Formal Methods in System Design, 27, 213–251, 2005

Network Event Recognition

KARTHIKEYAN BHARGAVAN bkarthik@seas.upenn.edu
Microsoft Research, Cambridge

CARL A. GUNTER gunter@cis.upenn.edu
University of Illinois, Urbana-Champaign

Abstract. Network protocols can be tested by capturing communication packets, assembling them into the high-level events, and comparing these to a finite state machine that describes the protocol standard. This process, which we call *Network Event Recognition (NER)*, faces a number of challenges only partially addressed by existing systems. These include the ability to provide precise *conformance* with specifications, achieve adequate *performance*, admit analysis of the *correctness* of recognizers, provide useful *diagnostics* to enable the analysis of errors, and provide reasonable *fidelity* by distinguishing application errors from network errors. We introduce a special-purpose *Network Event Recognition Language (NERL)* and associated tools to address these issues. We validate the design using case studies on protocols at application and transport layers. These studies show that our system can efficiently find errors in recognizers and implementations of widely deployed protocols; they also demonstrate how improved diagnostics and transformations can substantially improve understanding of information generated by packet traces.

Keywords: Network monitoring, protocol verification, correctness of implementations, languages for network protocols, formal analysis, network event recognition, NERL

1. Introduction

Internet applications, such as email or the World Wide Web, consist of several pieces of software executing on different sites and communicating by sending packets over an unreliable network. One way to analyze such applications is to observe the packets they send over the network and analyze their conformance to specified behavior. The observer, known as a *passive monitor*, captures packets from a communication link and assembles them into events described in the specifications of the communicating systems, a process called *Network Event Recognition (NER)*. This approach to testing has an number of advantages. For instance, no instrumentation of the implementations is required, and the monitored systems need not be affected by (or even know about) the monitoring. On the other hand, NER faces at least five key problems. First, there is the practical challenge of uncovering interesting *implementation conformance* problems in this way. Second, the *performance* of NER needs to reasonably match the amount of packet data being sent in the network. Third, since adequate performance is often achieved by creating recognizers that abstract away from unimportant aspects of the standard specification, there is a challenge in establishing *recognizer correctness*. Fourth, the large number of events creates a need for good *diagnostics* that can help to find the cause of an error. Fifth, observing packets on an unreliable network link must address *buffer fidelity* issues that result in large numbers of false positives.

The aim of this paper is to describe techniques to address these problems using a special-purpose language we call the *Network Event Recognition Language (NERL)*, that can be used to describe the specifications of network protocols from their representations as finite state machines. A NERL program is compiled into a C program that generates a packet monitor for a specific protocol. To be useful as a passive network monitoring system, NERL supports a number of basic requirements, like the ability to represent and process packet formats, model protocol layers, and interoperate with standard libraries for packet capture and processing. It also offers a clean semantics and modular design. A collection of NERL analysis tools and language features are used to address the key problems with conformance, performance, correctness, diagnostics, and fidelity. We illustrate these tools and features in two case studies.

Our first study is for the Simple Mail Transfer Protocol (SMTP) [30]. To demonstrate the effectiveness of NERL in testing, we implement a recognizer to determine conformance of implementations with the SMTP standard specification. This entails writing a layered set of NERL recognizers. We use this recognizer stack to find various errors in widely-used implementations of SMTP. In particular, we demonstrate errors in the implementation of HELO commands in versions of Sendmail, Postfix, and Exim, including 3 previously unknown errors. We use this study to illustrate NERL support for diagnostics. This support reveals control dependencies in a packet trace as determined by the semantics of the application being monitored. This allows the search for the cause of an error to be limited to a much-reduced subset of the packets monitored. The SMTP recognizer stack was shown to handle 150 concurrent SMTP sessions and up to 8400 packets per second with a peak memory requirement of only 12.8 KB per session. We also consider the problem of recognizer correctness with respect to SMTP error messages. We apply our approach, which is based on the Spin model checker, to our recognizer as well as *Bro* and *Altivore*, two other systems for which we found SMTP recognizers. We checked that our NERL recognizer is correct by analyzing about 2300 states and 19,000 transitions. The same technique finds a minor error in the Bro recognizer and significant errors in the Altivore recognizer, producing counterexamples for each violation.

Our second study is for the Transfer Control Protocol (TCP) [16]. Monitoring for protocols like SMTP that run above a reliable transport layer gains a significant benefit from the fact that high level events can be determined from a reconstructed TCP session. This allows the analysis to avoid infidelities arising from incomplete knowledge that the monitor has about network buffers. A naive approach to monitoring at the lower, unreliable layer yields many false positives caused by apparent errors in the implementations that are actually due to these infidelities. Our study of TCP uses a heuristic to transform the formal monitor to account for possible buffer affects. We apply this to study TCP traces generated from Linux, Windows, and Solaris machines to determine whether they conform to the standard specification for ACKs. When we first run a NERL monitor for TCP ACKs on a trace we get numerous false reports of errors. When we apply our transformation these false positives are reduced and we find some (previously known) deviations from the TCP standard in which ACKs are delayed. This demonstrates how to use NERL to implement our earlier work [4] on algorithms for reducing false positives arising from network infidelities. This approach is shown to be practical and useful, but we do also get a small number of false negatives in cases where the transformation masks an actual error.

After this introduction we begin with an overview of the requirements for NER, elaborating on the five primary challenges listed above, and discussing some of the systems that could be used to write, analyze, and use network recognizers. In the third section we introduce our language NERL and its associated tools, and discuss its performance. In the fourth and fifth sections we carry our our case studies for SMTP and TCP respectively. These studies show how NERL and its tools address conformance, performance, correctness, diagnostics, and infidelity. We end with a brief conclusion and acknowledgments. Appendices A and B provide enough background on TCP and SMTP respectively to follow our discussion in the case studies.

2. Requirements and related work

The goal of a network event recognizer is to monitor a protocol implementation and check that it conforms to its specification. A typical monitoring configuration is depicted in figure 1. The device under test running the protocol implementation (P), and the passive monitor running the recognizer (R), are separate machines communicating over the Internet. A common case is a co-networked configuration in which the passive monitor is on the same LAN as one or both of the monitored devices. The passive monitor captures network-layer packets sent and received by the device under test. The recognizer R then reconstructs the message events that must have occurred at P and checks them for conformance with the specification S.

If P operates directly on the network layer then the captured packets immediately yield the protocol messages. For higher-layer protocols, however, the messages need to be

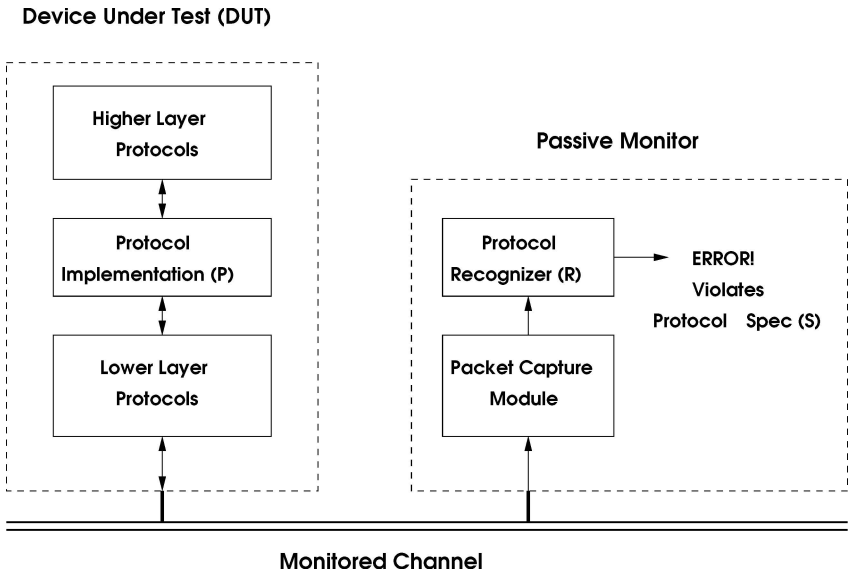

Figure 1. Passive protocol monitoring.

reconstructed layer-by-layer, and we need protocol recognizers for every layer up to P. This discussion already indicates that protocol monitoring has requirements that are different from run-time monitoring of other software. In the rest of this section, we elaborate on some of these requirements and survey previous approaches to satisfy them.

2.1. Conformance

A good protocol standard precisely defines the state machines executed by the participants and the format of the messages exchanged by them. Each participant is expected to conform to its protocol state machine at run-time. In practice, we are only interested in some important fragment, S, of this state machine. So, the first and obvious requirement for the recognizer R is the ability to detect violations of the specification S.

At the very least, programming a recognizer requires the ability to encode a protocol state machine and represent input and output events. While this requirement is fairly standard for run-time monitoring, it means, for instance, that a pure logic such as Linear Temporal Logic (LTL) [21] is probably not well suited to NER, while an Extended Finite State Machine (EFSM) based language such as Promela [13] is better suited.

In addition, we require the ability to represent packet formats and protocol layers.

2.1.1. Packet format representation. In the network monitoring context, the interface between the device under test and the monitor is based on packet transmissions. Packets on the network have a number of fields, such as the source and destination, and the format of these fields is particular to the protocol under analysis. For example, a TCP packet is an IP Packet in which the payload field is *overlayed* (replaced) by another record with fields corresponding to the TCP header and the TCP data. In turn, and SMTP packet is a TCP packet with the TCP data overlayed by the SMTP format, and so on.

Therefore, in order to extract and reconstruct the messages sent and received by a protocol, a network event recognizer must first capture and understand packet events. For a natural representation of packet events, a NER language must allow each event to have attached with it a record of attribute fields that mimics the packet format. This requirement is specific to protocol monitoring; many runtime verification tasks do not need to deal with such low-level structures and formats.

Our approach is to represent packet formats by a C-struct-like data structure. We note, however, that structs are often not expressive enough to precisely represent packet structure. In general, a language for NER would require a dependent type system such as PacketTypes [22].

2.1.2. Layering and Modularity. Network protocols are typically designed in layers. For instance, SMTP runs over TCP, which runs over IP; emails sent by users are broken up into SMTP commands, which are in turn transmitted as TCP segments inside IP packets. So, when an SMTP monitor captures these IP packets from the network, it then needs to reconstruct TCP streams and extract SMTP emails from these streams. A network event recognizer must therefore perform several layers of protocol analysis, up to the layer of interest.

This layering can be implemented either within a monolithic recognizer for the entire stack, or by writing separate monitoring *modules* for each protocol, which are then stacked one above another in the appropriate order. A modular design is clearly preferable. It parallels the protocol design, making protocol recognizers easier to write. Moreover, it enables several interchangeable versions of a recognizer, one for each property of interest. Modularity is, therefore, a key requirement for recognizer design.

2.2. Performance

Apart from programmability, another essential requirement for practical network event recognition is fast execution. When NER is used for live network monitoring, it has to keep up with real network speeds, which can involve data transfer of 100 Mbps or more. If we assume that most packets are more than a kilobyte in size, NER still needs to process tens of thousands of packets (primitive events) every second. This makes a fast execution environment necessary. For instance, when designing an earlier protocol monitoring system, Verisim [3], we found that our Java byte-code implementation, while adequate for monitoring 5 nodes at the same time, was too slow for analyzing the large amount of traffic generated by a 50 node network simulation.

While efficiency is a general goal for every aspect of recognizer design, there are two specific requirements that are mandated by it. The first is the ability to represent packet formats, which we have already discussed. The second is the ability to use existing packet and string parsing libraries. A network event recognizer must interface with low-level infrastructure that captures and presents packet traces to it. Therefore, a pragmatic requirement is that a network event recognizer must be compatible with these support libraries.

Commonly-used network libraries are typically written in C. For instance, the libpcap library (url tcpdump.org), implements functions that can capture packets from a variety of network interfaces, filter them based on protocols and devices of interest, and parse them into packet events in C. It would be wasteful to reimplement this functionality as part of the protocol recognizer. Similarly, to parse high-level messages, such as emails, it is most convenient to use existing parsers such as Yacc/Lex, that are also written in C. Writing efficient recognizers, therefore, requires the ability to convenently interface with these C libraries.

2.3. Correctness

Recall that the recognizer R is written to check the packet trace generated by P for conformance against some fragment, S, of the protocol specification. So far, the relationships between R, S, and the protocol specification have been left informal. Ideally, we would like to guarantee that R raises an error only if P has violated the protocol specification. In other words, there are no *false positives*. Moreover, we would like to guarantee that there are no false negatives: R raises an error whenever P violates S.

There are two main reasons why R may fail these guarantees. First, the abstraction used to derive the fragment S from the full protocol specification may be incorrect. Second, the

specification S may be non-deterministic, and the derivation of a deterministic recognizer R may be incorrect.

One way to address the second problem is to use the protocol specification directly as an executable recognizer. Indeed, this approach has been used for specifications written using formal description techniques such as Lotos and Estelle [6, 10]. However, if S is non-deterministic, using it as a recognizer becomes infeasible because it entails an expensive state-space search [10]. Moreover, this technique still does not solve the first problem of proving that the abstraction used to derive S is correct.

Instead, we advocate model-checking the recognizer R against the protocol specification and against S to verify that R does not produce any false positives or false negatives. For instance, given correct state machine models for all protocol participants (derived from the specification), and the state machine for the recognizer, a model-checker can be used to verify that the recognizer never produces an error.

The model-checking requirement entails writing the recognizer in a restricted language with a clean semantics. Programs written in a general-purpose programming language like C have proved difficult to check, but there have been successful attempts to model-check stylized programs in subsets of these languages [14].

2.4. Fidelity

Monitoring a protocol implementation P requires that the complete and correct trace of inputs and outputs to P be available. Often, however, the trace available (or observed) at R is incomplete and inaccurate, because packets get dropped and re-ordered between the device under test and the monitor [4]. This can cause the protocol recognizer to get desynchronized with the protocol implementation, resulting in false positives. Indeed, the inability to counter such *trace infidelities* has hampered earlier attempts to test TCP implementations [25], and has been identified as a significant vulnerability of intrusion detection systems [31].

For instance, suppose a recognizer R assumes that it will see all the IP packets at P. If some packets are dropped, and others are not captured in the right order, the protocol state R no longer matches the protocol state at P. So, even if P is in a valid state, R may raise an error. Moreover, R is then incapable of reconstructing events for higher-layer protocols. However, in some cases, R can recover from these trace infidelities. Suppose P sends an acknowledgement for every packet it receives. Then, when this acknowledgment is seen, the monitor can guess that the packet was not dropped. Using this information, the monitor can reconstruct portions of the correct packet trace, which might be adequate to find errors in P.

Therefore, to account for packet trace inaccuracies, a network event recognizer needs to recover the correct packet trace from the observed trace. In [4] we proposed a general tranformation algorithm that takes a recognizer that is unaware of packet trace infidelities and generates a recognizer that accounts for them. Our approach is to employ this transformation algorithm for transport-layer protocol event recognition. We note that carrying out such a program transformation for a general-purpose programming language would be quite difficult, because the transformation would need to take into account all the constructs and idiosyncracies of the language.

2.5. *Diagnostics*

Useful diagnostics are essential for a network monitoring system. A typical packet trace may contain several million packet events. So when an error is discovered at time *T*, it is insufficient to simply indicate that 'something went wrong', expecting the user to go through the packet trace backward starting at T and attempt to find out why the error occurred. While this problem occurs in all run-time verification systems, it is more pronounced in network monitoring because we do not have the option of instrumenting the monitored software to produce more information for the input events; we are bound by standardized packet formats.

One option is to use event attributes to provide more information to help interpret the error. For instance, when an SMTP error is detected, we may produce an SMTPErr event that contains a string attribute explaining the error. But this mechanism relies on the programmer providing custom error information, and the user's willingness to read it. Instead, we advocate diagnostic mechanisms that enable the user to trace the causal history of any event, much like a debugger in reverse. Our approach is to automatically produce the sequence of packets that caused the recognizer to produce an output event. This sequence is, in effect, a *dynamic slice* of the packet trace [5, 34] and can be generated for any event with no additional work by the programmer or the user.

2.6. *Related work*

Event recognition and analysis has been studied extensively in several contexts, including network monitoring. The earliest application of runtime verification to network protocols that we can find in literature is the Overseer [11], a proposed monitoring system for a pre-Internet network. If we restrict our attention to tools and languages that have been used analyzing Internet protocols, we can identify five lines of research. We consider each of these in order and pick a representative framework. Then, we examine whether any of these frameworks satisfies the requirements of network event recognition.

Internet protocol implementations are typically written in C. Several specialized languages, such as Prolac [18], and architectures, such as Ensemble [35], have been developed to ease the task of programming protocols efficiently and correctly. Still, none of these specialized languages have been directly used for protocol monitoring, while several protocol-specific monitors have been written in C, such as tcpanaly [25].

Formal description techniques, such as Lotos [20], have been used to specify protocols and to generate test suites. These languages have rich data structures and clean, well-defined semantics based on non-deterministic extended finite state machines. Lotos has been used for passive protocol monitoring of offline packet traces [6]. Since these specification languages allow non-determinism, the resulting deterministic monitors are quite inefficient and cannot be used for online monitoring.

Protocol verification systems, such as Spin, use models written in high-level specification languages, such as Promela [13], and provide sophisticated analysis tools for these models. Promela itself can be used to write protocol recognizers [2], although Spin provides no tools to monitor protocols.

Network intrusion detection systems [24] monitor a network and detect malicious user activity. Intrusion detection languages, such as Bro [26], are good candidates for network event recognition since they already have efficient implementations and can monitor packet traffic at high speeds.

Run-time verification logics and languages, such as MEDL [19], are primarily geared towards writing specifications that can be used to instrument and monitor programs running on the same machine. Still, MEDL has been used to write protocol recognizers for analyzing network simulation traces [3]. However, the lack of network-specific support limits its usefulness.

To examine to suitability of any of the above languages for NER, we consider each of our requirements:

- Which languages are designed to express protocol state machines?
  Lotos, MEDL, Promela
- Which languages have support for packet formats?
  Bro, C, Lotos, Promela
- Which languages have support for modularity and layering?
  Lotos, Promela
- Which languages support compilation to executable monitors?
  Bro, C, MEDL, Lotos, Promela
- Which languages are compatible with the C libraries for network monitoring and simulation analysis?
  Bro, C
- Which languages support automatic correctness checking?
  Promela, Lotos
- Which languages have diagnostic tools for monitoring?
  Bro
- Which languages lend themselve to automated transformations, such as those for fidelity?
  Promela, MEDL, Lotos

Unsurprisingly, no language satisfies all of our requirements. While recognizers written in Bro or C will generate efficient monitors, they are not easy to write, check, or transform. On the other hand, recognizers written in Promela, MEDL, or Lotos can be transformed and checked for correctness, but do not have efficient compilers for monitoring. Most of these limitations could be addressed by implementing new compilers, analysis tools, libraries, or programming interfaces for any of these languages. The key issue to consider is the effort involved in modifying these languages for our purpose, compared to the effort required to design a new domain specific language with its own compiler and analysis tools.

3. Network event recognition language (NERL)

Our approach is to design a high-level, domain-specific language, NERL, for programming network event recognizers. NERL recognizers are compiled to C programs that monitor protocol implementations over the network. The NERL compiler also generates code for

checking correctness, providing diagnostics, and combating infidelity in the executable monitor. NERL combines features of C, Promela, and MEDL to achieve a high-level language suited to efficient, formal network monitoring. It extends MEDL with richer data structures for events and state variables, and with a composition mechanism to put recognizer modules together, in a stack for example. NERL recognizers are translated to programs in a stylized subset of C, representing state machines with input and output events. While the NERL language itself has few novelties, its accompanying tools make NER feasible. In the next subsection, we give a short introduction to the syntax of NERL; reading this subsection is not necessary to understand the rest of the paper. Then, we describe the NERL implementation and tool suite. Finally, we discuss the efficiency of our implementation in the context of an earlier network-layer case study.

3.1. NERL syntax

As a running example, let us consider a simple protocol, called Ping, inspired by the ICMP Echo feature [29]. When a user asks node A whether node B is alive, A sends a request message to B. If B is alive, it sends back (echoes) the same message as a reply to A. When this reply reaches A, it informs the user that B is alive. The message exchange is shown in figure 2. A is called the Ping sender and B is called the Ping receiver.

The `EchoRequest` message contains two attributes: a sequence number `s`, and data `d`, and an `EchoReply` to this request must contain the same sequence number and data. The sequence number is used to match the reply to the request, while the data might contain information that is used by a higher-layer protocol—to compute round-trip time for instance.

The state machine in figure 3 represents the process at node B. It receives an `EchoRequest` and replies with an `EchoReply` that contains the same sequence number and data. In the figure, input events are suffixed by question marks while output events end in exclamation marks. Transient states—when the process has received an input event and is on the verge of producing an output—are unnamed. On the other hand, the process at

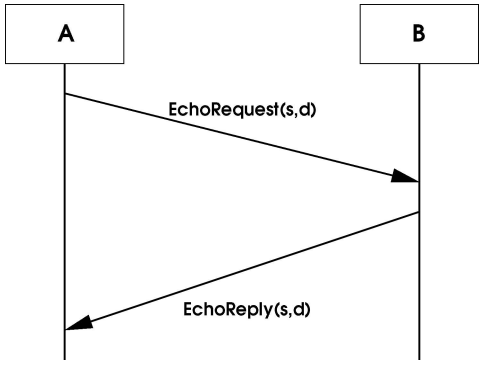

Figure 2. Ping message exchange.

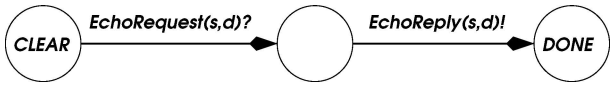

Figure 3. Ping receiver specification.

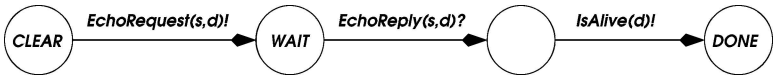

Figure 4. Ping sender specification.

node A sends a request, waits for the reply, and if they match, generates an `IsAlive` event for the higher-layer. The corresponding state machine is shown in figure 4.

We want to monitor the message exchange and carry out the following tasks

1. Check that the receiver follows the state machine in figure 3. In particular, does the receiver ever produce an `EchoReply(s,d)` without a sender having sent it an `EchoRequest(s,d)`?
2. Reconstruct the `IsAlive` higher-layer event. Whenever the Ping sender A produces an `IsAlive(d)` event for the user, the monitor should produce an `IsAlive(d)` event as well. This monitor event may be of interest to some higher-layer recognizer.

The first task corresponds to error checking, while the second corresponds to reconstructing messages for the abstract channel at the next higher layer.

We claim that the state machine shown in figure 5 performs exactly these two tasks. Notice that this state machine is essentially an extension of the Ping sender (A) state machine, except that both the lower-layer messages (`EchoRequest`,`EchoReply`) are considered inputs to the monitor, and an additional error checking transition is added. The monitor looks at a message exchange on the wire, such as the one in figure 2. It checks that the reply contains the same sequence number and data as the request, and if so produces an `IsAlive` meta-event indicating that node B successfully replied. If the reply does not match the request, a `PingError` error-event is generated. As before, transient states are unnamed.

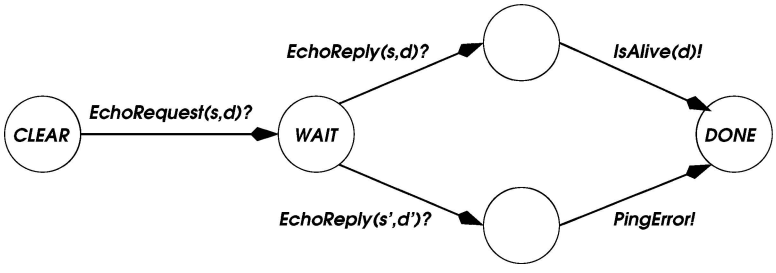

Figure 5. Ping monitor specification.

For now, the relationship between the three state machines described above is left informal. When monitoring more complex protocol state machines, it becomes necessary to demonstrate that the monitor specification (figure 5) correctly analyzes message sequences between correct protocol participants (figures 3 and 4). As we will demonstrate in Section 4, we can use model checking to formally establish this property.

In the rest of this paper, we consider the state machine as the definitive description of a protocol monitor. In the rest of this section, we illustrate the NERL program that implements the Ping monitor; the other protocol monitor state machines in this paper are implemented similarly. We refer the reader to the NERL manual in [1] for more details about the concrete syntax and other user issues for NERL.

3.1.1. Recognizer module. A NERL program consists of recognizer modules for each protocol of interest and a main module that puts recognizers together in a stack and executes them on an input packet trace.

Each recognizer module represents a monitor state machine, such as the one in figure 5, with input and output *events*. The input and output events of a recognizer constitute its *signature*. State is represented by typed *variables*, and *state transitions* are triggered by input events. New events, such as error events, are generated by *event definitions*.

A recognizer module is written as a sequence of guarded commands: each command checks whether an event has occurred and if yes, then it executes either a state transition or an event definition. There can be several *instances* of a recognizer, one for each protocol session. A recognizer *instance* operates in *rounds*; in each round, the following sequence is executed:

– Consume one input event.
– Execute every (guarded) state transition and event definition.
– Generate output events.

To see the syntax of recognizers in detail, we present the recognizer module for Ping in figure 6. The module, `Ping`, begins by defining the type, `ty_echo`, of echo packets that will be sent by A and B, using a record typedef syntax reminiscent of Promela and C (lines 2–4). For simplicity, we ignore the data fields of these messages. Lines 6–10 define a type, `ty_ping`, that represents a Ping session; each *instance* of this recognizer module monitors one session. Lines 12–18 declare the *signature* of the Ping recognizer: three input events and three output events. The input event `Init` is triggered when the recognizer intance is first initialized, that is, at the beginning of a new Ping session. Conversely, the output event `Done` indicates that the recognizer instance has finished analyzing the current session. The other events correspond to the synonymous events in figure 5. Lines 20–23 declare four variables that represent the recognizer state. The first three represent the current session parameters. The last variable, `sequence_no`, stores the sequence number seen in the last `EchoRequest` message. Following these declarations, the rest of the program consists of guarded commands representing two state transitions and two output events. When the `Init` input event occurs, the state of the recognizer instance is initialized: first the session parameters are loaded and then the sequence number is set to an initial value

```
1  Recognizer Ping =
2    typedef {
3      int sequence_no;
4    } ty_echo;
5
6    typedef {
7      int source;
8      int destination;
9      int session_id;
10   } ty_ping;
11
12   input event ty_ping Init;
13   input event ty_echo EchoRequest;
14   input event ty_echo EchoReply;
15
16   output event ty_ping IsAlive;
17   output event string PingError;
18   output event bool Done;
19
20   int source;
21   int destination;
22   int session_id;
23   int sequence_no;
24
25   transition Init(i) -> {
26                          source = i.source;
27                          destination = i.destination;
28                          session_id = i.session_id;
29                          sequence_no = -1;
30                         };
31   transition EchoRequest(req) -> {sequence_no = req.sequence_no;};
32
33   event EchoReply(rep)
34          OccurredWhen
35            (rep.sequence_no == sequence_no)
36                               -> IsAlive(a) WithAttributes
37                                  {
38                                   a.source = source;
39                                   a.destination = destination;
40                                   a.session_id = session_id;
41                                  }
42   event EchoReply(rep)
43          OccurredWhen
44            (rep.sequence_no != sequence_no)
45                               -> PingError(b) WithAttributes
46                                  {b="Incorrect Echo Reply"};
47
48   event EchoReply             -> Done(d) WithAttributes
49                                  {d = true};
50 EndRecognizer;
```

Figure 6. Ping recognizer module.

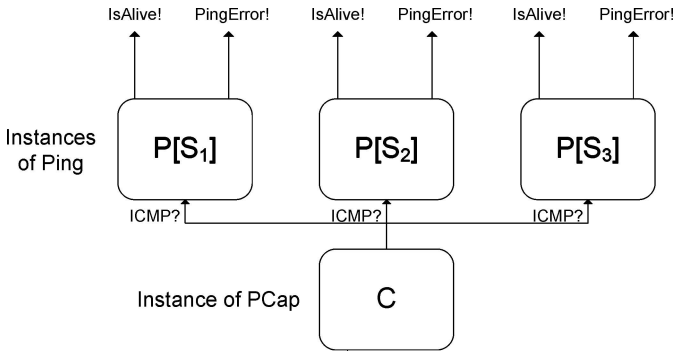

Figure 7. Ping monitoring stack.

(−1) (lines 25–30). When an `EchoRequest` event occurs, the sequence number is saved in `sequence_no` (line 31). Later, when an `EchoReply` occurs, the sequence number in it is correlated with `sequence_no`, using the `OccurredWhen` construct (lines 33–35, 42–44). If the sequence numbers match, the `IsAlive` output event is generated (lines 36–41). Otherwise the `PingError` output event is generated (lines 45–46). In either case, the `Done` event is produced indicating that the monitored session is over (lines 48–49).

Note that the Ping recognizer on its own does not tell us where the input events come from or where the output events go. This is the function of the *main* module.

3.1.2. Main module. The main module is used to set up a stack of recognizer instances, one for each protocol session of interest. For the Ping protocol, for instance, the stack is depicted in figure 7. An instance C of the *capture* module PCap 'sniffs' a stream of packets from a capture source, such as an Ethernet adapter or packet log, and, when it sees an ICMP packet, it extracts the ICMP fields from the packet and produces an output event `ICMP`. The main module maintains Ping instances `P[S_i]`, one for each observed session. When it sees the `ICMP` event, it detects whether the event contains an Echo Request or Reply, it finds out which Ping session the packet belongs to, and sends the event to the appropriate Ping instance. If the instance produces any output events, they are printed out for the user.

In this paper, we treat stacks like figure 7 as the complete description of a main module. The corresponding NERL syntax is shown in figure 8. The main modules for other protocols are written similarly. The main module begins by defining symbolic constants `REQUEST` and `REPLY` to represent the icmp type values that distinguish the `EchoRequest` and `EchoReply` messages (lines 2–3). Lines 5–9 define the type, `ty_icmp`, of ICMP packets captured from the network; the type represents the session parameters (`ip_src,ip_dst,icmp_id`), sequence number (`icmp_sequence_no`), and icmp type (`REQUEST` or `RESPONSE`) contained in these packets. Lines 11–22 declare the signatures for all the recognizer modules we use. Here, the signature of the Ping recognizer of figure 6 is repeated (lines 15–22). In addition, lines 11–14 define the signature `PCap_Sig`, for the capture module, `PCap`. The recognizer `PCap` is written in C, and uses the libpcap library to sniff ICMP packets and produce output events `ICMP`, of type `ty_icmp`. After initialization, it expects no input events. After declaring

```
1  begin
2   #define REQUEST 8
3   #define REPLY   0
4
5   typedef {
6     int ip_src;   int ip_dst;
7     int icmp_id;  int icmp_type;
8     int icmp_sequence_no;
9   } ty_icmp;
10
11  signature PCap_Sig = {
12    input event string Init;
13    output event ty_icmp ICMP;
14  }
15  signature Ping_Sig = {
16   input event ty_ping Init;
17   input event ty_echo EchoRequest;
18   input event ty_echo EchoReply;
19   output event ty_ping IsAlive;
20   output event string PingError;
21   output event bool Done;
22  }
23  recognizer PCap_Sig  PCap;
24  recognizer Ping_Sig  Ping;
25
26  instance PCap        C  WithAttributes
27                          {init = "eth0"};
28  instance Ping        P[ty_ping s]
29                          WithAttributes
30                          {init.source = s.source;
31                           init.destination = s.destination;
32                           init.session_id = s.session_id;
33                          };
34
35  event C.ICMP(e) OccurredWhen (e.icmp_type == REQUEST) ->
36        P[s].EchoRequest(p) WithIndex {
37                              s.source = e.ip_src;
38                              s.destination = e.ip_dst;
39                              s.session_id = e.icmp_id }
40                            WithAttributes
41                            {p.sequence_no = e.icmp_sequence_no;};
42
43  event C.ICMP(e) OccurredWhen (e.icmp_type == REPLY) ->
44        P[s].EchoReply(p) WithIndex {
45                              s.destination = e.ip_src;
46                              s.source = e.ip_dst;
47                              s.session_id = e.icmp_id }
48                           WithAttributes
49                           {p.sequence_no = e.icmp_sequence_no;};
50
51  event P[s].PingError(b) -> PRINT;
52  event P[s].IsAlive(a) -> PRINT;
53 end
```

Figure 8. Ping main module.

the signatures for `Ping` and `PCap`, the main module binds the recognizers to the signatures (lines 23–24). It is assumed that these recognizers have been defined elsewhere and will be found at link-time.

Next, the recognizer instances are declared and intialized. Lines 26–27 declare a single instance, `C`, of the `PCap` module and initialize to listen to the Ethernet interface for ICMP packets. Lines 28–33 declare an array of instances, `P`, of the `Ping` recognizer, one for each session `s`. Each of these instances is initialized with the appropriate session parameters: the `init` variable implicitly sets the attributes of the `Init` event. We read this instance declaration as if it generates and initializes all the possible instances `P[s]` in the beginning. During execution, however, an instance `P[s]` is generated on-demand, when the first message for a new session `s` is detected.

After initializing all instances, the rest of the module defines event forwarding definitions that send output events of one recognizer instance to the input events of another. The first two event definitions (lines 35–41, 43–49) take an `ICMP` output event with attributes `e` from the capture module `C` and check whether it is a request or a reply using the `e.icmp_type` field. Then, they compute the session, `s`, that the event belongs to, using the `WithIndex` construct, and trigger the appropriate input event, with the sequence number attribute assigned using the `WithAttributes` contruct, for the instance `P[s]` that is monitoring `s`. Note that the source and destination fields are flipped depending on the whether the ICMP packet is a request or a response. Finally, the `IsAlive` and `PingError` output events are sent to a predefined `PRINT` module that produces messages for the user (lines 51–52). We shall see examples of the messages produced by this module in the case studies of Sections 4 and 5.

3.2. Implementation and tool support

The design of NERL fulfills some of the requirements for network event recognition, namely packet format representation and layering. To satisfy the other requirements, we provide a suite of tools to help the NERL programmer. Here we outline the language and tool implementations. Their use will be demonstrated in the case studies to follow.

3.2.1. Compiler. The NERL compiler translates a NERL program to an equivalent C program. There are several advantages to using C as an intermediate language. First, the generated code can be read, modified and analyzed by C-based tools. Second, we can write and use specialized libraries and recognizers (such as `PCap`) written in C, for packet and string parsing, for example. Finally, we can rely on the optimizations and machine code generation provided by the C compiler.

Before compilation, a type-checker checks that the NERL modules are type-safe. While many of the errors caught by the type-checker are simple mistakes, such as confusing an integer for a string, it also enforces the scope of event variables: ensuring that an event's attributes are accessed only after checking that the event has occurred. This catches some subtle mistakes, and simplifies the compiler's translation of event attributes into C.

The translation from NERL to C is straight-forward. Each recognizer module is translated to a C function that executes the monitor state machine. A global data structure holds the state of each instance of a recognizer. An event is represented by a boolean flag indicating

its occurrence and a pointer to its attributes, if any. The C recognizer function takes the current state and an event as input, executes one round of the recognizer, and returns the modied state and any output events. Library modules, such as the packet capture module, are also implemented as C recognizer functions with the same type that internally invoke the appropriate library functions. The main module is translated to a main function that invokes these recognizer functions in the right order, forwarding events between them and printing out the error events.

3.2.2. Model checker for correctness. A recognizer is said to be correct if it is faithful to the protocol specification. We propose to use the Spin model-checker [12, 13] to verify the correctness of NERL recognizers.

An extension of the NERL compiler translates recognizers to Promela, the modeling language for the Spin model checker. Promela models consist of process declarations (`proctypes`), and an initial process (`init`) that generates and initializes other processes. Each `proctype` represents an extended finite state machine that sends and receives messages on channels and modifies state variables containing integers, arrays, or records. Except for some superficial differences, the syntax of Promela is quite similar to that of C. Given a model written in Promela, and a correctness property for this model written in Linear Temporal Logic (LTL), Spin then performs a state-space search to verify that the property holds for all possible executions of the model.

The translation from NERL to Promela follows the same pattern as the C translation. NERL recognizers naturally translate to `proctypes`, and the main module translates to `init`. Events are represented as inputs and outputs on channels. Protocol state is represented by global variables. For the most part, the datatypes and operators in NERL are translated to their counterparts in Promela. In two cases, the translation produces abstractions that approximate the intended data type: floating-point numbers are translated by rounding them off to the nearest integer, strings are translated to integers representing the string length.

Once we translate a NERL program to Promela, we can model-check it using Spin. For instance, for the Ping recognizer, we may wish to check the following property:

Property 3.1. *If a ping sender process correctly sends an* `EchoRequest` *message to a ping receiver that correctly responds with an* `EchoReply` *message*, *then* (1) *the ping recognizer eventually produces an* `IsAlive` *event*, *and* (2) *it does not produce any* `PingError` *error events.*

To check this property, we first write Promela models for the ping sender and receiver by encoding the state machines in figures 3 and 4. Then, we construct a composite Promela model that executes the sender, receiver, and the translated recognizer. We then encode the above safety property in LTL and verify, using Spin, that in all possible executions of this monitoring configuration, the property holds. For the Ping recognizer in this section, this check succeeds, giving us some confidence about its correctness. For protocols with more states, however, Spin may fail to complete its analysis. In such cases, we manually abstract away details from the translated Promela recognizer and check it for weaker properties.

We illustrate the use of this model-checking feature in more detail for the SMTP case study in Section 4.

3.2.3. Event tracing for diagnostics. A protocol recognizer represents a state machine with transitions labelled with input and output events. When an output event is triggered, it is caused not only by the current input, but also by the sub-sequence of preceding inputs that resulted in the current state. For instance, in Figure 5, the `PingError` event is caused by the mismatch between the current `EchoReply(s',d')` input event and the previous `EchoRequest(s,d)` input event. To understand this error, it is useful to see both these inputs. On the other hand, any other input event that did not affect this state machine is irrelevant.

We define an *event tracing* extension of the compiler that automatically computes the sequence of input events that caused an output event. Formally, this extension computes the forward dynamic slice of the recognizer program [5, 34]. Every packet event that is captured is given a unique identifier. Then, with each variable and event in the translated C recognizers, we attach its event trace: the set of packet events that affected it. This set is updated every time a variable is modified, or an event is generated. For instance, if a variable is copied to another, then the event trace is copied as well; and if an event is triggered by another, it inherits the triggering event trace. In this way, the event trace captures both the control and data dependencies between events. Finally, when an output event is sent to the Print module, its event trace is printed out as a sequence of packet identifiers.

The SMTP case study of Section 4 demonstrates the utility of this diagnostic feature in finding errors in protocol implementations.

3.2.4. Trace search for fidelity. By default, the protocols recognizers written for the monitoring configuration shown in figure 1 assume that the packet trace captured by the passive monitor has perfect fidelity: that it corresponds exactly to the input-output packet trace at the device under test. In practice, however, the presence of input buffers at the monitored devices can cause these traces to be substantially different even in a local area network [4]. In particular, packets may be delayed or dropped by the buffers. If the protocol recognizer ignores these buffers, it will be unable to separate real errors in the protocol implementation from packet trace inaccuracies, resulting in false positives.

To account for these input buffers, the protocol recognizer must guess the actual packet trace given the observed packet trace. In an earlier work [4], we proposed a *trace-search* algorithm that takes a recognizer that ignores buffers and transforms it to one that searches for a plausible buffered packet trace. We have implemented this algorithm as an extension of the NERL compiler.

Each input event in NERL can be though of as a message in some direction. If the message may be bufferred then the recognizer needs to perform trace-search to reconstruct the actual input event trace. To trigger the trace search extension, the programmer must annotate the input events of a recognizer with the parameters of the input buffer: the number of packets it can hold, and the maximum number of packets that can be lost in a burst. Then the compiler automatically generates trace search code when translating the recognizer to C.

While trace-search is effective for small protocols, the number of possible traces turns out to be exponential in the states of the recognizer. As a result, for complex protocols, trace search can be quite inefficient. Fortunately, for several classes of recognizers, we can use specialized trace-search algorithms that avoid or limit this state-space search. We have

identified 11 such recognizer classes and implemented optimized algorithms for some of them [4].

In the TCP case study of Section 5, we study this phenomenon in more detail based on the trace-search algorithm and one optimization.

3.3. Performance

One use of NERL is to analyze the correctness of trace data from network simulations. Ideally this analysis can use the same trace data as simulations intended to study performance. In earlier work on simulation trace analysis [3] we used NER to analyze conformance properties of simulations for an early version of the Ad-hoc On-Demand Distance Vector routing (AODV) [27, 28]. Our original system, Verisim 1, was based on MEDL. We used it to analyze a simulation of AODV from the CMU Monarch group. We subsequently revised the tool to produce Verisim 2, which is based on NERL. When we re-did our study of the AODV simulation using Verisim 2, we found a significant improvement in performance. We end this section on NERL with a brief analysis of these improvements.

The primary simulation we studied involves routing between 50 nodes moving at 20 meters per second in a rectangular region of 1500 $\times$ 300 meters. In this scenario there are 150 data connections transmitting four 64 byte packets every second. Each of the nodes runs AODV, and the protocol attempts to provide paths for all these connections. The simulation is run on the network simulator NS. The simulation and our subsequent analyses were carried out on a dual Pentium-III 550 Mhz Xeon processors machine with one gigabyte of memory. The OS was Red Hat Linux 7.2 with the 2.4.9-13 SMP Kernel. We used NS version 2.1b6. The simulation itself required about 5220 seconds to complete and generated 6,446,316 packet events. Unfortunately, Verisim 1 was unable to naively analyze such a large trace.[1] We estimated that the time required to check a basic property called monotonicity at each node and for each destination (2500 relations) in Verisim 1, after each of the 6,446,316 input events, would more than 100 days. By contrast, Verisim 2 processes the complete trace for all 50 nodes (2500 relations), and for a collection of AODV properties (not just monotonicity), in only 675 seconds analyzing as many as 10,000 events every second. There were essentially three reasons for this improvement. First, our NERL implementation uses C, which is more efficient that the Java bytecode implementation of MEDL. Second, NERL provides better representations for packet formats. Third, NERL has better library compatibility with other packet analysis tools. The last two features are related to the specialization of our system to NER, and deserve some elaboration.

3.3.1. Packet format representation. The AODV recognizer analyzes input events (packets) that contain at least three fields. In the NERL recognizer, the packet can be represented as a record type and passed in events. On the other hand, in MEDL, events can have only integer attributes. This means that the packet must be represented as four separate events; one indicates that the packet event has occurred, and the other three each contain an integer attribute. This unnatural encoding of event attributes makes the recognizer look more complicated than it really is. The lack of packet format representation for is even more problematic for output events. Since event attributes are unavailable, we were forced

to parameterize key properties resulting in a substantial increase in the number of events needing to be checked. The ability of NERL programs to represent packet formats directly and include them as attributes with events leads to readable and significantly more efficient programs. By avoiding the event definition expansion, equivalent NERL recognizers are several orders of magnitude faster than their MEDL counterparts.

3.3.2. Library compatibility. A second reason for the performance limitation of the MEDL analysis is that MEDL is written in Java and is meant to interact with a running Java program through a socket interface. Instead, we were using it to interact with a network simulation written in C++, through a large trace file (around 1 GB). To do this, we had to write a different front-end for MEDL that took events encoded in a text file and translated them to internal MEDL events as if they had been received over the socket interface. This front-end, written in Java, was specific to AODV; it needed to understand and parse AODV packets and map them to AODV events.

But there already exists a support library for reading and parsing NS traces. This library, written in C, can parse packet trace formats for several protocols and can be extended for newer protocols. Ideally, we would like to leverage this existing code, so that we can avoid rewriting the parsers and benefit from the higher performance of the C implementation. So we wrote a wrapper for this library that makes it available as a recognizer written in C. The NERL compiler translates the AODV recognizer to a C function, which we compile with the wrapper to produce an executable monitor. This library compatibility, along with the C *vs.* Java speedup, increases the monitor performance significantly.

4. Application layer study: SMTP

The Simple Mail Transport Protocol (SMTP) is used to transfer email between mail servers on the Internet. SMTP is an application layer protocol that runs on top of TCP. Since email is one of the dominant applications on the Internet, errors in mail server software, such as Sendmail, potentially affect millions of users. In this case study, we monitor popular SMTP servers using NERL and check whether they conform to the protocol standard [17, 30]. We first demonstrate how NERL can be used to find conformance errors in even commonly used network applications. We then consider the issue of how to analyze specialized email recognizers to determine their conformance to the standard specification. We assume that the reader is familiar with SMTP. Appendix A summarizes the background we require.

4.1. Protocol layers

Since SMTP works at the application layer, it is several steps removed from the packet events on the wire: IP *fragments* are seen on the wire, several fragments make up an IP *packet*, each packet contains a TCP *segment*, a stream of TCP segments contains SMTP *commands* and *responses*, SMTP commands are put together into an SMTP *message*, messages are parsed into *emails* according to the Internet Mail Header format. Each protocol in the network stack takes messages at a lower layer and reconstructs the messages at the higher layer. So, to reconstruct the high-level SMTP events of interest, we must mimic the network

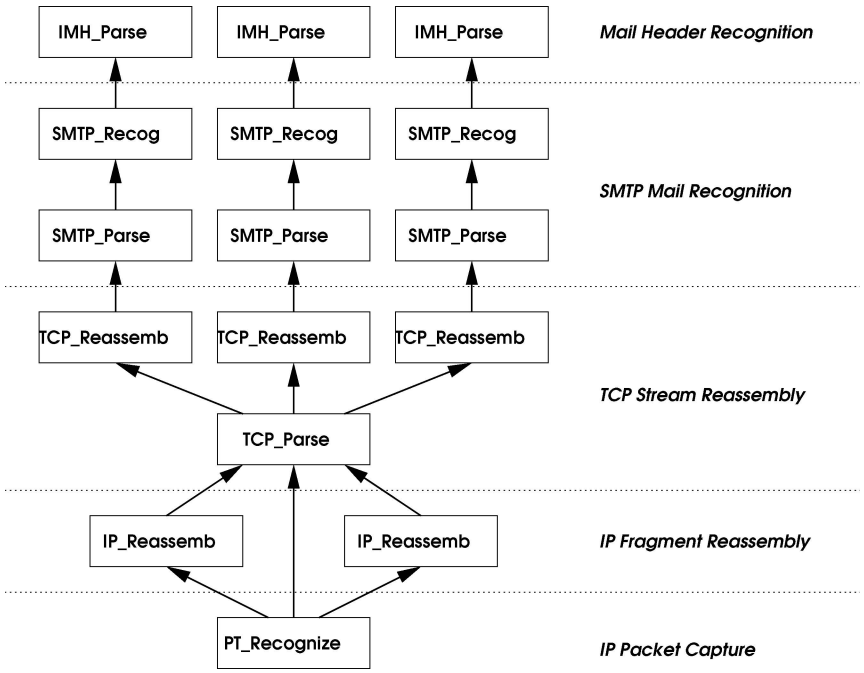

Figure 9. Recognizing emails: SMTP monitoring stack.

protocol stack. NERL allows us to write recognizers for protocols at each layer and put them together in a stack. We set up the SMTP monitoring stack as shown in figure 9. In the figure, the boxes depict instances of recognizers, while the arrows depict the input and output events. Error events are not shown. The dotted horizontal lines separate layers of the protocol stack. Some of the recognizers shown in the figure, are wrappers for libraries and perform low-level tasks such as packet capture and message parsing. These include:

– `PT_Recognize`: captures and parses IP packets and fragments
– `TCP_Parse`: takes IP packets and parses TCP segments
– `SMTP_Parse`: takes TCP text streams and parses SMTP commands and responses
– `IMH_Recog`: takes SMTP messages and parses Internet mail headers

Of these, the first two extract fields from a binary data structure; these recognizers are written in C and use libpcap for packet capture and manipulation. The last two extract tokens from a string using regular expression matching; they are written in Lex and compiled to C. By leveraging existing C-based libraries, we are thus able to conveniently write recognizers for low-level tasks, with the assurance that these will be compatible with our NERL recognizers.

Other modules in figure 9 are written in NERL and perform the logical protocol event analysis. These include

– `IP_Reassemb`: mimics the behavior of the IP reassembly layer [7, 15] at the ultimate receiver
– `TCP_Reassemb`: imitates both the sender and receiver to accurately reconstruct the data that has been successfully delivered by TCP
– `SMTP_Recog`: recognizes SMTP commands

The TCP recognizer layer simulates a channel on which the SMTP recognizer can monitor SMTP commands and responses. But note that SMTP can treat this channel as an *ideal* channel even if the IP channel on the wire was not. This is because TCP smooths out any packet loss and re-ordering; so the sequence of events as reconstructed by the TCP recognizer is guaranteed to be correct, as long as the TCP sender and receiver are working correctly. This greatly simplifies analysis at the SMTP level.

4.2. Specifying SMTP

Every SMTP server must support a minimal set of commands. The complete SMTP server state machine for this command set is shown in figure 10. The state machine is expected to be symmetric for the client, so the diagram can be read as a specification of the session. In the diagram, transitions are labeled by actions. A? indicates that command A is received at the server from from the client. B! indicates that response B is sent from server to client. There are two kinds of states in the state diagram: in *stable states* (ovals), the server is waiting for a new command, while in *transient states* (circles) the server has received a command and is about to produce a response. `OK` represents a positive server response (server accepts command), while `ERR` represents negative responses (server rejects command). `ERR` responses are fairly common in SMTP sessions; for instance if the client tries to send an email to a recipient who is not known at the server, the server sends back an `ERR` asking the client to try again with a different recipient.

To write the SMTP recognizer, we translate this state machine into NERL states and state transitions, much like the Ping example discussed before.

As an example, we illustrate the operation of the monitor for the HELO command and its responses. The monitor mimics the server state machine, and produces output events when deviations are observed. Initially, the server is in the CLEAR stable state. The HELO command can be issued in any stable state except DATAREAD, and then the state of the server is set to the transient state before HELORECD. If the HELO command has been received in a transient state or in DATAREAD, then the client has sent an incorrect command. We raise the `Command_Error` event and set the state to `JUNK`; the server should send a negative response (`ResponseErr`). When the recognizer sees a `ResponseOk` from the server, it changes state from the transient state to the next stable state. However, if the server sent `ResponseOk` when we were expecting `ResponseErr`, then the server has sent an incorrect response, so we raise the `Response_Error` output event. When the recognizer sees a `ResponseErr` from the server, it changes back to the previous stable state. An SMTP server is allowed to send an error response to any command at any time. However, if we know that the client is operating correctly, it would be unusual to see too many error responses from the server. So we also output these negative responses. If the user sees a large number

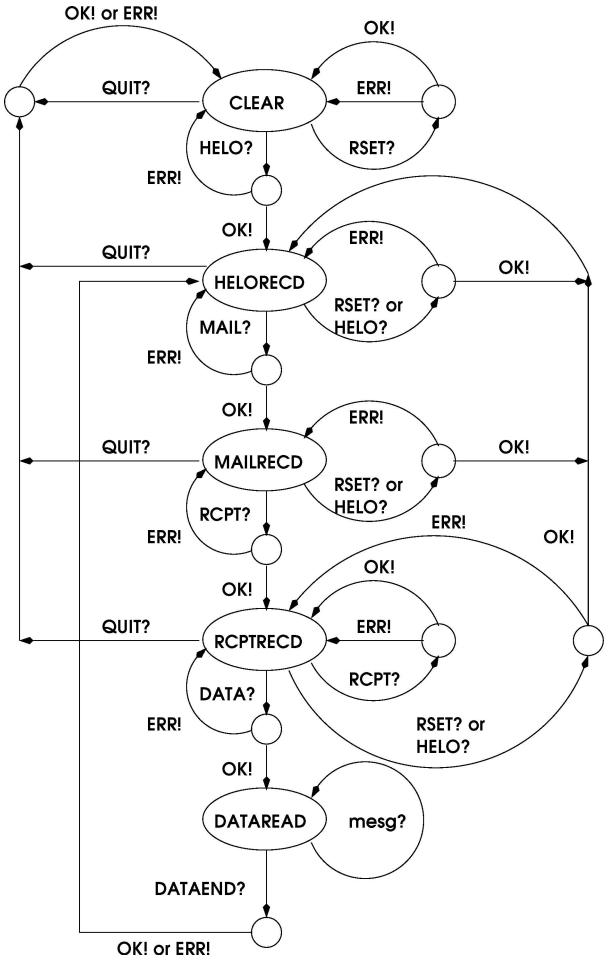

Figure 10. SMTP server state machine.

of `Response_Error` events with 'Negative response' in a correct session, he should be suspicious.

4.3. Conformance of SMTP implementations to the SMTP standard

We use our NERL SMTP recognizer for the online monitoring of SMTP sessions on a local area network. The recognizer stack observes all the traffic on the network and flags errors when it detects deviations from the standard specification. In addition, it captures all the emails sent on the LAN.

Our experimental setup consists of two Linux machines, one running a mail server and another running a mail client. We tested three popular SMTP servers: Sendmail 8.11.6,

Postfix 1.1.11 and Exim 4.10 . Each mail server also has a default mail client. But these mail clients are fully compatible with their servers, and they do not exercise the full range of commands allowed by SMTP. So we chose to design our own SMTP client. The client is written in the Expect terminal interaction language (expect.nist.gov). It connects repeatedly to an SMTP server and tries out random sequences of SMTP commands while trying to deliver a dummy email. If run for long enough, this client will try out most command-response sequences allowed by SMTP.

We run our NERL monitor stack on a third machine on the same LAN. It checks for two kinds of errors: the server accepts a bad sequence of commands, or it rejects a good sequence of commands. We found several new errors (of both kinds) in all three servers. Next, we describe one class of errors that we found through our analysis. We shall use this example to illustrate the diagnostic features built into NERL that prove quite useful in finding significant errors easily.

4.3.1. Event tracing. A typical SMTP transaction begins with a HELO command identifying the client, followed by a MAIL command identifying the sender, multiple RCPT commands identifying recipients, and a DATA command containing an email. Most transactions that we monitor between clients and servers both running Sendmail follow this rigid pattern. However, the standard itself allows many more sequences of commands. For instance, session state can be cleared at any time during the session, by issuing RSET, HELO, or QUIT. So our Expect client mixes up these commands and tries out random sequences on the server.

The rules governing the use of HELO's in a session have led to a lot of confusion. The SMTP standard says that a second HELO should be treated as a reset (RSET). But each mail server treats the second HELO differently. Sendmail 8.11.6 rejects the second HELO. On the other hand, Exim treats the second HELO incorrectly: it resets state even if the HELO was rejected. Postfix goes to the other extreme and never resets state on receiving a HELO. We illustrate the Postfix error in an execution of our client with the Postfix client; this trace has 19 SMTP sessions, 2787 packets, 7 emails transferred.

In this execution, the SMTP recognizer raises a large number of errors that indicate that the server accepted a command that should not have been accepted. For instance, consider the following error event

```
1804: SMTP Response Error:Junk Resp is OK! Code:250,state: JUNK,respstate: RESPDONE
1804: Depends on:<1690,1692,1694,1695,1697,1698,1700,1701,1709,1710,1712,1713,
      1714,1715,1717,1718,1720,1721,1722,1723,1725,1726,1728,1729,1730,1731,
      1733,1734,1736,1737,1739,1740,1742,1743,1745,1746,1748,1749,1751,1752,
      1754,1755,1767,1768,1785,1786,1795,1796,1804>
```

Here, we can see the automatic diagnostic feature built into NERL. The `Depends on` clause that follows each event indicates the control dependencies of the output event: the complete sequence of packets that has resulted in the current output event. For instance, if we wish to know what caused this `Response_Error` event to be triggered at packet 1804, we should look at 49 packets between packet 1690 and 1804. This reduces the relevant trace by a large

factor; even in the recent history since packet 1690 we only need to look at less than half the packets.

On tracing back through this packet sequence, we find the following sequence of commands and responses in the SMTP session

```
1718: MAIL FROM:<bkarthik@bangalore.cis.upenn.edu>
1720: 250 Ok
1721: RCPT TO:<bkats@bangalore.cis.upenn.edu>
1722: 250 Ok
1729: HELO bangalore.cis.upenn.edu
1730: 250 verinet.cis.upenn.edu
1731: MAIL FROM:<bkats@bangalore.cis.upenn.edu>
1733: 503 Error: nested MAIL command
1796: RCPT TO:<bkarthik@bangalore.cis.upenn.edu>
1804: 250 Ok
```

For brevity, we have not shown the intermediate commands that are not significant to the state machine.

In this sequence, we note that after the successful HELO at packet 1729, the server state should be clear, and so the MAIL command at packet 1731 should be acceptable, but is rejected as if there had already been a successful MAIL command before it and after HELO. Moreover, the next RCPT command should be rejected because there was no successful MAIL command before it and after HELO, but this command at packet 1796 is accepted, triggering the error event.

We surmise that the HELO command did not reset the server state even though it was successful. This is a violation of the standard state machine and represents another way HELO commands have been misinterpreted in the mail server community. We have confirmed the existence of this bug in the Postfix software and informed the developers.[2]

4.3.2. Results. We monitored executions of our Expect SMTP client against the Sendmail 8.11.6, Postfix 1.1.11, and Exim 4.10 mail servers. We generated around 10 error reports and analyzed them to find several errors in each mail server: 3 in Sendmail, 4 in Postfix, and 3 in Exim. Of these 6 were distinct errors, and at least 3 were previously unknown. All of the errors we found represent violations of the SMTP standard, and many of them were due to incorrect interpretations of the HELO command. We believe that the standard should be modified to include a clear specification of HELO, and of all other commands. It does not speak well of the current specification that three popular mail servers all got HELO wrong in different ways.

4.3.3. Performance. While monitoring SMTP sessions, our SMTP recognizer could always keep up with the client and the server and never missed any email or packet. To get a better idea of the processing time and memory used, we also carried out offline analysis of SMTP traces. In Table 1 we present our results for several large SMTP traces with fill tracing turned on. Traces 1, 2 and 3 are executions of Sendmail, Exim and Postfix respectively. Postfix was too slow for us to gather larger traces. Note how the analysis time is small compared to the actual execution time, indicating that our monitor can easily keep up with

Table 1. Time and memory usage for Expect trace analysis.

Trace	Pkts	Sessions	Emails	Exec(s)	Anal(s)	Mem(MB)	Output(MB)
1	25527	441	133	1487.26	5.75	5.69	19.4
2	23981	611	48	605.42	4.55	7.52	15.15
3	8825	120	45	4309.37	1.92	1.60	5.79

real sessions, in spite of executing several layers of modules. In the table the columns list from left to right: the execution trace number, number of packets in the execution, number of SMTP sessions started between client and server, actual number of emails exchanged (many sessions end without successfully sending an email because of the random nature of the client), the lifetime of the execution calculated as the difference in time between the start and end of the trace, the analysis time taken by the SMTP recognizer, the peak memory usage of the recognizer over the trace, and finally the size of the output error trace produced by NERL. This output trace file is quite large, because it contains all error events at all four layers (IP,TCP, SMTP,IMH), meta-events produced by each layer, and full tracing to help diagnose implementation errors. With different options turned off, such as tracing or lower-layer events, this file would be much smaller.

We find that the SMTP recognizer stack can easily keep up with up to 150 concurrent SMTP sessions. The speed of analysis varied between 4400 and 8400 packets per second. The peak memory requirement was 12.8 KB per session. Even with full alarm tracing we find that the NERL recognizers are fast enough to monitor live executions of SMTP.

4.4. Correctness of the SMTP recognizer

Writing a recognizer for a protocol involves several steps. First, we need to interpret the standard correctly and extract a complete state machine. Then, we may choose to monitor only the part of the state machine that interests us. Finally, we encode the relevant state machine in a programming language. Each of these steps may have errors. In this section, we show how model checking can be used to check that the resulting recognizer is correct.

Let us formalize some correctness properties for our SMTP recognizer. An SMTP recognizer that checks for server errors should be able to faithfully mimic the behavior of a correct SMTP server. So given any sequence of commands, it should anticipate the server responses exactly.

Property 4.1. *If a correct SMTP server process is sent an arbitrary sequence of SMTP commands, then the SMTP recognizer produces* `Response_Error` *events if, and only if, the server sends a negative response.*

In particular, the recognizer should not generate any false positives.

Additionally, an SMTP recognizer that reconstructs emails should capture an email only when a correct server would accept it.

Property 4.2. *If a correct SMTP client sends an email M to a correct SMTP server, then the SMTP recognizer produces an* `EmailAccepted` *event containing M if, and only if, the server successfully accepts the email M.*

We claim that the SMTP recognizer that we have programmed in NERL obeys these properties.

To demonstrate the value of model-checking for monitors we performed an analysis of two other systems, the *Bro* network intrusion detection system and the *Altivore* surveillance system. We encoded SMTP recognizers for each of these systems in Promela and checked the Properties 4.1 and 4.2.

To check Property 4.1, we encode an SMTP server process in Promela, and a non-deterministic client process that sends an arbitrary sequence of commands to the server. The SMTP monitor is a third Promela process that has access to the channel between the client and server. We automatically generate the SMTP monitor for our recognizer, `SMTP_Recog`, by using the translation from NERL to Promela. We express Property 4.1 in Linear Temporal Logic (LTL) and use the model-checker Spin to check that the monitor satisfies the properties. Spin model-checked this recognizer and found no errors; it analyzed 2330 states and 18,689 transitions to reach this conclusion. Then we encode Bro's SMTP monitor in Promela and check Property 4.1. Spin finds a short counter-example in the very first state: when an RSET is issued in the CLEAR state, Bro sets the new state to HELORECD, instead of CLEAR. As a result, Bro produces *false negatives*: it accepts sequences of commands that the server would reject. But apart from this minor error, the Bro SMTP monitor is correct.

To check Property 4.2, we write a complete SMTP client process in Promela and reuse the server process. As before, we express the property in LTL and run Spin to check for errors. Again, Spin found no errors in our NERL recognizer (translated to Promela). Then we check the Altivore version by encoding it in Promela. Spin finds an error and produces a counter-example: Altivore does not correctly handle the case when the client tries to send *two* emails and the first is rejected; Altivore fails to capture the second email. To find this error, Spin analyzed 871 states and 3135 transitions to produce a counter-example with 12 message exchanges. This error would have been difficult to find manually.

To summarize, we checked three SMTP monitors, and found that two of them had subtle flaws that caused them to fail in their stated objectives. For this case study, we took the trouble to read the C and C++ code for the monitors to extract the state machine and encode it in Promela. But in general, it is important to have tools that can automatically generate model-checkable code. Our NERL recognizer can be automatically checked, and so it was feasible to check and maintain it as it evolved through several versions, resulting in a correct recognizer program.

5. Transport layer study: TCP

TCP is the primary transport protocol on the Internet; it is considered an essential requirement for 'host system implementations of the Internet protocol suite' [7]. It imposes a reliable, in-order channel on top of IP. Many implementation-layer protocols, such as

SMTP, FTP, Telnet, need this reliability and operate on top of TCP. In fact, security requirements such as confidentiality are implemented as layers above TCP (TLS, SSH). Clearly, it is important to test implementations of TCP and of other transport protocols because they underlie many critical Internet services.

Our aim in this section is to carry out a study of issues in NER for TCP. In particular, we confront the challenge of dealing with infidelity introduced because of buffered and lossy monitoring channels. We address these problems through the use of *trace search* illustrated through a study of the conformance to standard of the acknowledgement rate for several TCP implementations. This is an application of the transformations studied in our earlier work [4].

5.1. Specifying TCP reliability

TCP describes a sequence of messages between two participants, a sender and a receiver, that allows the sender to reliably transmit an array of bytes to the receiver. TCP messages are sent inside IP packets. For the purposes of this study, we shall ignore IP fragmentation and assume that each TCP message is contained in exactly one packet. A TCP session starts with a handshake between the two parties to establish session parameters. Then the sender transmits a series of DATA packets containing segments of a byte array. The receiver periodically transmits ACK packets acknowledging the last contiguous byte it has received in the array; note that if some DATA packets get lost there may be 'holes' in the byte array received at the receiver. When the entire byte array has been received and acknowledged, the sender and receiver shut down the connection with another handshake. Appendix B summarizes the background on TCP needed to follow our analysis here.

This sequence of messages and several exceptional cases are described in the TCP standard in the form of a session state machine. We write a NERL recognizer to check deviations from this state machine. Encoding the state machine is straight-forward; states are represented by a `status` variable, and transitions in the TCP state machine are mapped to NERL state transitions that modify `status`; any deviations from the state machine is reported as an output (error) event, `TCP_Error`.

For instance, one simple property is the *responsiveness* of TCP receivers:

Property 5.1. *The TCP implementation must generate an ACK for at least every other DATA segment it receives.*

To check this, we count the number of outstanding DATA packets and ensure that the count never exceeds 2. Whenever a DATA packet is seen, the number of unacked packets is incremented. When an ACK is seen, this value is reset to zero. If more than two DATA packets are seen without an intervening ACK, an error is raised. In the remainder of this section, we shall concentrate on this property.

5.1.1. Ideal channel assumption. We first use the TCP recognizer to analyze a typical TCP session between two hosts on the same LAN. This session contains 270 packets between two Linux TCP implementations. Since we captured the packet trace for this session when

the LAN was lightly loaded, we assume that the monitored channel is ideal: no packets are lost or delayed.

When we run the recognizer on this session, the recognizer produces a large number of error events, indicating errors in the TCP implementation. For instance, the following sequence of packets leads to an error event at packet 62:

– Packet 60 is a DATA packet from sender to receiver
– Packet 61 is a DATA packet from sender to receiver
– Packet 62 is a DATA packet from sender to receiver
– TCP Recognizer raises error: `Too many unacked packets`
– Packet 63 is an ACK packet for packet 61 from receiver to sender
– Packet 64 is an ACK packet for packet 62 from receiver to sender

Seemingly, at packet 62 the sender has sent 3 DATA packets but not received any ACK packets. So the TCP recognizer for the responsiveness property raises the error.

However, looking further down the trace, we find that packet 63 is an ACK for DATA packet 61. This means that at the time the recognizer raised the error, the receiver had not yet seen the DATA packet 62. So the actual sequence of events needs to take into account the delay between the time packet 62 was seen on the wire and the time that it was actually processed at the receiver:

– Packet 60 (DATA) is sent and received
– Packet 61 (DATA) is sent and received
– Packet 62 (DATA) is sent
– (No error event)
– Packet 63 (ACK) for DATA packet 61 transmitted by receiver
– Packet 62 (DATA) is received
– Packet 64 (ACK) for DATA packet 62 transmitted by receiver

The delay in packet 62 is caused by an input packet buffer at the receiver. Most operating systems have input buffers that get populated when the machine cannot keep up with the input traffic. So in this example, the receiver TCP implementation was not incorrect, it was just slow to process the packets. The error event raised at packet 62 was a *false positive*. In fact, all the errors produced in this trace were false postives. In longer traces, the resulting spurious error events make finding real errors impossible. For instance, in a TCP trace with 25527 packets there were as many as 788 false positives. They only serve to distract the monitor from finding real implementation errors.

In our analysis of TCP traces generated from Linux, Windows, and Solaris machines, our NERL recognizer raises the same error, 'Too many unacked packets', several times for all three implementations. Some of these may be false positives, while others may indicate real errors. In the next subsections, we shall attempt to distinguish between these cases.

5.2. Addressing Infidelities through trace search

The behaviour of the non-deterministic packet buffer at the receiver is the following:

– When a DATA packet arrives, cache it.
– At any time, pick the oldest DATA packet and deliver it.
– At any time, pick the oldest DATA packet and drop it.

Our TCP recognizer, which only models the sender and receiver, must now also model this intermediate buffer process. The new buffer-aware TCP recognizer would behave as follows:

– When a sequence of DATA and ACK packets is seen, search for all possible executions of the buffer, sender, and receiver that can account for this sequence.
– If no execution can account for the sequence, raise an error.

We call this the *trace-search algorithm* [4]. Ideally, given the parameters of the buffer, such as the maximum number of packets it can hold and the maximum number of packets it can lose in one burst, we should be able to automatically transform the original TCP recognizer to this buffer-aware version. The NERL compiler comes with a tool that can perform this automatic channel transformation.

To use the trace search feature, the user only needs to modify the signature of the TCP recognizer to provide it with the channel parameters. In our study we define one input channel for DATA packets and an output channel for ACK packets. We assume input buffer sizes of 5, with no loss. The NERL compiler uses these channel annotations to compile the trace-search algorithm into the generated C code. Now when we run the modified TCP recognizer on the same packet traces as before, we notice that a large number of the error events disappear. For instance, the earlier error at packet 62 of the 270 packet trace now looks like this:

```
62: ----------------------------
62: Possible TCP Error
62: TCP Error:Too many unacked packets:
62: ----------------------------
```

This error is *possible* which means that one of the plausible sequences generated by the trace-search algorithm is rejected because of this error. But this error never translates to a ‘Definite TCP Error’ because there are other plausible sequences which do not have this error. No *definite* TCP errors are raised in this trace; the recognizer successfully classifies all the possible error events as false positives.

It is important to note that while finding an error would have been significant, just knowing that the few thousand error events generated earlier are probably false alarms arising from buffer infidelities is a big time saver. The trace-search algorithm effectively sifts through these ‘Possible TCP Error’ events and throws away everything that can be explained by

buffering and loss. Normally, this would be done manually, leading to a substantial waste of time.

5.3. Conformance

We generated several traces of TCP sessions between Linux 2.4, Solaris 5.8, and Windows XP machines and analyzed them using our buffer-aware TCP monitor. To exercise the full behavior of the TCP implementations, we introduced loss at the sender by randomly dropping one out of 10 packets to simulate high load.

While Linux TCP traces did not trigger any error events, both Solaris and Windows TCP traces failed this property. We find that the Windows and Solaris implementations are not very prompt in issuing ACKs. To find out how many packets they wait for before producing an ACK, we relax the requirement that every two packets must be ACKed. We find that both Windows and Solaris produce ACKs for up to 5 packets at a time in these traces. Both these TCP implementations seem to generate ACKs based on a timer rather than counting the number of DATA segments received. We have observed them sometimes producing ACKs for as many as 10 packets at a time. Since the source code for these OSes is not available, we could not check their procedure for generating ACKs.

The Solaris' ACK delay was also detected by Paxson's analysis of TCP implementations [25]; the notion of delaying ACKs is a known optimization technique in the networking community. However, consistently delaying ACKs can have many drawbacks. Paxson argues that delaying ACKs is network-unfriendly and, in fact, provably sub-optimal when packets are being lost by the network.

So, while the error that we found does not count as a major error in the TCP implementations, the study does demonstrate the usefulness of packet buffer modeling. For instance, let us look at fragments of 2 real TCP traces that we captured for our study:

Linux TCP trace	Windows XP TCP trace
`57: DATA from 158.130.013.033`	`11: DATA from 158.130.013.033`
`to 158.130.012.217`	`to 158.130.012.110`
`58: DATA from 158.130.013.033`	`12: DATA from 158.130.013.033`
`to 158.130.012.217`	`to 158.130.012.110`
`59: DATA from 158.130.013.033`	`13: DATA from 158.130.013.033`
`to 158.130.012.217`	`to 158.130.012.110`
`60: DATA from 158.130.013.033`	`14: DATA from 158.130.013.033`
`to 158.130.012.217`	`to 158.130.012.110`
`61: ACK (for Pkt 58) from 158.130.012.217`	`15: ACK (for Pkt 14) from 158.130.012.110`
`to 158.130.013.033`	`to 158.130.013.033`

The trace on the left was for a Linux TCP receiver (158.130.012.217), and the trace on the right was for a Windows TCP receiver (158.130.012.110). In both trace fragments, the receiver seems unresponsive: we see 4 DATA packets in a row before an ACK. But the

Table 2. Time and memory usage for TCP trace analysis

Trace	Pkts	Buf	Unacked	Exec(s)	Anal(s)	Mem(MB)
Linux 1	4930	5	2	47.46	11.6	22.23
Linux 2	9804	5	2	95.36	23.21	21.89
WinXP 1	4752	5	5	55.2	12.38	14.62
WinXP 2	9474	5	10	113.96	25.23	13.16
Solaris 1	4930	5	5	47.46	13.92	18.11
Solaris 2	9814	5	5	95.36	27.75	21.64

Linux implementation is in fact correct, and the trace fragment on the left can be explained as 2 packets getting buffered and 2 being consumed before the receiver sends an ACK. On the other hand, the Windows XP implementation is incorrect because as we go further in the trace we find that there is no amount of buffering that can account for the small number of ACKs produced. Clearly, only by taking buffers into account correctly can we hope to distinguish between these traces.

5.4. Performance

While the buffer-aware TCP recognizer was able to find errors and ignore false positives, it takes a long time to process a packet trace. In Table 2, each row lists, from left to right: the trace number (and the corresponding OS), the number of packets in the trace, the buffer size used for input channels, the maximum number of unacked packets allowed, the total time (in seconds) over which the TCP session was played out, the time taken for the NERL analysis, and the peak memory usage of the monitor. The buffer-aware recognizer can use up to 23 MB of memory to analyze a 5000 packet trace, and consumes packets at around 500 packets per second. This is significantly slower than our AODV study, where we analyzed 10,000 packets per second and used a minuscule amount of memory. On the other hand, the traces we have generated are a little pathological with a 10 per cent loss rate. We find that the monitor does significantly better in the absence of loss, since loss exponentially increases the state space of the buffer. But, for lossy traces, it cannot handle more than 4 TCP sessions at a time.

5.5. Optimized trace search

The trace-search algorithm we have used to monitor TCP traces effectively carries out a state-space search of the composition of a non-deterministic buffer with a deterministic non-finite-state recognizer. For a general protocol, there is no bound on the amount of work needed at each packet. We may need to maintain a set of plausible recognizer instances whose size is exponential in the number of packets in the input trace.

Fortunately, there are several property-based optimizations that we can carry out for specific kinds of protocols. In an earlier paper on network monitoring [4], we proposed

several optimizations based on the property that we were interested in monitoring. It turns out that for several classes of properties, buffering and/or loss on input channels can be ignored. For some other classes, we can significantly reduce the number of plausible sequences generated by the algorithm.

Here, we describe one optimization for the responsiveness property (Property 5.1). It is an instance of a property class that we call *Counting properties*. Such properties do not care about the content of packets, they only count the number of inputs and outputs.

Property 5.2. *[Counting Property] Every output produced by the device under test must consume at least* $c_{\min}$ *and at most* $c_{\max}$ *inputs.*

For TCP receivers (DATA as input, ACK as output), $c_{\min} = 0$ and $c_{\max} = 2$.

Suppose that input channels have a buffer size of B, and output channels are unbuffered, and assume that no packets are lost in the input buffer. Then, we can design a special recognizer for checking the counting property in the presence of buffering. The recognizer implements the algorithm shown in Table 3. Here inputs (DATA) are indicated by i and outputs (ACKs) are indicated by o. This algorithm maintains two integers, $buf_{\min}$ and $buf_{\max}$, representing the minimum and maximum number of inputs that are currently buffered on the channel between the monitor and the device under test. If $buf_{\min}$ ever grows too large, it indicates that the particular i/o 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 i/o string seen so far could not reflect a valid execution because even if each output has consumed the maximum number of inputs, there have not been sufficient inputs to account for every output. In each case an error flag e is set.

Note that a recognizer that implements this algorithm (raising an error event whenever e is true), already takes buffering into account and therefore does not need the trace-search algorithm. In effect, the algorithm in Table 3 is a specialized trace search for one class of

Table 3. Algorithm for checking Counting Property.

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
– i:
$buf_{\max} = buf_{\max} + 1$;
$buf_{\min} = buf_{\min} + 1$;
if ($buf_{\min} > B + c_{\max}$) then $e =$ true
else $buf_{\max} = \min(buf_{\max}, B + c_{\max})$
– o:
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.

Table 4. Time and memory usage for TCP Counting property.

Trace	Pkts	Buf	$\mathbf{c}_{\max}$	Exec(s)	Anal(s)	Mem(MB)	Result
Linux 1	4930	5	2	47.46	0.19	0.071	Pass
Linux 2	9804	5	2	95.36	0.51	0.071	Pass
WinXP 1	4752	5	2	55.2	0.22	0.071	Fail
WinXP 2	9474	5	2	113.96	0.37	0.071	Fail
Solaris 1	4930	5	2	47.46	0.2	0.071	Pass
Solaris 2	9814	5	2	95.36	0.41	0.071	Fail

recognizers. This notion is formalized as a proof of correctness for a reduction from the trace-search algorithm to the counting algorithm [4].

When we use this algorithm to check the TCP traces generated before, we find that the performance has improved significantly. Table 4 contains the time and memory usage for checking the TCP counting property for the same traces as before. Both the analysis time and memory usage are much lower than for trace-search. There is one quirk in the results: the Solaris 1 trace passed the counting test even though it generated this error in the general TCP recognizer. This is a *false negative*. This shows that our counting property is a little too liberal: it sometimes passes incorrect traces. The reason for this is that without keeping track of the receiver state, the counting property has no way of knowing whether a single ACK is acknowledging one or two TCP packets, so it conservatively assumes that it may be acknowledging 2 packets. This means that some errors that would be caught by a more complete recognizer are missed by our specialized algorithm.

The counting algorithm presented in this section shows how buffer-aware recognizers for special properties may be programmed. We have investigated such programming strategies for a variety of other property classes [4]. These could be automated for NERL using a similar strategy to the one used here for the counting algorithm.

5.6. Results

We find that the Windows XP implementation of TCP violates the standard and does not promptly produce acknowledgments for received data packets. The standard says that every ACK should acknowledge at most 2 DATA packets. Windows XP sometimes generates ACKs for as many as 10 DATA packets at a time. We found that the Solaris TCP implementation also violates the standard: it sometimes generates an ACK for as many as 5 DATA packets. On the other hand, we found no conformance errors in the Linux implementation.

Transport layer protocols such as TCP cannot make ideal channel assumptions. In fact, often their purpose is to create a reliable channel on top of one that allows loss and delay. As a result, a monitor for a transport layer protocol must also take channel buffering and loss into account. Our approach is to allow the programmer to write a recognizer for the ideal monitoring channel, and then automatically transform this recognizer into one that works on the real channel.

We demonstrated how to use NERL's trace search feature to find errors in TCP implementations. Even on correct TCP executions, we found that the transformed monitor could filter out all the false positives generated by the naive TCP recognizer. It could consistently distinguish between errors in the implementation and abnormalities caused by channel buffering and loss.

6. Conclusion

We have described the concept of network event recognition and a collection of requirements for convenient and efficient runtime analysis using NER. We have designed a language NERL and implemented a prototype interpreter and some tools for using NERL to provide automated network event recognition. This system has been used to analyze protocols at network, transport, and application layers. These studies show the importance of the requirements and the existence of feasible approaches for addressing them in significant and non-trivial applications.

The idea of a developing a language for network event recognition came from a meeting between the authors of this paper and Satish Chandra and Pete McCann, who also provided us with considerable help and encouragement in various stages of the project. Our efforts benefited from our collaboration with Moonjoo Kim, Insup Lee, Davor Obradovic, Oleg Sokolsky, and Mahesh Viswanathan on the Verisim 1 system. We appreciated contributions from David Farber and Jonathan Smith, including their ideas on extending Overseer to active networks and Gerard Holzmann for his advice on implementing NERL. This research was supported by DARPA F30602-98-2-0198, ARO DAAG-98-1-0466, ONR N00014-99-1-0403 and ONR N00014-00-1-0641.

Appendix A: Background on SMTP

SMTP uses a TCP session between two Mail Transfer Agents (MTAs) to deliver multiple emails between them. After the TCP session is established, the SMTP dialogue begins when the sender MTA (the *client*) sends the `HELO` *command* to the recipient MTA (the *server*), which then sends either an Ok or an Error *response*. The client can then send the next command, and wait for the next response and so on.

A typical SMTP session that delivers an email is given in Figure 11 where `C:` indicates client commands, and `S:` indicates server responses. Here, the MTA at domainA is talking to the MTA at domainB, and in the first 3 lines the two MTAs identify their domains. The client then initiates an email delivery by naming the sender (S) in a `MAIL FROM` command, and then naming the recipient (R) in an `RCPT TO` command. The client can name multiple recipients for an email, but only one sender. The server can reject the sender or a recipient by sending an error message (line 9). After sending the envelope information, data delivery begins after the `DATA` command in Line 10 is accepted by the server. Lines 12 to 21 contain the message sent by the client. Data delivery ends at Line 22 with a special line that just has a full stop in it. The client can then send another email or close the session with a `QUIT`.

```
 1. S: 220 domainB Simple Mail Transfer Service Ready
 2. C: HELO domainA
 3. S: 250 domainB greets domainA
 4. C: MAIL FROM:<S@domainA>
 5. S: 250 OK
 6. C: RCPT TO:<R@domainB>
 7. S: 250 OK
 8. C: RCPT TO:<Rsec@domainB>
 9. S: 550 No such user here
10. C: DATA
11. S: 354 Start mail input; end with <CRLF>.<CRLF>
12. C: To: "Rob R Roy" <R@domainB>
13. C: From: Sam S Smith <S@domainA>
14. C: Reply-To: "Smith:Personal" <S@personal.domainA>
15. C: Cc: "Roy's Secretary" <Rsec@domainB>,
16. C:     "Smith's Secretary" <Ssec@domainA>
17. C: Subject: Saying Hello
18. C: Date: Fri, 21 Nov 1997 11:00:00 -0600
19. C: Message-ID: <abcd.1234@local.machine.tld>
20. C:
21. C: This is a message just to say hello.
22. C: .
23. S: 250 OK
24. C: QUIT
25. S: 221 domainB Service closing transmission channel
```

Figure 11. Sample SMTP session.

An SMTP client can also issue a `RSET` command at any time during a transaction to reinitialize the session. Similarly, `HELO` and `MAIL` commands can also be issued at any time to reinitialize the session.

In addition to these commands, SMTP allows the `VRFY`, `EXPN`, and `HELP` commands at any time, to extract information from a server. Newer versions of SMTP also allow service extensions to the standard protocol. We do not consider these commands and extensions here since they do not affect standard message delivery.

SMTP transfers emails between mail servers. When an email is delivered to the MTA at the recipient domain, it is stored in a mailbox owned by the recipient on the mail server. The recipient can then log in to the mail server to check his mailbox. Many users, however, like to read their email on desktop computers that are not powerful enough to act as mail servers. Protocols such as the Post Office Protocol [23], and the Internet Message Access Protocol [8] enable users to access their mailboxes remotely, with facilities to download message headers and bodies, and delete them from the mail server. These protocols work well for incoming email. In order to send email, however, the desktop computer must still use SMTP as a client to hand over messages to the MTA at a mail server.

Internet mail format. We have described the Internet mail architecture and the SMTP protocol that delivers email between the MTAs at two mail servers. Internet users, however, never need to be aware of SMTP or even the MTA at their own server. This is because most mail users use Mail User Agents (MUAs), such as Lotus or Outlook, that help them to compose, send, and receive email messages.

MUAs give the user a lot more flexibility in describing the attributes of a message. For instance, the sender S can define who the email is `From` as well as who the recipient should `Reply-to`. S can choose who to address the email `To`, and who should get a copy (`Cc`,`Bcc`). S can even specify the `Subject` of the email. All these attributes are included at the beginning of an email according to a standardized Internet Message Format [9, 32]. Although the complete format is quite involved, and has a number of options, a typical Internet message is as shown in Lines 12 to 21 in Figure 11.

The attributes at the beginning of the message comprise the message *header*, separated from the *body* by an empty line. The MUA is responsible for taking such a message and automatically generating the email envelope by looking at the addresses in the `To`, `Cc`, and `From` fields.[3] It then hands over the complete message and envelope to an MTA to carry out the actual delivery. When the MTA at the recipient's mail server receives the message, the recipient R looks at the email through his own MUA, which parses the mail headers and cleanly presents them.

It is important to note, though, that a mail user need not go through an MUA in order to generate an Internet message. S can type the message in a text editor, and directly interact with the MTA to deliver the typed message, according to a specified envelope. So there is no guarantee that the mail headers have any relation to the actual senders or recipients of a message. Moreover, since a user can specify an envelope to an MTA, the sender and even the recipient in the envelope may not exist. Some MTAs will refuse to accept messages if they can determine that the senders or recipients are unknown, but many MTAs do not have enough information to make this decision and will accept the messages anyway.

In addition to the message formats described in this section, further structure can be imposed on the message body, for instance to describe and include attachments (using MIME), and to authenticate or encrypt the message (using S/MIME).

Appendix B: Background on TCP

TCP has been described in great detail and with increasing clarity in several RFC's [7, 16] and books [33]. TCP provides a stream interface to application layer protocols, much like file input-output. When a higher layer protocol such as Telnet needs to send data to a host across the network, it issues an OPEN command with a local port, remote host and remote port number. TCP then sets up a socket that connects `<local_host, local_port>` with `<remote_host,remote_port>`. This initial setup is achieved by a handshake consisting of a SYN packet to, a SYNACK packet back and an ACK packet to the remote host.

Once the socket is open, the application layer protocol simply writes data onto it using the SEND command, while the remote host reads data using the RECEIVE command. These are treated like file write and read at each host, while TCP carries out the actual

data transfer using DATA packets. TCP receivers acknowledge receipt of DATA, FIN and SYNACK packets by sending ACK packets.

TCP allows data transfer in both directions (a duplex channel) irrespective of which host actually initiated the connection. This is important for synchronization in higher-layer protocols. For instance, in SMTP, even though emails are only transferred from client to server, the client needs to wait for a server OK after almost each line it sends.

Finally, when the transmission of data is completed, the sending application-layer protocol issues a CLOSE command to the TCP socket, and the sending TCP sends a FIN packet to the the remote host. Once this FIN and all preceding data has been acknowledged, the remote host sends its own FIN and the connection is closed.

TCP reliability. 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. Suppose the receiver advertised a window size W in the initial handshake, the sender has sent data segments up to sequence number $S_{\max}$, the receiver has received contiguous data up to sequence number S_{Cont}, and the last acknowledgment it sent had sequence number A_L, then the next acknowledgment A_N produced by the receiver must follow

$$A_N = S_{\text{Cont}} + 1$$

Since S_{Cont} monotonically increases, this also means that

$$A_N \geq A_L$$

When the sender receives this ACK, since it does not know S_{Cont} and only knows $S_{\max}$, it accepts the ACK if it follows

$$A_L \leq A_N \leq S_{\max} + 1$$

When the sender sends the next segment with sequence number S_N, and length L, this segment must obey the limitations of the receivers' window, based on the last ACK A_L it

received.

$$A_L \leq S_N < A_L + W$$
$$A_L \leq S_N + L - 1 < A_L + W$$

The receiver accepts this segment if it satisfies its window.

$$S_{\text{Cont}} + 1 \leq S_N \leq S_{\text{Cont}} + W$$
$$S_{\text{Cont}} + 1 \leq S_N + L - 1 \leq S_{\text{Cont}} + W$$

Note that the rules adopted by the sender and the receiver are symmetric and equivalent as long as frequent acknowledgments synchronize their views of the receiver window. TCP implementations are required to send acknowledgments for every two data segments, so an implementation may either send an ACK for every DATA, or may decide to ACK only every other segment.

Notes

1. We were able to get significant results with smaller traces and, by applying manual optimizations that improved performance, we were able to get good information from this large trace. See [3] for details.
2. It turns out that Postfix treats the HELO command differently from EHLO (the extended HELO defined in [17]) and for EHLO it does reset the state correctly. However, the processing for EHLO commands contains a separate error: it always resets the state even if the EHLO failed.
3. Resnick [32] describes a number of other fields that may contain addressing information as well.

References

1. K. Bhargavan, "Network Event Recognition," PhD thesis, University of Pennsylvania, 2003.
2. K. Bhargavan, C.A. Gunter, and D. Obradovic, "Fault origin adjudication," *Formal Methods in Software Practice*, 2000.
3. K. Bhargavan, C.A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan, "Verisim: Formal analysis of network simulations," *IEEE Transactions on Software Engineering*, Vol. 28, No. 2, pp. 129–145, 2002.
4. K. Bhargavan, S. Chandra, P.J. McCann, and C. A Gunter, "What packets may come: Automata for network monitoring," in *Proceedings of the Symposium on Principles of Programming Languages (POPL'01)*, ACM Press, pp. 206–219, 2001.
5. D. Binkley and K. Brian Gallagher, "Program slicing," *Advances in Computers*, 1996.
6. G.V. Bochmann and O. Bellal, "Test result analysis with respect to formal specifications," in *Proc. 2-nd Int. Workshop on Protocol Test Systems.* Berlin, 1989 pp. 272–294,.
7. R. Braden, "Requirements for internet hosts—communication layers," Technical Report RFC 1122, IETF, 1989.
8. M. Crispin, "Internet message access protocol—Version 4rev1," Technical Report RFC 2060, IETF, 1996.
9. D. Crocker, "Standard for the format of ARPA internet text messages," Technical Report RFC 822, IETF, 1982.
10. S.A. Ezust and G.V. Bochmann, "An automatic trace analysis tool generator for estelle specifications," *Computer Communication Review*, Vol. 25, No. 4, pp. 175–184, 1995 Proceedings of ACM SIGCOMM 95 Conference.

11. D.J. Farber and J.B. Picken, The overseer, a powerful communications attribute for debugging and security in thin-wire connected control structures, in *Proceedings of International Computer Communications Conference*, 1976.
12. G.J. Holzmann, "SPIN-formal verification," Web Page. Available at http://netlib.bell-labs.com/netlib/spin/whatispin.html.
13. G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991. http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html.
14. G.J. Holzmann, *Logic Verification of ANSI-C Code with SPIN*, Springer Verlag/LNCS, 1885, pp. 131–147, 2000.
15. Information Sciences Institute, "Internet protocol," Technical Report RFC 791, IETF, 1981a.
16. Information Sciences Institute, "Transmission control protocol," Technical Report RFC 793, IETF, 1981b.
17. J. Klensin, "Simple mail transfer protocol," Technical Report RFC 2821, IETF, 2001.
18. E. Kohler, M. Frans Kaashoek, and D.R. Montgomery, "A readable TCP in the Prolac protocol language," in *Proceedings of the ACM SIGCOMM'99 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, Cambridge, Massachusetts, pp. 3–13, 1999.
19. 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.
20. Lotos: A formal description technique, 1987.
21. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.
22. P. McCann and S. Chandra, "PacketTypes: Abstract specification of network protocol messages," in *ACM Conference of Special Interest Group on Data Communications (SIGCOMM)*, August 2000.
23. J. Myers and M. Rose, "Post Office Protocol—Version 3." Technical Report RFC 1939, IETF, 1996.
24. NSS Group, "Intrusion detection systems—group test," December 2001.
25. V. Paxson, "Automated packet trace analysis of TCP implementations," in *ACM SIGCOMM'97*, September 1997.
26. V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, Vol. 31, pp. 2435–2463, 14 December 1999. This paper is a revision of paper that previously appeared in *Proc. 7th USENIX Security Symposium*, January 1998.
27. C. Perkins, "Ad hoc on-demand distance vector (AODV) routing," Internet-Draft Version 00, IETF, 1997.
28. C.E. Perkins and E.M. Royer, "Ad-hoc on-demand distance vector routing," in *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, 1999, pp. 90–100.
29. J. Postel, "Internet control message protocol," Technical Report RFC 792, IETF, 1981.
30. J.B. Postel, "Simple mail transfer protocol," Technical Report RFC 821, IETF, 1982.
31. T.H. Ptacek and T.N. Newsham, "Insertion, evasion and denial of service: Eluding network intrusion detection," Technical report, Secure Networks, Inc., 1998.
32. P. Resnick, "Internet message format," Technical Report RFC 2822, IETF, 2001.
33. W.R. Stevens, "*TCP/IP Illustrated*, Volume 1, *The Protocols*," Addison-Wesley, Reading, Massachusetts, Vol. 1, 1994.
34. F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, Vol. 3, pp. 121–189, 1995.
35. R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building adaptive systems using Ensemble," *Softw. Pract. Exper.*, Vol. 28, No. 9, pp. 963–979, 1998.