the sender), for transitions t_1 and t_3, it does not append a corrupt message to the output string variable y. The transition t_4 remains unchanged.

Verification Approach The verification problem is to check equivalence between the protocol and the specification SSTs. Both these transducers are deterministic. Thus, for every input string w, we want to check whether the output of the protocol and the specification SSTs are same. The equivalence checking problem for (deterministic) SSTs is decidable [6]. The input to the verification algorithm consists of the sender and receiver SSTs and the specification SST as shown in Fig. 4. As discussed in Section 3.3, the protocol SST is obtained by sequential composition of the sender and the receiver SSTs.

Equivalence Checking We briefly outline the steps involved in equivalence checking of two SSTs. To check whether two SSTs, say S_1 and S_2, are equivalent,

the equivalence checking algorithm generates a 1-counter automaton, M. The objective is for M to determine whether there is an input string w and a position p such that the output symbols of S_1 and S_2 on w differ at position p. This can be generalized to infer whether (1) there is an input string w such that the output is defined for only one of S_1 or S_2, or (2) the outputs are defined but the lengths of the output strings differ.

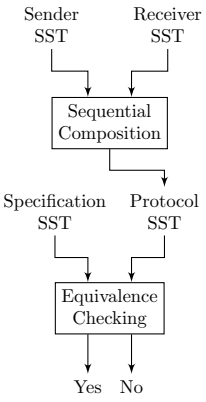


Fig. 4. Verification approach

The automaton M non-deterministically simulates S_1 and S_2 in parallel. For each of them, it guesses the position p and uses its counter to check whether the guess matches between S_1 and S_2. The complete details about the configurations of the states and transitions between them is available in [6]. A finite set, say F, of states of the automaton are identified such that if any state in F is 0-reachable in M, then the two transducers are not equivalent. Thus, the equivalence checking problem is reduced to checking 0-reachability in a 1-counter automaton. This problem is in NLOGSPACE. The number of states in M is linear in the number of states of S_1 and S_2, and exponential in the number of string variables of S_1 and S_2. Therefore, the SST equivalence problem is in PSPACE.

Decidable Extensions Instead of emitting the error string ERR for corrupt messages, the sender may resend the messages themselves. In such a case, the receiver needs to eliminate duplicates. However, for the sender to be a regular transducer, the length of the output strings must be a constant times the length of the input strings. This requirement can be satisfied by considering only a *bounded number of retransmissions* and using different string variables for different rounds. Several protocols like Philips Bounded Retransmission Protocol (BRP) fall in this class of protocols [21].

Another extension can be to model the effect of a noisy channel with non-determinism using the non-deterministic SSTs (NSSTs). NSSTs are closed under sequential composition, but the equivalence problem for them is undecidable [7]. However, there is a subclass of NSSTs, called *functional* NSSTs, whose equivalence checking problem is decidable. In the future, we plan to explore of non-deterministic and bounded versions of retransmission protocols.

5 Case Studies

We model two practical protocols as case studies:

1. **TinyOS** : TinyOS is an open source real-time operating system for wireless sensor networks. Serial communication is used for host-to-mote data transfer. The SerialP [1] software module of TinyOS computes the checksum and uses the stop-and-wait protocol in the host-to-mote direction.
2. **HDLC** : HDLC [26] is a bit-oriented protocol, that operates at data link layer. The software implementation computes checksum and uses go-back-n.

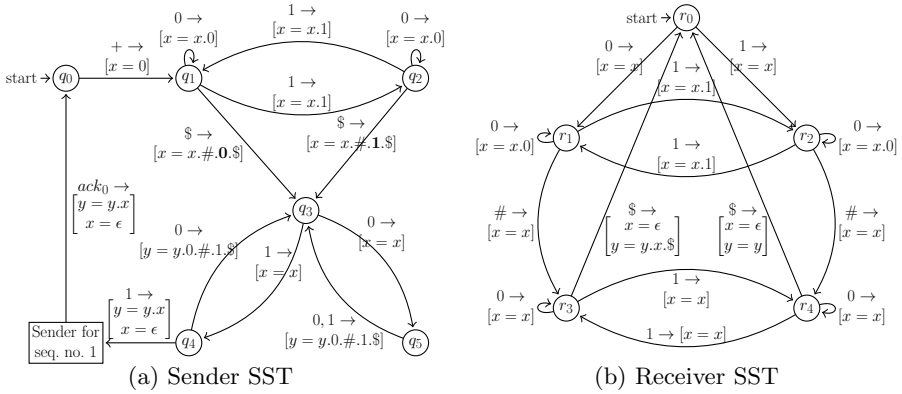
Table 1. Case studies

Protocol	CRC polynomial	W	Sender		Receiver		Protocol		Specification	
			$\|Q\|$	$\|X\|$	$\|Q\|$	$\|X\|$	$\|Q\|$	$\|X\|$	$\|Q\|$	$\|X\|$
TinyOS	$z+1$	1	12	2	5	2	38	11	10	2
TinyOS	z^2+1	1	20	2	9	2	80	15	14	2
HDLC	$z+1$	2	83	3	7	2	153	24	72	3

Table 1 summarizes the three protocol configurations that we model. For each protocol, we indicate the window size (W) and the CRC polynomials used. Real-world implementations of these protocols may use polynomials of higher degree in the CRC computation. The table also shows the number of states ($|Q|$) and variables ($|X|$) required in our case studies for sender, receiver, specification and protocol SSTs. The protocol SSTs are derived by our implementation of the sequential composition algorithm.

Modeling As an example, we present the sender and the receiver SSTs for the first protocol configuration in Table 1 (see Fig. 5). We build these according to the constructions described in Section 3. In these SSTs, if a variable update for any string variable v is not mentioned, it means that v is not updated, that is $v = v$. We use certain meta-symbols as separators: \$ to separate consecutive messages and acknowledgements, # to separate the message content from the checksum and + to indicate the start of a message. Here, ack_0 is encoded as 00, where the first 0 indicates the acknowledgement number and the second 0 indicates the checksum bit. Similarly, ack_1 is encoded as 11, whereas $b\_ack \in \{01, 10\}$ is a bad acknowledgement. The sender uses two string variables: x to store an encoding of the input message, and y to hold the output string. States q_0 to q_3 store the encoding of the input message in x, that is, $[x = enc(0, msg)]$. States q_1 and q_2 track the checksum value. State q_0 represents the empty sliding window, whereas q_3 represents the full sliding window. In state q_3, on receiving either ack_0 or b_ack, the SST appends a fixed error message, ERR (string 0.#.1.\$), to y. On receiving ack_1, the SST appends x to y, frees x and moves to the initial state for sequence number 1. The output function maps states q_0 and q_3 to y, and is undefined for other states. The part of the SST modeling sequence number 1 is similar (omitted from Fig. 5(a)).

The receiver SST shown in Fig. 5(b) needs two string variables: x to store the current message contents, and y to store the output (sequence of correctly received messages). Starting in r_0, the receiver SST first validates the sequence number but does not copy it. Then, the message content is extracted and stored in x, in states r_1 and r_2. State r_3 represents the checksum value 0 and state r_4 represents the checksum value 1. Thus r_3 indicates that the received input message encoding is not corrupt. So, after receiving the end symbol \$ in r_3, x is appended to y, and x is freed. State r_4 says that the received message is corrupt, and leaves y unchanged on receiving \$. The output function maps state r_0 to y,



(a) Sender SST (b) Receiver SST

Fig. 5. SSTs for the TinyOS SerialP protocol with CRC polynomial $z + 1$

and is undefined for other states. Note that all the variable updates are copyless in both the SSTs.

Verification We have implemented a prototype tool in OCaml to perform sequential composition and equivalence checking. For equivalence checking, our tool constructs the 1-counter automaton and uses ARMC for reachability checking [30]. The maximum time taken by sequential composition was 4s (for the HDLC protocol). Our implementation successfully verified both variants of TinyOS, but timed out on HDLC. The state space of 1-counter automata is exponential in the number of string variables. For HDLC, this proved to be a bottleneck. We aim to address scalability as part of the future work. One possibility is to explore minimization techniques for reducing the number of states and string variables of the protocol SST.

6 Related Work

The undecidability of verification problems for finite-state machines communicating asynchronously over unbounded *perfect* FIFO channels is shown in [12]. For some classes of systems and properties, decidability results are obtained in [19, 32]. For unbounded *lossy* FIFO channels, reachability, safety of system traces, and eventuality properties are decidable [4]. The semantics of lossy channels is orthogonal to our notion of noisy channels. For example, the scenario in Section 2 is applicable to lossy channels as well. A lossy forward channel may sometimes deliver corrupt messages to the receiver (and at other times, drop them). If we model messages as symbolic constants then the receiver cannot detect corruption and would deliver an incorrect sequence of messages to its client. Further, unlike these approaches, our choice of formalism for protocol modeling is transducers rather than communicating finite-state machines.

Sliding window protocols, being both complex and heavily used, are popular targets of verification techniques. Several automated techniques handle them by abstracting message contents. The work in [3] uses a class of regular expressions to represent channel contents where each message comes from a finite alphabet. In contrast, in our work, a message is treated as a finite bit string. The finite-state models obtained by abstracting message contents (and other parameters) are also verified by model checking (see survey [8]). Protocols with a fixed number of retransmission rounds are verified algorithmically in both untimed and timed cases in [17]. Our model does not impose bounds on the number of retransmission rounds. However, modeling timing constraints is beyond the scope of our method.

A number of deductive or semi-automated techniques have been developed for verification of sliding window protocols. The process-algebra framework LOTOS is used in [28], whereas, I/O automata are used with Coq theorem prover in [24]. Deductive theorem proving is used for reducing the complexity of protocol models and to obtain simpler abstractions suitable for algorithmic verification [23, 33]. Higher-order logic specifications of the protocols in the language of PVS are designed in [31]. Colored petri net models are used in [10] for modeling of stop-and-wait protocols. In [15], timed state machines are used for modeling sliding window protocols. Process algebra is also used in [9] to establish bi-similarity of these protocols with a queue.

The assumptions on reliability of channels vary across approaches. Similar to most of the above approaches, we consider non-duplicating FIFO channels. The approaches [33, 10] permit channels to re-order messages, whereas, [28, 15] permit both re-ordering and duplication. Most of the above approaches assume lossy channels and do not model message contents. Noisy channels are modeled in π-calculus through probabilistic semantics [38, 13]. A recent work [18] investigates decidability of control state reachability for ad-hoc networks in the presence of different types of node and communication failures.

Finite-state transducers are used in regular model checking for representing transition relations of systems whose configurations can be modeled as words [27, 37, 11]. The set of reachable states of these systems are represented by finite automata. However, in this context, the termination of fix-point computation is not guaranteed and the verification problem is in general undecidable. In contrast, the verification problem for the transducer models of the protocols presented by us, posed as functional equivalence checking, is decidable. Recently, more expressive transducer models are being designed by researchers, leading to new approaches to verification. SSTs have been introduced for pre/post verification and equivalence checking of single-pass programs operating over lists [6]. Symbolic finite transducers are introduced to analyze web sanitization functions [36].

7 Conclusions and Future Work

In this paper, we consider noisy communication channels that may corrupt messages. We show that for accurate modeling of the retransmission protocols, in the setting of noisy channels, the messages, checksums, and acknowledgements

must be modeled at the bit-level. We propose streaming string transducers as a modeling framework for retransmission protocols. Even though the message lengths and retransmission rounds are unbounded, we present an algorithm to verify the protocol models. In future, we want to explore non-deterministic and bounded versions of the retransmission protocol models.

Acknowledgements The first two authors were partially supported by Robert Bosch Centre for Cyber Physical Systems at the Indian Institute of Science.

References

1. http://www.tinyos.net/tinyos-2.x/doc/html/tep113.html.
2. http://www.ietf.org/rfc/rfc793.txt.
3. P. A. Abdulla, A. Annichini, and A. Bouajjani. Symbolic Verification of Lossy Channel Systems: Application to the Bounded Retransmission Protocol. In *TACAS*, pages 208–222, 1999.
4. P. A. Abdulla and B. Jonsson. Verifying Programs with Unreliable Channels. *Inf. Comput.*, 127(2):91–101, 1996.
5. R. Alur and P. Cerný. Expressiveness of streaming string transducers. In *FSTTCS*, pages 1–12, 2010.
6. R. Alur and P. Cerný. Streaming Transducers for Algorithmic Verification of Single-pass List-processing Programs. In *POPL*, pages 599–610, 2011.
7. R. Alur and J. V. Deshmukh. Nondeterministic Streaming String Transducers. In *ICALP*, pages 1–20, 2011.
8. F. Babich and L. Deotto. Formal Methods for Specification and Analysis of Communication Protocols. *IEEE Comm. Surveys and Tutorials*, 4(1):2–20, 2002.
9. B. Badban, W. Fokkink, J. Groote, J. Pang, and J. Pol. Verification of a Sliding Window Protocol in μCRL and PVS. *Formal Asp. Comput.*, 17(3):342–388, 2005.
10. J. Billington and G. E. Gallasch. How Stop and Wait Protocols Can Fail over the Internet. In *FORTE*, pages 209–223, 2003.
11. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *CAV*, pages 403–418, 2000.
12. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.
13. Y. Cao. Reliability of Mobile Processes with Noisy Channels. *IEEE Trans. Computers*, 61(9):1217–1230, 2012.
14. V. Cerf and R. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, 1974.
15. D. Chkliaev, J. Hooman, and E. P. de Vink. Verification and Improvement of the Sliding Window Protocol. In *TACAS*, pages 113–127, 2003.
16. M. Chytil and V. Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In *ICALP*, pages 135–147, 1977.
17. P. R. D'Argenio, J. P. Katoen, T. C. Ruys, and G. J. Tretmans. The Bounded Retransmission Protocol must be on time! In *TACAS*, pages 416–431, 1997.
18. G. Delzanno, A. Sangnier, and G. Zavattaro. Verification of Ad Hoc Networks with Node and Communication Failures. In *FMOODS/FORTE*, pages 235–250, 2012.
19. A. Finkel. Decidability of the termination problem for completely specified protocols. *Distrib. Comput.*, 7(3):129–135, March 1994.

20. B. Forouzan. *Data Communications and Networking*. McGraw-Hill Companies, 2012.
21. J. Groote and J. Pol. A Bounded Retransmission Protocol for Large Data Packets. In *AMAST*, pages 536–550, 1996.
22. E. Gurari. The equivalence problem for deterministic two-way sequential transducers is decidable. *SIAM J. Comput.*, 11(3):448–452, 1982.
23. K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *FME*, pages 662–681, 1996.
24. L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. Proof-Checking a Data Link Protocol. In *TYPES*, pages 127–165, 1993.
25. G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
26. ISO. Data Communication - HDLC Procedures - Elements of Procedure. Technical Report ISO 4335, International Organization for Standardization, 1979.
27. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *CAV*, pages 424–435, 1997.
28. E. Madelaine and D. Vergamini. Specification and Verification of a Sliding Window Protocol in LOTOS. In *FORTE*, pages 495–510, 1991.
29. W. W. Peterson and D.T. Brown. Cyclic Codes for Error Detection. In *IRE*, pages 228–235, 1961.
30. A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *PADL*, pages 245–259, 2007.
31. V. Rusu. Verifying a Sliding Window Protocol using PVS. In *FORTE*, pages 251–268, 2001.
32. A. P. Sistla and L. D. Zuck. Automatic Temporal Verification of Buffer Systems. In *CAV*, pages 59–69, 1991.
33. M. A. Smith and N. Klarlund. Verification of a Sliding Window Protocol Using IOA and MONA. In *FORTE*, pages 19–34, 2000.
34. V. Stenning. A Data Transfer Protocol. *Computer Networks*, 1:99–110, 1976.
35. A.S. Tanenbaum and D. Wetherall. *Computer Networks*. Pearson, 2010.
36. M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. Symbolic Finite State Transducers: Algorithms and Applications. In *POPL*, pages 137–150, 2012.
37. P. Wolper and B. Boigelot. Verifying Systems with Infinite but Regular State Spaces. In *CAV*, pages 88–97, 1998.
38. M. Ying. π-calculus with noisy channels. *Acta Inf.*, 41(9):525–593, 2005.