What Packets May Come: Automata for Network Monitoring

Karthikeyan Bhargavan
University of Pennsylvania
bkarthik@seas.upenn.edu

Satish Chandra
Bell Laboratories
schandra@bell-labs.com

Peter J. McCann
Bell Laboratories
mccap@bell-labs.com

Carl A. Gunter
University of Pennsylvania
gunter@cis.upenn.edu

Abstract

We consider the problem of *monitoring* an interactive device, such as an implementation of a network protocol, in order to check whether its execution is consistent with its specification. At first glance, it appears that a monitor could simply follow the input-output trace of the device and check it against the specification. However, if the monitor is able to observe inputs and outputs only from a vantage point *external* to the device—as is typically the case—the problem becomes surprisingly difficult. This is because events may be buffered, and even lost, between the monitor and the device, in which case, even for a correctly running device, the trace observed at the monitor could be inconsistent with the specification.

In this paper, we formulate the problem of external monitoring as a *language recognition problem*. Given a specification that accepts a certain language of input-output sequences, we define another language that corresponds to input-output sequences observable externally. We also give an algorithm to check membership of a string in the derived language. It turns out that without any assumptions on the specification, this algorithm may take unbounded time and space. To address this problem, we define a series of properties of device specifications or protocols that can be exploited to construct efficient language recognizers at the monitor. We characterize these properties and provide complexity bounds for monitoring in each case.

To illustrate our methodology, we describe properties of the Internet Transmission Control Protocol (TCP), and identify features of the protocol that make it challenging to monitor efficiently.

1 Introduction

Computer networking protocols have always been appealing candidates for applications of automata theory. Not only are protocols commonly specified as finite-state automata, much of the current technology of implementing and verifying protocols relies on application of automata theory. Examples include Spin/Promela [5], Verisoft [4], Esterel [1], etc.

We consider the problem of *monitoring* the execution of network protocols. Suppose a given piece of computer equipment claims to implement a certain network protocol through one of its communication interfaces. We seek to introduce a passive monitor outside this interface that can watch all the bits traveling to and from the device under test, and check them for some properties. Such a monitor could give important information about the proper running of the protocol. Monitoring is complementary both to the implementation and to the verification of network protocols.

A passive monitor would essentially mimic the actions that the device is expected to take in response to the input that it receives, except it would not actually put any output on the wire. Rather, it would pick up the output generated by the device under test, and compare the output that it computed against the output it sniffed from the wire. In this role, a passive monitor is a *language recognizer*, where the language that it understands is the input-output behavior of a given networking protocol.

Passive monitors can observe many useful properties of real-world protocols. We give examples of such properties for the Internet protocol TCP in Section 6. Such a capability can play a useful role in testing new implementations, and also in guarding against network intrusion and other security violations.

Several complications hamper our ability to construct passive monitors accurately and efficiently in the real world. The most important complication is the *fidelity* of observation of traffic by the monitor. In general, our monitor is not located exactly *at* the device, in the sense that it does not synchronously observe input and output actions of the automaton inside the device. Therefore, the monitor might fail to observe certain packets that the device sees, or might see certain packets that the device fails to see. Also, because of buffers at the input and output ports of the device, the monitor might observe a sequence of events in a different order than the device. Another complication is that of *under-specification* of protocols. For example, the TCP specification allows certain leeway as to how often a receiver must generate an acknowledgment to a data packet. Because of these complications, the input-output language that the monitor must recognize could be significantly different from the language that a given device processes, or claims to process.

In this paper, we give a systematic presentation of passive monitoring as a language recognition problem, while incorporating these real-world complications to the extent possible. Furthermore, the algorithms we present are suitable for on-line monitoring of protocol execution, that is, we do not collect traces and analyze them separately. Thus, speed of monitoring is also an important concern. Most prior work in network monitoring either performs off-line monitoring [11], or deals with real-world problems in *ad hoc* ways [12]. We believe an automata-theoretic approach helps us achieve robust algorithms and also lets us understand the kind of properties that we can or cannot monitor efficiently.

In the general case, we could perform on-line monitoring using an expensive brute-force search algorithm, which is essentially an inefficient "parser" for the "real world" version of the language. In some cases, there is not even any *a priori* upper bound on the amount of space this process may take. However, for many kinds of properties that we may wish to check, it is possible to create faster language recognizers. In this paper, we also give a systematic exploration of the space of properties that lets us construct efficient monitors.

We draw an analogy to the problem on constructing parsers for the syntax of programming languages. In general, a particular syntax could be a context-free grammar. However, parsers for general context-free grammars are expensive: they may take $O(n^3)$ time in the size of the input. Thus, most real syntax specifications require restrictions on the grammar, such as the language should be LL(k), LALR(1), etc. For grammars that obey such restrictions, fast parsers can be constructed. We have defined an analogous range of languages for monitoring. If the language of the protocol obeys certain properties, efficient monitoring programs can be created. These language characteristics could be used during the specification process, to select properties that can be easily monitored at run-time, or even during the protocol design phase, if the designer is especially concerned that a protocol be amenable to efficient monitoring as might be the case with critical security protocols.

The contributions of our work are the following:

- We define automata suitable for network monitoring, taking into account real-world complications such as buffering and packet loss.
- We classify properties of network protocols that let us construct efficient monitors, and also point out cases in which monitoring would necessarily be an intractable problem.
- For an important real-world protocol, TCP, we identify properties that can be efficiently monitored based on the theory that we develop.

The rest of the paper is organized as follows: in Section 2, we give a description of the kinds of mismatches that can be expected between a trace collected by a monitor and the trace actually seen by a device under test. In Section 3 we characterize these differences formally, and in Section 4, we give a brute-force approach to checking whether a given trace collected at a monitor could have resulted from a correct execution. We proceed in Section 5 to give special properties of specifications that, when satisfied, allow us to construct more efficient on-line property monitors. In Section 6 we show how our techniques could be applied to monitoring real properties of TCP. Finally in Section 7 we discuss connections to related work and conclude.

2 Fidelity in Network Monitoring

In the context of monitoring, the term *fidelity* refers to the closeness with which the sequence of input and output events seen by the device under test matches the sequence of events observed by the monitor. The extent of infidelity is determined by the performance of the monitor and its placement in the network. A perfect fidelity monitor, one that sees exactly the inputs and outputs of the device under test, can be obtained by instrumenting the protocol stack on the device itself. Such a monitor is depicted as *M1* in Figure 1. Such a *co-located* monitor observes input and output actions of the device synchronously with the device—it encounters no fidelity problems. Such monitors are hard to deploy because of the need to put new software on the monitored system.

An alternative is to place the monitor somewhere in the network and observe inputs and outputs as they pass across the network links. While there are many possible points at which a monitor could be placed (see Figure 1), a *co-networked* monitor and a *bottleneck* monitor are particularly useful, as they are able to observe all traffic between the device and the remote host. A co-networked monitor sits outside the device on the physical network to which its network interface card (NIC) is attached, whereas a bottleneck monitor sits outside the NIC of the gateway of a local area network. Of the two, a co-networked monitor can enjoy better fidelity, as there is no network element between it and the device.

We have conducted initial experiments to determine the extent and nature of infidelities for co-networked monitors. Our experiment consisted of taking a PC (400 MHz Pentium II) running Linux and using it to monitor input to another similar Linux PC over 100 Mbps Ethernet. The monitor accepted traffic off an Ethernet hub that we introduced between the device and the rest of the network. Our experiments show that with proper operating system engineering, a co-networked monitor could easily keep up with inter-packet arrival time of about 20 microseconds, assuming no expensive computation is performed per packet. At these speeds, it is feasible to monitor typical TCP/IP protocol traffic over 100 Mbps Ethernet, without the monitor dropping any packets. Moreover, this is achieved using only stock hardware (desktop PC's and an Ethernet hub[1]); extra hardware support could handle heavier loads. Consequently, we can assume the existence of a co-networked monitor that does not drop packets for this kind of host and network.

The primary fidelity challenges arise in dealing with buffering on the device itself. If a protocol S is being run by the device, then the goal of the monitor M is to determine if S is properly implemented. However, this must be done by observing behavior on the network, and there are input and output buffers between the device and the network. The situation is depicted in Figure 2. The input and output buffers may over-fill and cause packet losses between the device and the network, thus introducing infidelities between the events observed at the device, and the co-networked monitor M. Note that the over-fill may be caused by packets from some other protocol that the device is engaged in; the monitor may never even look at these packets. Our experiments showed that under heavy network loads this kind of infidelity was quite possible for input buffers. However, the

[1]In this sense it is not exactly co-networked as defined above, but we assume this hub repeats all traffic to all ports. A truly co-networked monitor would need special hardware support.

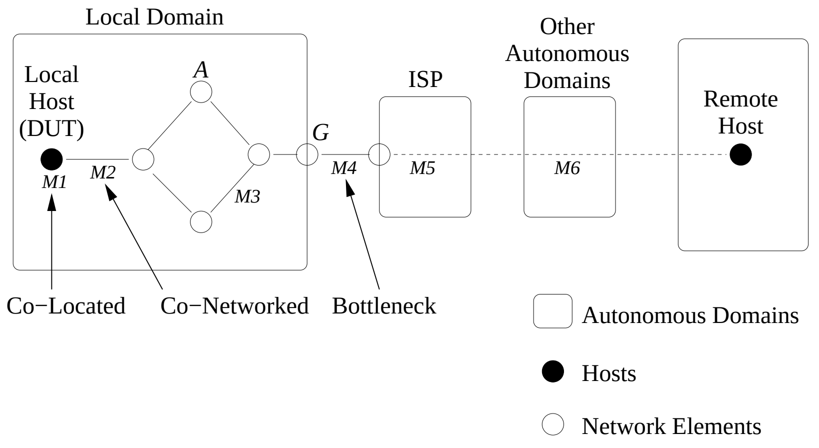

Figure 1: Monitor Placement: Monitor *M1* is co-located with the device under test (DUT); it is unfeasible to deploy without modifying DUT software. Monitor *M2* is co-networked and monitor *M4* is at a bottleneck location. Monitor *M3* will not observe traffic passing through network element A. *M5* is located at an Internet service provider (ISP), and *M6* is located in another service provider's network. Because the Internet Protocol may drop, duplicate, or re-order datagrams, *M5* and *M6* can experience significant infidelities.

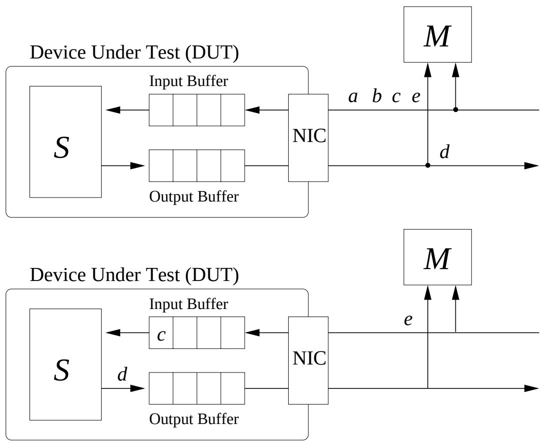

Figure 2: Buffers in the DUT. M may observe inputs and outputs in a different order than S. The second figure shows one possible execution sequence at the DUT.

kernel will ordinarily throttle the protocol implementation to avoid loss in output buffers so we will assume in this paper that outputs are not lost at a device's output buffer.

Aside from input buffer loss, the major infidelity between the device and a co-networked monitor M arises from different perceptions of packet arrival and dispatch caused by the input and output buffering. We describe this phenomena in some detail in the next section. To see the issue briefly, note that it is impossible for M to tell in Figure 2 whether output d of S was created by the device before or after the device observed a or b, or even whether b was dropped before it reached S.

3 Model

We assume that the device under test is a deterministic and reactive automaton, following a specification S. At each step, either it consumes an input, or produces an output. Outputs may be produced in reaction to inputs, or in reaction to events such as timer expirations, that are not visible to the outside. Our model of S is similar to a deterministic I/O Automaton [9]; it is a machine with a (possibly infinite) state space carrying out parameterized input and output actions.

Let us denote input of a symbol a by the token i_a and output of a symbol a by the token o_a. S recognizes certain finite sequences of tokens, for example, $i_a\ i_b\ o_d$. Call the set of such finite sequences L_S.

We introduce a co-networked monitor $M(S, m, n)$ with input and output buffers between it and the device under test. The system now contains three components: S; I, which is an input buffer of size m; and O, which is an output buffer of size n. For convenience, we will normally abbreviate $M(S, m, n)$ to M.

We introduce four new tokens. iq_a corresponds to enqueing of an *i*nput symbol a in *q*ueue I. id_a corresponds to deletion of a from head of I and simultaneous *i*nput into *d*evice. od_a corresponds to *o*utput of a from *d*evice and simultaneous enqueuing into O. Finally, oq_a corresponds to *o*utput of a from the head of *q*ueue O. Note that M gets to see only tokens iq and oq. We also assume that observation of iq_a at the monitor is simultaneous with enqueing of a into I, and observation of oq_a at the monitor is simultaneous with the dequeing of a from O.

An execution of this system can be represented by a sequence of such tokens. Consider the sequence

$$iq_a\ iq_b\ iq_c\ id_a\ id_b\ od_d\ iq_e\ oq_d$$

This sequence represents the following events: a, b, and c got enqueued in the input queue I; the device input a from the head of the queue; the device input b from the head of the queue and then produced output d, which went on the output queue O; another symbol e came into I; and, finally, d left O. The monitor sees the sequence

$$iq_a\ iq_b\ iq_c\ iq_e\ oq_d$$

and we will define the language recognized by the monitor in terms such sequences. We first introduce a notion of *admissible* execution sequences under the buffering restrictions. We are interested in the admissible executions that are allowed by S.

Definition 1 (Admissibility) *ω, a string of iq, oq, id and od tokens, is said to be an* admissible *execution sequence with respect to $M(S, m, n)$, if the following are true:*

FIFO Input: *the sequence of id tokens in ω is the same as the sequence of iq tokens,*

FIFO Output: *the sequence of oq tokens in ω is the same as the sequence of od tokens,*

Causality: *the k'th id token can only occur after the k'th iq token, and the k'th oq token can only occur after the k'th od token,*

Input Buffer Limit: *in any prefix of ω, the number of iq tokens minus the number of id tokens must not exceed m, and*

Output Buffer Limit: *in any prefix of ω, the number of od tokens minus the number of oq tokens must not exceed n.*

Now we have a way of defining the language recognized by M. Essentially, a string τ belongs to L_M, if there is some admissible sequence ω whose *iq/oq* projection is τ, and whose *id/od* projection belongs to L_S. For each string s in L_S, we can construct several τ that must be in L_M. Let us first construct an admissible sequence whose *id/od* projection is s. We take each input token i_a in s and split it into two consecutive tokens iq_a and id_a. We split each output token o_a into two consecutive tokens od_a and oq_a. Clearly, this is an admissible sequence (no buffering is carried out). But we could also generate other sequences of tokens corresponding to this execution. We can move each *iq* token *backwards*, skipping over any number of tokens, as long as we do not violate relative orders of *iq* events, and we do not violate input buffering limits. Likewise, we can move each *oq* token *forward*, as long as we do not violate relative orders of *oq* events, and we do not violate output buffering limits. Every such sequence ω yields a string τ in the language recognized by M (L_M), simply by erasing out the *id* and *od* tokens.

Example. Consider the following string in S:

$$i_a\ i_b\ o_c\ i_d\ o_e\ i_f\ i_g$$

We can arrive at the following admissible sequence (labeled A). We shall normally ignore particular symbols a, b, etc., and instead label the various tokens by order of their appearance in the string, using the notation id^k for the k'th occurrence of *id*.

$$A : iq^1\ id^1\ iq^2\ id^2\ od^1\ oq^1\ iq^3\ id^3\ od^2\ oq^2\ iq^4\ id^4\ iq^5\ id^5$$

Now, let us allow an input buffer of three elements and an output buffer of two elements. Here are some additional admissible strings:

$$B : iq^1\ iq^2\ id^1\ id^2\ od^1\ oq^1\ iq^3\ id^3\ od^2\ oq^2\ iq^4\ id^4\ iq^5\ id^5$$

$$C : iq^1\ iq^2\ iq^3\ id^1\ id^2\ od^1\ oq^1\ id^3\ od^2\ oq^2\ iq^4\ id^4\ iq^5\ id^5$$

$$D : iq^1\ iq^2\ iq^3\ id^1\ id^2\ od^1\ id^3\ od^2\ oq^1\ oq^2\ iq^4\ id^4\ iq^5\ id^5$$

$$E : iq^1\ iq^2\ iq^3\ id^1\ id^2\ od^1\ iq^4\ iq^5\ id^3\ od^2\ oq^1\ oq^2\ id^4\ id^5$$

The following string is not admissible because the input queue cannot accommodate iq_4.

$$F : iq^1\ iq^2\ iq^3\ iq^4\ id^1\ id^2\ od^1\ iq^5\ id^3\ od^2\ oq^1\ oq^2\ id^4\ id^5$$

For each admissible string given above (A-E), we can arrive at a string in L_M by erasing the *id* and *od* tokens, as shown below. Not each admissible string gives a unique string in L_M. Also, some non-admissible execution sequences can yield the same string in L_M, as an admissible sequence (e.g. E and F).

$$A, B : iq^1\ iq^2\ oq^1\ iq^3\ oq^2\ iq^4\ iq^5$$

$$C, D : iq^1\ iq^2\ iq^3\ oq^1\ oq^2\ iq^4\ iq^5$$

$$E, F : iq^1\ iq^2\ iq^3\ iq^4\ iq^5\ oq^1\ oq^2$$

3.1 Eliminating Output Buffering

In this section, we give a construction that lets us assume that the monitor M does not need to consider an output buffer, by suitably adjusting the size of the input buffer. This simplification is useful in reasoning about properties of protocols that permit efficient monitoring.

Theorem 1 *$L_{M(S,m,n)} \subseteq L_{M(S,m+c_U*n,0)}$, where c_U is a constant dependent on S: c_U is the maximum number of input symbols without an intervening output symbol in any string in L_S.*

We claim that a monitor similar to M, but that has an associated input buffer of size $m + c_U * n$ and no output buffer, can admit all the observable behaviors of M. That is, if a sequence of tokens τ observed by M can be derived constructively for M using the procedure described in the previous section from a string $s \in L_S$, then τ can also be derived constructively for this new monitor from the same s. We denote this new monitor $M(S, m + c_U * n, 0)$ by M'.

In the monitoring process, we will make use of a consequence of this theorem that $\tau \notin L_{M'} \Rightarrow \tau \notin L_M$. By construction of L_M, $\tau \notin L_M$ implies that the device is not following S. Thus, by inferring $\tau \notin L_{M'}$, M' can infer that a certain input-output sequence it—or M—observes is definitely inconsistent with S.

Sketch of Proof. The theorem does not trivially hold, because a string in L_M may be a result of output buffering, whereas there is no output buffer in M'.

First assume that every input to S generates an output ($c_U = 1$). Let $\tau \in L_M$. By construction, τ is a result of erasing *id* and *od* tokens from some admissible execution ω with respect to M. ω contains a unique execution s in L_S. We now show a transformation on ω that makes it admissible with respect to M', while still containing the same s (*id/od* projection) and the same τ (*iq/oq* projection) in it. Then, $\tau \in L_{M'}$, because we can obtain τ from s by first constructing the transformed ω, which is admissible with respect to M', and then erasing *id* and *od* tokens from it.

We perform the following transformation on ω. We start from the end of ω, proceeding backwards. On encountering

od^k, the k'th occurrence of od, if od^k is not immediately followed by oq^k, we move it forward some number of steps to make it so. In doing this, we might skip over other iq and oq tokens. Any id tokens in between od^k and oq^k are moved in order after oq^k. No od tokens can occur in between because they must have already been moved to their appropriate positions. By skipping over all the intermediate oq tokens, we reduce the output buffering requirement to zero. For every iq token that some id token skips over, we increase the input buffering requirement by one. We argue that the maximum increase in the input buffering requirement is n, the size of the output buffer O.

Consider the largest number of iq tokens between od^k and oq^k. Suppose that at the position after the od^k, there are p elements in the input buffer. There is at least one element in the output buffer (the content of od^k). The first $m-p$ iq tokens fill up the input buffer. Every iq token after these must be preceded by some id token, which produces an od. Therefore, $n-1$ such iq tokens will cause the output buffer to be filled up. One more id and iq can also appear—in that order—for whom the output has not yet been produced by the device. No more iq's are allowed before oq^k. After the last such iq, the number of iq tokens minus the number of id tokens can be at most $(m-p+n)+p$, since we pushed all the intermediate id tokens out. This will be admissible in a monitor with input buffer size $m+n$.

In general, S need not expect an output for every input; the above transformation no longer works. However, suppose it is given that S cannot accept any more than c_U inputs before it must see an output. If we supply M' with an input buffer of size $m+c_U*n$, the theorem still holds. To see this, we need to modify our previous counting argument, as to how many iq tokens might occur between od^k and oq^k. The first $m-p$ iq tokens fill up the input buffer. The next c_U*n iq's must be preceded by c_U*n id's, that produce $n-1$ od's filling up the output buffer. No more iq's are possible.

Example. Let $m=2$ and $n=2$. Consider the following ω, where $c_U=2$. This string is admissible with respect to M.

$$iq^1\ iq^2\ \boxed{od^1}\ \boxed{id^1}\ iq^3\ \boxed{id^2}\ iq^4\ \boxed{od^2}\ \boxed{id^3}\ iq^5\ \boxed{id^4}\ iq^6\ oq^1\ oq^2$$

After moving the od tokens, we arrive at the following transformed ω. Note that the relative ordering of id, od tokens must be maintained, otherwise we change the underlying input to S.

$$iq^1\ iq^2\ iq^3\ iq^4\ iq^5\ iq^6\ \boxed{od^1\ oq^1}\ \boxed{id^1}\ \boxed{id^2}\ \boxed{od^2\ oq^2}\ \boxed{id^3}\ \boxed{id^4}$$

This transformation increases the maximum input buffering requirement to 6, which is admissible to M', which has an input buffer of size 6 $(2+2*2)$.

3.2 Dealing with Loss

We now introduce into this model the possibility of losing a packet between the co-located monitor and the device, i.e., the monitor observes some input packets that the device does not. The model is as follows: we assume that the output from the input queue I could either be absorbed by a *loss unit* L, never to be seen again, or goes into the device as before. We use token il to denote the loss event that consumes the head of the input queue.

Example. Consider the following sequence:

$$iq_a\ il_a\ iq_b\ iq_c\ id_b\ od_d\ il_c\ oq_d$$

In this sequence, S executes the following string:

$$i_b\ o_d$$

Whereas, the monitor observes the following tokens:

$$iq_a\ iq_b\ iq_c\ oq_d$$

Inputs a and c are lost in the loss unit.

Note that the M to M' conversion of the previous subsection would need to know about the maximum effective number of inputs seen by the monitor before an output appears. This implies that we must impose a limit on the number of il tokens that can appear in a sequence without an id token. If there can be only less than c_L il tokens between two id tokens, then M' needs an input buffer of size $m+c_L*c_U*n$ to be able to eliminate the size n output buffer.

Corollary 1 *If S cannot accept more that c_U id tokens without an intervening od token, and there must be less than c_L il tokens between two id tokens, then,*

$$L_{M(S,m,n)} \subseteq L_{M(S,m+c_U*c_L*n,0)}$$

For the remainder of the paper, we assume a monitor of the form M', appropriately parameterized with the values of m, n, c_U and c_L. Furthermore, we shall use B to refer to the input buffering requirement $(m+c_U*c_L*n)$. The admissible strings ω may now also contain il tokens (in addition to iq, id, od and oq), and their placement will follow conditions mentioned here. We modify the definition of admissibility to take il tokens into account: the FIFO input, causality, and input buffer limit clauses are updated appropriately, treating il tokens just like id tokens. In addition, we have an *Input Loss Limit* condition: in any prefix of ω, the number of consecutive il's without an intervening id token is less than c_L.

Corollary 2 *Consider a string admissible to M'. Ignoring any intervening id or il tokens, the maximum number of iq tokens without an intervening od oq pair is bounded by $B+c_U*c_L$.*

The corollary holds because c_U*c_L id and iq pairs can appear in addition to B iq's, without increasing the effective buffering requirement. (Another id must force an od.) The importance of this corollary will become apparent in the algorithms presented in Section 5.

4 Algorithm for Co-networked Monitoring

In general, we would like to check that a given device under test is a proper implementation of a specification S, given a trace collected from a co-networked passive monitor. That is, given a trace τ of iq and oq events observed at a monitor that is separated from the device by a loss unit (L), input (I) and output buffer (O) as specified in Section 3, we would like to determine whether the device is behaving in accordance with S. The device is not incorrect as long as $\tau \in L_{M'}$: we can exhibit a sequence ω which is derived from τ by the addition of id, il, and od events and which is consistent with the following set of conditions:

- ω is admissible with respect to M', and

- the projection of ω that includes only *id* and *od* tokens (denoted $[\omega]_{id,od}$) is consistent with the specification of S.

Assume we have a function g that checks S on a sequence α of *id* and *od* tokens and tells us whether the sequence is in L_S; g could be used directly as a co-located monitor for S. We write the query to g in the form $\alpha \in g$. We assume that g is a safety property: it is prefix-closed, and can be checked over finite traces. Our problem is to construct a function $F(g, \tau)$, that given a trace τ of *iq* and *oq* events collected by a co-networked monitor and a function g, tells us whether the trace corresponds to some proper execution with respect to S. A non-deterministic algorithm for F is straight-forward. Given a τ, guess a sequence ω admissible with respect to M', such that $[\omega]_{iq,oq} = \tau$. If $[\omega]_{id,od}$ satisfies g, τ is OK. Otherwise, report failure.

A naive determinization of the above algorithm is brute-force search: simply construct all possible ω's from τ, checking each admissible ω against g, until a match is found or all possibilities are exhausted. Additionally, we would like F to be computable *on-line*, meaning that it should make only one pass over the input τ. We give such a brute-force breadth-first-search algorithm in Figure 3. We call this algorithm $BF(g, \tau)$.

Data Type. Ω is a set of pairs (admissible string, string of input symbols). Initially, $\Omega = \{(\epsilon, \epsilon)\}$

Event Handlers. On receiving

```
• $iq_x$:
    1. $\forall(\omega, b) \in \Omega$:
           delete $(\omega, b)$ from $\Omega$;
           check-add $(\omega :: iq_x, b :: iq_x)$ to $\Omega$.
    2. iterate until no more additions to $\Omega$:
           $\forall(\omega, iq_y :: b) \in \Omega$:
               check-add $(\omega :: id_y, b)$ to $\Omega$;
               check-add $(\omega :: il_y, b)$ to $\Omega$
• $oq_x$:
    1. $\forall(\omega, b) \in \Omega$:
           delete $(\omega, b)$ from $\Omega$;
           check-add $(\omega :: od_x :: oq_x, b)$ to $\Omega$.
    2. iterate until no more additions to $\Omega$:
           $\forall(\omega, iq_y :: b) \in \Omega$:
               check-add $(\omega :: id_y, b)$ to $\Omega$;
               check-add $(\omega :: il_y, b)$ to $\Omega$
```

where check-add adds (ω, b) to Ω if and only if:

- $|b| < B$, and
- $[\omega]_{id,od} \in g$
- in any prefix of ω, the number of consecutive *il*'s without an intervening *id* token is less than c_L.

If $\Omega = \phi$ after executing either event handler, flag an error.

Figure 3: Brute-Force Monitoring Algorithm

Theorem 2 *$BF(g, \tau)$ produces a set Ω containing* all *the strings ω that satisfy the following:*

C1 $[\omega]_{iq,oq} = \tau$.

C2 $[\omega]_{id,od} \in g$.

C3 *ω is admissible (with respect to M').*

There are two sources of inefficiencies in $BF(g, \tau)$. First, we might maintain a large number of plausible sequences. Second, we need to examine the suitability of each candidate extension of each sequence with respect to the abstract specification. If there is no additional information about g, then there is no bound on the amount of computation required at each event. Suppose that N_{iq} input and N_{oq} output events have taken place. Then the amount of work done at each event is essentially the size of Ω at that point. The size of Ω is exponential in $N_{iq} + N_{oq}$, even in the absence of loss. To see this, notice that the number of placements of matching *id* tokens relative to a sequence of *iq* tokens is itself exponential in N_{iq}; possibility of loss factors in another N_{iq} binary choices between *id* and *il*. Therefore, even in the absence of loss, the computation that needs to be performed at each event grows with the number of events that have occurred. For an effective online procedure, we must arrive at a more efficient monitor process that bounds this per-event computation. We shall exploit properties of g that allow us to achieve this optimization.

5 Property Based Optimization

The algorithm given in Figure 3 exhibits space and time complexity that is exponential in the number of inputs and outputs in the trace. Clearly, for a long trace τ, the algorithm will quickly become intractable. However, some assumptions allow us to prune large sections of the brute-force search. In the following subsections, we examine properties of g, and assumptions on the input-output traces that allow effective online monitoring.

A notable feature that is common to many (but not all) of the algorithms presented in this section is that they simply buffer *iq* tokens as they appear, and take interesting action only when *oq* tokens are observed. This is in contrast to the brute-force algorithm, in which an iteration is performed after each *iq* event to consume the pending buffer in all possible ways. If this iteration is not performed, up to $B + c_U * c_L$ *iq* tokens must be buffered between outputs in the worst case, following Corollary 2. However, after an output, the maximum number of buffered *iq* tokens cannot be more than B.

5.1 No Loss

Each of the following sub-sections describes a class of g's that can be effectively monitored if we assume there is no loss between the monitor and the device, i.e. $c_L = 1$. This is an assumption of the network behavior, which when true, allows us to use very efficient algorithms for monitoring.

5.1.1 P1: Counting Properties

A very basic kind of property is a simple constraint on the number of inputs that must be consumed before an output is produced. If g specifies that every output must consume between $c_{\min}$ and $c_{\max}$ inputs without placing any additional constraints on the allowed sequences, then g satisfies:

$$\begin{aligned}\forall \alpha, \beta,\ od_x, od_y : \alpha \in g\ \wedge\ &(\alpha = \epsilon \vee \alpha \text{ ends in some } od)\\ &\wedge\ \beta \text{ has only } id\text{'s} ::\\ \alpha\ \beta\ od_y \in g\ &\Longleftrightarrow\ c_{\min} \le |\beta| \le c_{\max}\end{aligned}$$

That is, a string is in g if and only if it is constructed from another string that is also in g, which is itself empty or ends in an output event, by adding between $c_{\min}$ and $c_{\max}$ input events.

An algorithm for checking such g's in the presence of buffers is shown in Figure 4. This algorithm maintains two integers,

```
Constants. $c_{\max}, c_{\min}$ are integers.

Data Type. $buf_{\min}$ and $buf_{\max}$ are integers.
    $e$ is a boolean.
    Initially, $buf_{\min} = buf_{\max} = 0$ and $e$ = false.

Event Handlers. On receiving

        • $iq_x$:
          $buf_{\max} = buf_{\max} + 1,\ buf_{\min} = buf_{\min} + 1$
          if $(buf_{\min} > B + c_{\max})$ then $e$ = true
          else $buf_{\max} = \min(buf_{\max}, B + c_{\max})$
        • $oq_x$:
          if $(buf_{\max} < c_{\min})$ then $e$ = true
          else $buf_{\max} = \min(B, buf_{\max} - c_{\min})$,
               $buf_{\min} = \max(0, buf_{\min} - c_{\max})$

    If $e$ is true after executing either event handler, flag
    an error.
```

Figure 4: Algorithm for checking P1

$buf_{\min}$ and $buf_{\max}$, representing the minimum and maximum number of inputs that are currently buffered between the monitor and the device under test. If $buf_{\min}$ ever grows too large, it indicates that the particular iq, oq string seen so far could not be a valid execution without additional buffering between the monitor and the device. That is, too few outputs have been seen to account for all the inputs seen so far. Similarly, if $buf_{\max}$ ever becomes too small, it indicates that the particular iq, oq string seen so far could not reflect a valid execution because even if each output has consumed the minimum number of inputs, there have not been sufficient inputs to account for every output given that each output must consume at least $c_{\min}$ inputs. In each case an error flag e is set.

To prove the correctness of this algorithm, we can do a reduction from the brute-force algorithm of Figure 3 to the one in Figure 4. The reduction proceeds by defining a mapping between the two state spaces and showing that it is maintained when each algorithm takes a step in response to the same event. The proof is given in the appendix; we omit proofs of the remaining properties.

5.1.2 P2: Independent Inputs and Outputs

If g checks a composition of two independent properties, one of the input trace and the other of the output trace, then construction of the monitor can be greatly simplified. Formally, we say that:

$$([\omega]_{id} \in g \wedge [\omega]_{od} \in g) \Leftrightarrow [\omega]_{id,od} \in g$$

For example, suppose we want to check that as long as there is no loss and all data is acked, the TCP sender we are monitoring will keep pumping out new data. First we check that every output data segment has a sequence number strictly greater than that of the last output. However, this needs to be true only as long as all data is acked by the receiver. So in conjunction, we need to check that every input ack has a sequence number strictly greater than that of the last output. As long as both of these are true, the sender is behaving correctly. If the second property fails, our assumption has been violated, and the sender is assumed correct by default.

The monitoring algorithm for P2 simply needs to check that the sequence of id_x events corresponding to the iq_x events, as well as the sequence of od_x events corresponding to the oq_x events, is acceptable according to g. In addition, if we wish to monitor more complex properties, they can be checked independently, since P2 does not place any restrictions on the relative orderings of od's and id's. The procedure feeds one token to g at every event and needs to maintain no state extraneous to g. The algorithm is as shown in Figure 5.

```
Data Type. $\omega_i$ is a string of $id$'s.
    $\omega_o$ is a string of $od$'s.
    $e$ is a boolean.
    Initially, $\omega_i = \omega_o = \epsilon$ and $e$ = false.

Event Handlers. On receiving

        • $iq_x$:
          $\omega_i = \omega_i :: id_x$;
          if $\omega_i \notin g$ then $e$ = true;
        • $oq_x$:
          $\omega_o = \omega_o :: od_x$;
          if $\omega_o \notin g$ then $e$ = true;

    If $e$ is true after executing either event handler, flag
    an error.
```

Figure 5: Algorithm for checking P2

5.1.3 P3: Periodic Outputs

Suppose the specification requires that the device produces an output exactly every P inputs, we can have an efficient algorithm to parse a trace; no other assumptions on g are required. The condition can be given as follows:

$$\begin{gathered}\forall \alpha, \beta : \beta \text{ consists only of inputs} \wedge |\beta| > 0\ \wedge\\ (\alpha = \epsilon \vee \alpha \text{ ends in some } od) ::\\ (\alpha\ \beta\ od_y\ \in g) \Rightarrow |\beta| = P\end{gathered}$$

Many protocols have a periodic output behavior. For instance, ICMP echo protocol has an output period (P) of 1: every input must be responded to by an output. Some TCP implementations maintain a P of 2. P3 can be thought of a restricted counting property ($c_{\min} = c_{\max} = P$) in conjunction with a (possibly) complex property.

The algorithm we use is that any time an output event is seen by the monitor, it feeds exactly P buffered inputs to g, and then checks that the output is enabled at this point. The algorithm is as shown in Figure 6.

Constants. P is the number of inputs consumed by g before each output.

Data Type. ω is a string of *id*'s and *od*'s.
b is a string of *iq*'s.
e is a boolean.
Initially, $\omega = b = \epsilon$ and $e =$ false.

Event Handlers. On receiving

```
• $iq_x$:
  $b = b :: iq_x$;
  if $|b| > B + P$ then $e =$ true;
• $oq_x$:
  if $|b| < P$ then $e =$ true;
  else
    1. iterate $P$ times:
         delete the first element $iq_y$ from $b$;
         $\omega = \omega :: id_y$;
    2. $\omega = \omega :: od_x$;
       if $\omega \notin g$ then $e =$ true;
```

If e is true after executing either event handler, flag an error.

Figure 6: Algorithm for checking P3

5.1.4 P4: Deterministic Placement of Outputs

Suppose g has the property that the placement of outputs in a sequence of inputs has no *leeway*: there is exactly one position at which each output could be placed such that the resulting string is in g. Formally, the deterministic output placement property can be given as follows:

$$\begin{aligned}&\forall \alpha, \beta : \beta \text{ consists only of inputs } \wedge |\beta| > 0 :: \\ &\quad \alpha\ od_k \in g \Rightarrow \alpha\ \beta\ od_k \notin g\end{aligned}$$

In practice, we only need that $\alpha\ \beta\ od_k \notin g$ for $|\beta| < B$. A property in P4 is checked by maintaining a buffer of unconsumed inputs. On getting an output, we feed inputs from the buffer to g until the first point where the output is enabled, and then we feed the output to g. This works, because P4 guarantees that the output could not occur after any other input in the future. This algorithm maintains buffer b of inputs of size $(B + c_U)$. At each output we feed the first $|b| - B$ inputs to g, followed by at most B strings (of length at most B).

In general, finite-state machines do not satisfy P4. However, some do, and specific instances of other kinds of automata (PDA etc.) could satisfy P4 too. For example, the following grammar obeys P4:

$$\begin{aligned}&G \rightarrow \epsilon \\ &G \rightarrow G\ G \\ &G \rightarrow H\ o_{\text{STOP}} \\ &H \rightarrow i_{\text{A}} \\ &H \rightarrow i_{\text{LPAREN}}\ H\ i_{\text{RPAREN}}\end{aligned}$$

5.1.5 P5: Contiguously Enabled Commutative Outputs

Sometimes, there is a range of positions in a buffered input stream where one could reasonably place an output. Consider the following two restrictions: the range of positions over which the output is enabled is contiguous, in the sense that the input is enabled at every point in the input stream between the first and last positions it is enabled. Also, we require that the output *commute* with each of the inputs in this contiguous window, such that if the output is consumed and then the input, we arrive at a state which is indistinguishable from consuming the input first and then the output. These two conditions are embodied in the following formula:

$$\begin{aligned}&\forall \alpha, \beta : \beta \text{ consists only of inputs } :: \\ &\quad (\alpha\ \beta\ od_x \in g) \wedge (\alpha\ od_x\ \beta \in g) \Rightarrow \\ &\quad\quad (\forall \delta_1, \delta_2 : \delta_1\ \delta_2 = \beta :: \\ &\quad\quad\quad (\alpha\ \beta\ od_x \sim_g \alpha\ \delta_1\ od_x\ \delta_2))\end{aligned}$$

where

$$\alpha \sim_g \beta \equiv (\forall \gamma : \alpha\gamma \in g \iff \beta\gamma \in g)$$

An example of such a specification is TCP flow control. If a TCP ack od_x is allowed just before data segment id^k and just after id^{k+p}, then it must be allowed at all points in between, since none of them *matter* to the ack. Moreover, the receiver's window is the same whether this ack occurs before or after any of these data segments in the window $id^k \ldots id^{k+p}$.

For g in P5, we can work with a *credit* scheme. We always place output at the first place it is enabled, but remember the credit we get for not placing it at a later place. If at some point in the future, the buffer overflows because we consumed too few inputs, we can then use up this credit, i.e. eat up that many input tokens.

The algorithm for P5 (Figure 7) maintains a buffer of inputs of size $(B + c_U)$, and one integer indicating the credit: the range of positions where the last output could have taken place. At each output we feed the first $|b| - B$ inputs to g, followed by at most B strings (of length at most B).

5.1.6 P6: Output-checkpointed Automata

We say that g is an output-checkpointed automaton if, starting from a state, for each output od_x, there is at most one next state that g can be in. In terms of strings, if two strings α and β are equivalent with respect to g, and if they are concatenated with different strings of inputs followed by the same output od_x, then the two strings ending in od_x are still equivalent with respect to g. Formally,

$$\begin{aligned}&\forall \alpha, \beta : \\ &\quad (\alpha \sim_g \beta) \Rightarrow \\ &\quad\quad (\forall od_x, \delta_1, \delta_2 : \delta_1, \delta_2 \text{ consist only of inputs } :: \\ &\quad\quad\quad ((\alpha\ \delta_1\ od_x\ \in g) \wedge (\beta\ \delta_2\ od_x\ \in g)) \Rightarrow \\ &\quad\quad\quad (\alpha\ \delta_1\ od_x \sim_g \beta\ \delta_2\ od_x))\end{aligned}$$

An instance of P6 is a protocol in which each output gives complete information about the state in which it is enabled. For example, some transport layer protocols have a notion of selective acknowledgments (sack's) where data receivers send information about all data received up to that point. Although we need to check if the sack was allowed, once it

```
Data Type. $\omega, \omega'$ are strings.
    $b, b'$ are strings of inputs.
    $credit$ is an integer.
    $e$ is a boolean.
    Initially, $\omega = b = \epsilon$, $credit = 0$ and $e$ = false.

Event Handlers. On receiving

    • $iq_x$:
        $b = b :: iq_x$;
        if $(|b| > (B + c_U) \wedge credit > 0)$ then
            delete the first element $iq_y$ of $b$;
            $\omega = \omega :: id_y$;
            $credit = credit - 1$;
        if $(|b| > (B + c_U) \wedge credit = 0)$ then $e$ = true;
    • $oq_x$:
      1. repeat until
         $((|b| \leq B) \wedge (\omega :: od_x \in g \vee b = \epsilon))$:
             delete the first element $iq_y$ of $b$;
             $\omega = \omega :: id_y$;
      2. if $b \neq \epsilon$ then
             $b' = b$;
             delete the first element $iq_y$ of $b'$;
             $\omega' = \omega :: id_y$;
             $credit = 0$;
             repeat until $\omega' :: od_x \notin g$ or $b' = \epsilon$:
                 $credit = credit + 1$;
                 delete the first element $iq_y$ of $b'$;
                 $\omega' = \omega' :: id_y$;
      3. $\omega = \omega :: od_x$;
         if $\omega \notin g$ then $e$ = true;

    If $e$ is true after executing either event handler, flag
    an error.
```

Figure 7: Algorithm for checking P5

has happened, the state at the receiver is completely known to us. So we can carry out our analysis output to output, forgetting everything that occurred before the last output.

The algorithm for monitoring P6 maintains just one admissible string ω because after an output occurs, all strings allowed by g must be equivalent; we just need to keep one of them, and we keep the one with the first possible placement of that output. In addition, we maintain a buffer b of inputs of size $(B + c_U)$, and B bits indicating positions at which the last output could also have happened.

At each output, we need to concatenate every sub-string of buffered inputs, starting from these positions, to ω and check if the output is allowed after this string. There may be up to B^2 such input strings (of length at most $(B + c_U)$) that need to be checked. This gives us a bound for the per-event computation, polynomial in the length of the buffers.

5.1.7 P7: Finite State Machines

If g is known to be a finite state machine with a set of states Γ, then there is a bound to which the brute-force search set Ω can grow. Suppose $g = (\Sigma, \Gamma, \delta, s_0, Err)$, where Σ consists of all id,od symbols, Γ is the finite set of states, δ is the deterministic transition relation, s_0 is the initial state, and Err is the set of error states. Each (ω, b) in Ω can be represented as (s, b) where $s = \delta^*(s_0, \omega)$. Now, b must contain some suffix of the iq elements in ω, of length at most B. However, every ω in Ω must contain the same sub-sequence of iq's and oq's. Therefore, every b in Ω must be a suffix of the *same* B-element string. This gives us a bound of $|\Gamma| * B$ elements for Ω. This means that we may have to feed up to $|\Gamma| * B^2$ elements to g at each output. The bound is polynomial in the number of states and the size of the buffers. Moreover, it gives us a way of executing the general brute-force algorithm with an efficient representation of the string ω as the state s. The algorithm is as shown in Figure 8. Notice that, because we perform an iteration after an iq event, as in the brute-force algorithm, we cap the buffering requirement by B.

```
Constants. $g = (\Sigma, \Gamma, \delta, s_0, Err)$.

Data Type. $\Omega$ is a set of pairs (state, list of input sym-
    bols). Initially, $\Omega = \{(s_0, [])\}$

Event Handlers. On receiving

    • $iq_x$:
      1. $\forall (s, b) \in \Omega$:
             delete $(s, b)$ from $\Omega$;
             check-add $(s, b :: iq_x)$ to $\Omega$.
      2. iterate until no more additions to $\Omega$:
             $\forall (s, [iq_y :: b]) \in \Omega$:
                 check-add $(\delta(s, id_y), b)$ to $\Omega$;
    • $oq_x$:
      1. $\forall (s, b) \in \Omega$:
             delete $(s, b)$ from $\Omega$;
             check-add $(\delta(s, od_x), b)$ to $\Omega$.
      2. iterate until no more additions to $\Omega$:
             $\forall (s, [iq_y :: b]) \in \Omega$:
                 check-add $(\delta(s, id_y), b)$ to $\Omega$;

    where check-add adds $(s, b)$ to $\Omega$ if and only if:

    • $b$ obeys the buffering constraints $(|b| < B)$, and
    • $s$ is not an error state of $g$ $(s \notin Err)$.

    If $\Omega = \phi$ after executing either event handler, flag
    an error.
```

Figure 8: Algorithm for checking P7

5.2 Dealing with Loss

Allowing loss of input events between the monitor and the device introduces additional complexity to our search algorithm. However, as before, we can derive efficient algorithms for some classes of g. In each of the following sub-sections, we describe a property of g that allows us to monitor it effectively even in the presence of loss. We will see that the classes of g that were efficiently monitorable without loss, become less efficient or even unfeasible when loss is allowed.

5.2.1 Counting Properties (P1 revisited)

As mentioned before, properties based solely on the number of inputs and outputs consumed are often used to characterize protocols. For such g, monitoring in the presence of loss is very similar to the loss-less case. Suppose that we know that no more than c_L inputs can be lost in succession. Then in order to check that each output must consume between $c_{\min}$ and $c_{\max}$ inputs, we follow the same procedure of maintaining $buf_{\max}$ and $buf_{\min}$, except that we allow $buf_{\max}$ and $buf_{\min}$ to grow up to $m + c_{\max} * c_L * (n+1)$ during iq processing. We leave the oq processing unchanged. There is clearly a constant amount of work to be done at each event.

5.2.2 P2o: Independent Output Properties

Properties about the output stream can be checked even in the presence of loss because by assumption outputs never get lost. The checking procedure simply involves checking g against the trace seen at M. No buffering is needed.

An example of a P2o property is output monotonicity: every output contains a sequence number that is strictly greater than that of the last output. This just involves storing an output and comparing it with the next one. We feed g at most one token at every output.

5.2.3 P8: Insert-closed Commutative Outputs

We describe a class of monitorable properties that are impervious to the presence of input loss. Essentially, if a string ending in od_x is acceptable, so is the string with an arbitrary string of inputs δ inserted before od_x. Moreover, od_x commutes with every input in δ. In effect, this says that if g enables od_x at the end of the string α, it remains enabled forever, and it can be placed at any point after α with the same effect.

$$\begin{array}{l}(\forall \alpha, od_x : \alpha\ od_x \in g \Rightarrow \\ \quad ((\forall id_y : \alpha\ id_y\ od_x \in g) \wedge \\ \quad\ (\forall \delta_1, \delta_2 \text{ consisting only of inputs :} \\ \quad\quad \alpha\ od_x\ \delta_1\ \delta_2 \sim_g \alpha\ \delta_1\ od_x\ \delta_2)))\end{array}$$

Note that P8 is really a special case of P5. The fact that outputs once enabled are enabled forever implies the contiguously-enabled property, in addition to which we already have commutativity of outputs. An instance of P8 is a containment property on outputs. For example, a property could state that every output contains a field *bytes-received* that is less than the sum of the sizes of inputs received.

The algorithm for checking P8 is simple. Always assume that no inputs have been lost and place the output after the first input where it is enabled. To keep track of the succeeding positions where the output could have occurred, we maintain a credit as in the algorithm for P5. Suppose in reality, some of the inputs between two outputs were lost, then we have generated a string that has some extra inputs inserted between these outputs. However, P8 tells us that inserting these inputs still keeps the string in g. Next, suppose that an output that we placed after an input id^k in the input stream, did not in reality occur until id^{k+p}. Here again, P8 assures us that the output commutes with the inputs $id^k, id^{k+1} \ldots, id^{k+p}$. Finally, note that the string we construct is an admissible string: one in which no losses occur. Therefore, as long as this string belongs to g, we must accept it. The algorithm is as shown in Figure 9. This algorithm feeds g with at most $(B + c_U * c_L)$ input tokens at each output.

```
Data Type. $\omega$ is a string.
    $b$ is a strings of inputs.
    $credit$ is an integer.
    $e$ is a boolean.
    Initially, $\omega = b = \epsilon$, $credit = 0$ and $e$ = false.

Event Handlers. On receiving
    • $iq_x$:
      $b = b :: iq_x$;
      if $(|b| > (B + c_U * c_L) \wedge credit > 0)$ then
          delete the first element $iq_y$ of $b$;
          $\omega = \omega :: id_y$;
          $credit = credit - 1$;
      if $(|b| > (B + c_U * c_L) \wedge credit = 0)$ then
          $e$ = true;
    • $oq_x$:
      1. repeat until
           $((|b| \le B) \wedge (\omega :: od_x \in g \vee b = \epsilon))$:
             delete the first element $iq_y$ of $b$;
             $\omega = \omega :: id_y$;
      2. $\omega = \omega :: od_x$;
         if $\omega \notin g$ then $e$ = true;
      3. $credit = |b|$;
    If $e$ is true after executing either event handler, flag
    an error.
```

Figure 9: Algorithm for checking P8

5.2.4 Finite State Machines (P7 revisited)

If g is an FSM, we can bound the amount of state that we need in order to model loss in the brute-force search strategy. In fact, the bound on Ω is exactly the same as in the absence of loss. There can be at most $|\Gamma| * B$ elements in Ω. As before, this bound means that we can effectively execute the brute force algorithm to monitor g. However, in this case we have to consider all 2^B lossy substrings of the buffer at each event. Therefore, we may need to feed as many as $|\Gamma| * B^2 * 2^{B-1}$ tokens to g at each event.

5.2.5 P9: Deterministic Stateless Transducers

Let us assume that all the inputs that the monitor sees will be distinct. This restriction forbids the environment from producing duplicate inputs for the lifetime of the monitor. Under this assumption, we can describe a g that specifies a stateless transducing behavior. Suppose that an output od_y can occur after an input id_x. Then, in every string allowed by g that contains od_y, it must occur immediately after id_x. Moreover, any string ending in an $id_x\ od_y$ pair is equivalent to the string $id_x\ od_y$. This means that g is stateless: all it cares about are $id_x\ od_y$ pairs.

$$\begin{array}{l}\forall \alpha, id_x, od_y : \alpha \in g \Rightarrow \\ (id_x\ od_y \sim_g \alpha\ id_x\ od_y) \wedge \\ (id_x\ od_y \in g \Rightarrow \\ \quad (\forall id_z \neq id_x : id_z\ od_y \notin g) \wedge \\ \quad (od_y \notin g)\end{array}$$

The condition that each input is distinct is a rather strong one. In practice, what we need to impose is that no input is repeated within a space of $(B + c_U * c_L)$ iq tokens, which is the maximum number of inputs that the monitor needs to analyze at a time.

An example of P9 is the specification of the ICMP (*ping*) protocol. In a *ping* session, all inputs (ICMP Requests) are unique because they have monotonically increasing sequence numbers. Given an ICMP Reply, we can map it uniquely to an ICMP Request on the basis of the sequence number. Moreover, once we have mapped the Reply to a request, there is nothing that we need to remember about the analysis up to that point.

P9 can be checked efficiently even in the presence of loss. Essentially, each output od_y points to at most one input id_x that triggered it and we do not need to check any other input. If id_x is not in the buffer, and od_y cannot occur without an input, then there is an error, otherwise, everything up to id_x can be dropped from the buffer. This algorithm feeds g at most $(B + c_U * c_L)$ elements at every output. The algorithm is as shown in Figure 10.

```
Data Type. $\omega$ is a string.
    $b$ is a string of inputs.
    $e$ is a boolean.
    Initially, $\omega = b = \epsilon$ and $e =$ false.

Event Handlers. On receiving

    • $iq_x$:
      $b = b :: iq_x$;
      if $(|b| > (B + c_U * c_L))$ then $e =$ true;
    • $oq_x$:
      if $b = \epsilon$ then $e =$ true;
      else
          delete the first element $iq_y$ of $b$;
          repeat until
          $((|b| \leq B) \wedge (id_y :: od_x \in g \vee b = \epsilon))$:
              delete the first element $iq_y$ of $b$;
          $\omega = \omega :: id_y :: od_x$;
          if $\omega \notin g$ then $e =$ true;

    If $e$ is true after executing either event handler, flag
    an error.
```

Figure 10: Algorithm for checking P9

5.2.6 P10: Output-checkpointed Stateful Transducers

In P9, the statelessness assumption allowed us to forget about the history of the monitoring, once an output had been placed in the input stream. We can generalize this notion, by taking a hint from output-checkpointed automata (P6). Instead of saying that the state of g after consuming $\alpha\ id_y\ od_x$ must be the same as the state after consuming just $id_y\ od_x$, we say that starting from a state, if od_x is allowed after some inputs then there is a unique state that g can be in after consuming od_x.

As before, all inputs in τ must be distinct. Formally, g is expressed as a conjunction of P6 with a generalization of P9:

$$\begin{array}{l}\forall \alpha, \beta : \\ \ (\alpha \sim_g \beta) \Rightarrow \\ \quad (\forall od_x, \delta_1, \delta_2 : \delta_1, \delta_2 \text{ consist only of inputs} :: \\ \quad\ ((\alpha\ \delta_1\ od_x\ \in g) \wedge (\beta\ \delta_2\ od_x\ \in g)) \Rightarrow \\ \quad\quad (\alpha\ \delta_1\ od_x \sim_g \beta\ \delta_2\ od_x)) \wedge \\ \forall \alpha, id_x, od_y, \delta_1, \delta_2 : \delta_1, \delta_2 \text{ consist only of inputs} :: \\ \quad (\alpha\ \delta_1\ id_x\ od_y \in g) \Rightarrow \\ \quad\ (\forall id_z \neq id_x : \alpha\ \delta_2\ id_z\ od_y \notin g) \wedge \\ \quad\ (\alpha\ od_y \notin g)\end{array}$$

An example of such a g is a protocol in which there are two kinds of inputs: data inputs containing integers and ack-request inputs that contain unique magic numbers. The protocol is supposed to send an output when it receives an ack-request, and the output contains the magic number of the request and a sum of all inputs seen since the last output. Clearly, every output has a unique input (the ack-request) after which it can occur. Moreover, it contains information that is compatible only with some sub-sequences of inputs since the last output.

The P6 part of P10 allows us to just maintain one state after each output. Moreover, the P9 part tells us that we can uniquely place the output in the input stream. In conjunction, this gives us a simple algorithm for checking P10. At each output, we place it in the sequence of buffered inputs as in P9. Once we have found a position, we must search for a lossy sub-sequence of the inputs before that position that makes the string acceptable to g. If both these tests succeed, we can proceed with a single next state and a single buffer to monitor the rest of the string.

The algorithm maintains a input buffer b of size $(B + c_U * c_L)$. Then at each output, we may need to check g against $2^{(B+c_U*c_L)}$ input sequences. The maximum number of tokens to be fed to g is $(B + c_U * c_L) * 2^{(B+c_U*c_L)-1}$.

5.2.7 Output-checkpointed Automata (P6 revisited)

P6 can be handled in the presence of loss by doing some further exploration of buffer strings. However, no exploration of g's state space needs to be done. We maintain a set of positions in the buffer where the last output could have happened. Then for the next output, we need to find all sub-sequences starting from these positions that enable the output. We are guaranteed that all these sub-sequences will end in the same state, so we will just need to remember the positions where we can place the output and the common next state. If there is no position where we can place the output, there is an error. This algorithm would require us to check a maximum of $2^{(B+c_U*c_L)}$ input strings against g at each output, where $(B + c_U * c_L)$ is the size of the input buffer. We may have to feed a maximum of $(B + c_U * c_L) * 2^{(B+c_U*c_L)-1}$ tokens to g.

5.3 Property Relationships

In this section, we describe the relationships between all the classes of g that we have discussed in this section. In Table 1, we summarize the complexities of the algorithms described for the various property classes. We assume that g has been written in a way that allows us to feed it input and output tokens one by one, and it will take a constant amount of time to analyze each token. Then the complexity of the monitoring is the number of tokens that need to be fed to g at any step. Note that most of the complexities are in terms of the size of the buffer B. The value of B is dependent on the sizes of the input (m) and output (n) buffers, protocol entities like the maximum number of inputs allowed between two outputs (c_U), and environmental entities like the maximum number of inputs that can be lost in succession (c_L), between the monitor and the system. We define the buffer size in the absence of loss as $B = m + c_U * n$, and in the presence of loss as $B = m + c_U * c_L * n$.

Table 1: Monitoring Complexities for Property Classes of g

Property	Without loss	With loss
All	Unbounded	Unbounded
P7	$\Gamma * B^2$	$\Gamma * B^2 * 2^B$
P6	B^3	$B * 2^B$
P5	B^2	Unbounded
P4	B^2	Unbounded
P8	B^2	B^2
P3	P	Unbounded
P10	B	$B * 2^B$
P9	B	B
P2	1	Unbounded
P2o	1	1
P1	1 (just count)	1

As Table 1 shows, most of the properties described in this section belong to different complexity classes for monitoring. Many of the properties are in fact inclusions of each other. The inclusion relationships are as shown in Figure 11. However, there are some other interesting relationships as well. In going from the no-loss to the lossy case, properties P3, P4 and P5 become very inefficient to monitor. Therefore, we need to pick finer subsets of P5: P8, P9 and P10, that will still lend themselves to efficient monitoring in the presence of loss. These subsets collapse to similar complexities as P3, P4 and P5 in the no-loss case.

Properties P5, P6, and P7 are three orthogonal ways of reigning in the unbounded size of Ω in the brute-force search. It is possible to combine some of these properties to arrive at a property that relates to a real-world protocol. For example, we combine P6 and P7 in the next section to arrive at a TCP property. It may also be possible to combine one of more of these properties with some protocol-specific knowledge to reduce the size of Ω. Therefore, if a protocol property that one wishes to monitor does not fall in one of P5, P6, or P7, it does not immediately rule out an efficient procedure, contrary to what seems to be the case by looking at Figure 11 ("All" category). Whether there are other useful protocol-independent properties that are orthogonal to the three above, is an open question, one we hope to address in future work.

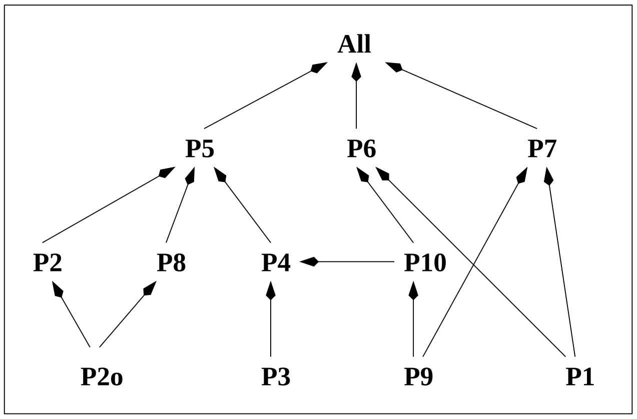

Figure 11: Relationships between Properties of g. An arrow from property Q to property R means that Q implies R.

6 Monitoring TCP

We now show how to use the techniques described in the previous sections in order to monitor TCP [15] implementations. For reasons of space, we consider only the receiver-side behavior.

TCP implements a "sliding window" protocol to implement reliable, in-order delivery. To transfer a large buffer of data, a TCP sender breaks it into small fixed-sized segments and sends each numbered segment in a separate message. Initially, the sender assumes that the receiver is ready for the first segment, and that the receiver can accept a certain count of segments in its "window". It may send some of these segments out on the network to the receiver. The receiver acknowledges (acks) these messages, with a sequence number of the segment immediately after the contiguously received part of the buffer. Thus, if segments 1, 2, 4, and 5 have been received, where segment 3 was delayed or lost in transit, TCP receiver may generate acks 2, 3, 3, and 3. The sender forms a judgment of which segments (e.g. 3) got lost in transit, and resends them. If the receiver now receives segment 3, it generates an ack with sequence number 6, because segments 1-5 have been all received. Occasionally, the receiver also gives an indication of newly available capacity in its window, once some prefix of the earlier contiguous packets are consumed by the receiving application.

The TCP specification prescribes the sequence number that must be contained in an ack, based on the state of the receiver window as described above. It, however, allows a leeway that receivers may generate an ack at least every two messages, so an implementation may decide to not ack every single message.

Typically, we may wish to monitor TCP properties such as the following:

1. The implementation is generating an ack for at least every other message it receives.
2. Ack sequence numbers are non-decreasing.
3. Acks always acknowledge exactly the contiguously received set of segments.

Each such property falls in a different class of g, and hence entails different complexities in the monitoring process. Consider data segments as input symbols and acks as output symbols.

- TCP property 1 describes a g that is a counting property, $g \in P1$, with $c_{\min} = 1$ and $c_{\max} = 2$. This property is easy to monitor both in the presence and absence of loss.
- TCP property 2 describes a g in which inputs and outputs are independent. Since $g \in P2o$, even in the presence of loss this property is easy to check. Moreover, property 2 can be checked independently of the other properties.
- TCP property 3 describes a g that falls in class P5. In the absence of loss between the monitor and device, we can check this efficiently using the algorithm in Figure 7. Interestingly, we can efficiently check property 1 in conjunction with property 3, because the algorithms for P1 and P5 compose well. Both define ranges over which the last output could have taken place, the algorithm for the conjunction simply maintains the intersection for these ranges.
- The algorithm described above does not work if there is loss between the monitor and device. However, property 3 can also be expressed as the composition of an output-checkpointed property (P6) and a finite state property (P7). In order to see this, note that the TCP monitor's state could be seen as a triple of the form (unacked-data, contig-recd, recv-window), where *unacked-data* is the number of unacknowledged data packets since the last ack, *contig-recd* is the largest sequence number in the contiguously received data, and *recv-window* is the data that has been received in the window but is missing some segments. Here (unacked-data,contig-recd) is output-checkpointed: when you see an ack for sequence number $s + 1$, unacked-data must be 0, and contig-recd must be s. On the other hand, the recv-window is finite-state, and its state space can be as large as 2^{W-1}, where W is the window size. So to monitor property 3 (together with property 1) we can compose the algorithms for P6 and P7. The complexity of such an algorithm is the sum of the complexities of the two algorithms as shown in Table 1, which is essentially the complexity of the P7 part ($2^W * B^2 * 2^B$).

7 Conclusions and Related Work

This paper is most closely related to work on passive testing [7]. They also use on-line monitoring to test protocol implementations, but do not address the problems of infidelity of traces. To our knowledge this paper is the first attempt to create an abstract model of the infidelities that can arise when monitoring a remote device under test, and to formally specify conditions under which efficient correctness-checking algorithms can be deployed.

Paxson [13] gives a good empirical overview of sources and manifestations of infidelity, with a focus on wide-area networks. Our work concentrates on infidelity in a local-area network setting and is a more formal treatment.

Most of the other literature devoted to network monitoring is targeted toward network security concerns. Paxson's Bro system [12] and the Network Flight Recorder [14] are good examples of practical monitoring tools that must deal with remote monitoring of trusted and untrusted devices. These tools offer high performance and customizability with the addition of new specification scripts, but such scripts are developed in an ad-hoc manner and must take infidelity into account by hand.

Testing of reactive systems from abstract specifications is also an area of active research. Bhargavan et al. [3] give an overview of networking-related analysis techniques. Formal specification and automatic verification have also been used in the context of telecommunications systems [6, 4]. Our work differs from these in that we are not attempting to generate test cases but rather to monitor the correctness of an implementation during deployment. O'Malley et al. [10] discuss the creation of property checking automata from a graphical specification notation. Event specifications [8] have been used for run-time property verification, and the Verisim tool [2] applies this work to properties of network protocols. However, all of these systems assume a co-located style of specification—that is, they assume that the device under test can be directly instrumented and properties can be checked in a synchronous manner. If the device under test cannot be directly instrumented then the specifications used must be transformed to take the resulting infidelity into account. This paper has been a systematic exploration of such transformations.

References

[1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[2] Karthikeyan Bhargavan, Carl A. Gunter, Moonjoo Kim, Insup Lee, Davor Obradovic, Oleg Sokolsky, and Mahesh Viswanathan. Verisim: Formal analysis of network simulations, August 2000. To appear in: International Symposium on Software Testing and Analysis.

[3] Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic. A taxonomy of logical network analysis techniques. Technical Report MS-CIS-00-14, University of Pennsylvania, 2000.

[4] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, January 1997.

[5] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[6] L. J. Jagadeesan, A. Porter, C. Puchol, J. C. Ramming, and L.G.Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the International Conference on Software Engineering*, May 1997.

[7] D. Lee, K. Sabnani, A. Netravali, B. Sugla, and A. John. Passive testing and its applications to network management. In *Proceedings of the International Conference on Network Protocols*, 1997.

[8] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M.Viswanathan. Runtime assurance based on formal

specifications. In *Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[9] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quaterly*, 2(3):219–246, 1989.

[10] T.O. O'Malley, D.J. Richardson, and L.K. Dillon. Efficient specification-based test oracles. In *Second California Software Symposium (CSS'96)*, April 1996.

[11] V. Paxson. Automated packet trace analysis of TCP implementations. *Computer Communication Review*, 27(4), October 1997.

[12] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, December 1999.

[13] V. Paxson. End-to-end internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[14] M.J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a generalized tool for network monitoring. In *Proceedings of the Eleventh Systems Administration Conference (LISA XI)*, pages 1–8, 1997.

[15] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, Reading, Massachusetts, 1994.

Appendix: Proof for P1

Definition 2 (iqtail) *Let* iqtail(ω) *equal the sequence of iq elements in ω that are prior to the last oq element but which were not consumed before the oq, that is, which do not have a corresponding id element before the oq, plus all iq's that appear after the last oq element, regardless of whether the corresponding id is in ω. If ω does not contain any oq's, then* iqtail(ω) *is just the sequence of iq's in ω.*

Definition 3 (idtail) *Let* idtail(ω) *equal the sequence of id elements in ω that appear after the last od element. If ω does not contain any od's, then* idtail(ω) *is just the sequence of id's in ω.*

State Space Mapping. The mapping is given by the following inductive hypothesis:

$$\begin{array}{ll} & \neg\, e \Longrightarrow (\forall i :: \mathit{buf}_{\min} \le i \le \mathit{buf}_{\max} \Longleftrightarrow \\ & \qquad (\exists(\omega, b) \in \Omega :: |\mathrm{iqtail}(\omega)| = i)) \\ \wedge & \quad e \Longleftrightarrow \Omega = \phi \end{array}$$

Base Case. Initially, Ω contains only one element, which has a zero length ω and a zero length b. The initial conditions for $\mathit{buf}_{\max}$, $\mathit{buf}_{\min}$, and e are all consistent with the state space mapping.

Inductive Step. Consider the input actions in each algorithm corresponding to *iq* events. In Figure 3, this action deletes each member of Ω and adds a new member with the *iq* action added to both ω and b, as long as ω is admissible and its projection $[\omega]_{id,od}$ is in g. In Figure 4, an input event increments both $\mathit{buf}_{\min}$ and $\mathit{buf}_{\max}$, checks for an error condition, and then adjusts $\mathit{buf}_{\max}$ if necessary. We need to show that the state space mapping still holds. First, note that if any $(\omega, b) \in \Omega$ has an ω such that $|\mathrm{iqtail}(\omega)| >= B + c_{\max}$, it will be deleted from Ω because the result of adding the *iq* to ω will be inadmissible according to M', which recall is equal to $M(S, m+c*n, 0)$. In this case $c = c_{\max}$ and $B = m+c*n$. ω will be inadmissible because at most $c_{\max}$ *id*'s can appear after the last *od*, *oq* by way of property P1. The remaining B *iq*'s fill up the buffer and adding one more causes it to overflow, removing the resulting string from consideration.

If $\mathit{buf}_{\min} >= B + c_{\max}$ before the input action, this implies (by the inductive hypothesis) that every $(\omega, b) \in \Omega$ has a iqtail(ω) whose size is at least $B + c_{\max}$. When the *iq* event is added, all will be deleted and Ω will be empty. This corresponds to setting e to *true* in Figure 4. If $\mathit{buf}_{\min} < B + c_{\max}$ before the input action, then there is at least one $(\omega, b) \in \Omega$ such that $|\mathrm{iqtail}(\omega)| = i$ for every i between $\mathit{buf}_{\min}$ and $\mathit{buf}_{\max}$. Each will be removed and replaced with an (ω, b) pair with exactly one *iq* event added to each of ω and b, except for those that become inadmissible with respect to M'. This corresponds exactly to $\mathit{buf}_{\min}$ and $\mathit{buf}_{\max}$ being incremented by one, and $\mathit{buf}_{\max}$ being capped at $B + c_{\max}$ in Figure 4. The iteration step 2 in Figure 3 has no effect on the mapping because it only adds *id* events to ω and does not modify iqtail(ω). Note, however, that this iteration guarantees that every pattern of idtail(ω) such that $0 \le |\mathrm{idtail}(\omega)| \le \min(c_{\max}, \mathit{buf}_{\max})$ will be generated. This is because each element of the iqtail(ω) will be converted to an *id* and check-added to Ω. Those with more than $c_{\max}$ *id*'s will be discarded, since they are not in g, but all others will be accepted due to property P1. The longest iqtail has length $\mathit{buf}_{\max}$, leading to the upper bound.

Next, consider the output actions in each algorithm corresponding to the *oq* events. In Figure 3, this action first deletes every element of Ω, replacing each with one that has the *od*, *oq* event pair added. Then the same iteration discussed previously is repeated. Note that if any (ω, b) has an idtail(ω) $< c_{\min}$, it will be deleted because the result of adding the *od* will not be in g, according to P1. In Figure 4, if $\mathit{buf}_{\max} < c_{\min}$, then all idtail($\omega$) must have length less than $c_{\min}$, because idtail(ω) $<$ iqtail(ω) $< \mathit{buf}_{\max}$. Ω thus becomes empty after the first step. This corresponds to setting e to true in Figure 4. Otherwise, Figure 4 decrements $\mathit{buf}_{\max}$ by $c_{\min}$, but caps $\mathit{buf}_{\max}$ at B because even though iqtail can grow larger than B, the new iqtail will be at most B because it consists of only the un-consumed *iq* events in ω, of which there can be at most B. The algorithm also decrements $\mathit{buf}_{\min}$ by $c_{\max}$, setting it to zero if it goes negative. This corresponds to the fact that pairs (ω, b) will be deleted from Ω if they violate the $c_{\min}$ and $c_{\max}$ limits given by g, which means that the new minimum and maximum values of iqtail(ω) will be given when the maximum number of inputs are consumed from the minimum previous *iqtail* and the maximum number of inputs are consumed from the maximum previous *iqtail*. These consumptions are justified because we have a guarantee that every pattern idtail(ω), such that $0 \le |\mathrm{idtail}(\omega)| \le \min(c_{\max}, \mathit{buf}_{\max})$, will have been generated at the end of the previous *iq* or *oq* event.