NETWORK EVENT RECOGNITION

Karthikeyan Bhargavan

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy
2002

______________________________
Carl A. Gunter
Supervisor of Dissertation

______________________________
Benjamin C. Pierce
Graduate Group Chair
Abstract

NETWORK EVENT RECOGNITION
Karthikeyan Bhargavan
Supervisor: Carl A. Gunter

With the increasing reliance of computing on communication, it has become important to be able to rely on the correctness of network protocol implementations. Run-time monitoring has long been recognized as an effective technique to check that a protocol is safe: it does not do anything bad. We cast the protocol monitoring problem as an instance of network event recognition. While we are primarily interested in identifying error events that map to faults in the protocol implementation, in order to do so it becomes necessary to correctly reconstruct high-level protocol events as well. We outline the unique requirements of network event recognition, as opposed to program monitoring, and argue that existing network monitoring systems do not satisfy these requirements. To carry out network event recognition, it is necessary first to formally specify the protocol and error events we wish to recognize, and second to implement a run-time monitor that executes this specification. These two problems are inter-related: specification languages are constrained by efficient implementations. We evaluate existing specification languages and monitor implementations and make the case for a new domain-specific language. We present NERL, a language for specifying the run-time behavior of protocols and a suite of tools for checking and diagnosing faults. We demonstrate the language and tools on three case studies, finding new errors in existing software. These case studies empirically prove that network event recognition is an effective bug-hunting technique.
Contents

Abstract iii

1 Introduction 1
1.1 Network Event Recognition ........................................... 3
1.2 Summary of Contributions ........................................... 3
1.3 Outline ................................................................. 4

2 Background 6
2.1 Network Protocol Design Principles ................................. 7
2.1.1 Layering ............................................................... 7
2.1.2 Protocol Software Development ................................... 10
2.2 Protocol Testing Techniques ........................................... 12
2.2.1 Formal Verification ................................................... 12
2.2.2 Live Network vs. Simulation ...................................... 12
2.2.3 White Box vs. Black Box ........................................... 13
2.2.4 Active vs. Passive .................................................. 13
2.2.5 Ad Hoc vs. Randomized vs. Specification-based ............... 14
2.2.6 Statistical vs. Logical ............................................. 14
2.2.7 Conformance vs. Requirements .................................. 14
2.2.8 Online vs. Offline ................................................... 15
2.3 Passive Protocol Monitoring .......................................... 15
2.3.1 Passive Monitoring vs. Network Monitoring ................. 16

3 Monitoring Model 18
3.1 NER Architecture ....................................................... 18
3.1.1 Protocol Event Recognizers ....................................... 18
3.2 Monitoring Environments ............................................. 21
3.3 Strategies for Co-networked Monitoring ............................ 27
3.3.1 Co-networked Monitor Model ..................................... 28
3.3.2 Dealing with Loss .................................................. 31
3.4 Motivations for a new Monitoring Language ....................... 33
3.5 Tuning ................................................................. 33
3.6 Fault Origin Adjudication ............................................ 33

4 Passive Protocol Monitoring 35
4.1 Related Work ............................................................ 35
4.1.1 The Case Studies .................................................... 35
4.1.2 Passive Testing ....................................................... 36
4.1.3 Run-time Monitoring .............................................. 38
4.1.4 Network Intrusion Detection ..................................... 38
D TCP Event Recognition
  D.1 TCP Recognizer in NERL ........................................... 174
  D.2 Proof of Counting Property Optimization ......................... 188

Bibliography ................................................................. 189
List of Tables

3.1 FOA outcomes and their interpretations ................................. 34
6.1 NERL Exceptions ............................................... 53
7.1 NERL Static Event Slicing .......................................... 69
7.2 Trace-search Monitoring Algorithm ................................. 73
8.1 AODV Requirement: Loop Freedom ................................ 79
8.2 Properties from the AODV Standard Specification .................. 79
8.3 NERL Alarms for AODV .......................................... 81
8.4 AODV Simulation Trace $T$ ....................................... 84
8.5 AODV Property Violations Detected by NERL ......................... 84
9.1 Events and State Transitions for the MAIL Command ............... 98
9.2 Output Trace of the Mail Monitoring Stack .......................... 101
9.3 Time and memory usage for Expect trace analysis .................... 106
9.4 Time and memory usage for Multimail trace analysis ............... 107
10.1 Time and memory usage for TCP trace analysis .................... 121
10.2 Time and memory usage for Remaining TCP Properties ............. 122
10.3 Time and memory usage for TCP Monotonic ACK property ........... 122
10.4 Algorithm for checking P1 ....................................... 123
10.5 Time and memory usage for TCP Counting property ................ 124
List of Figures

2.1 An Internet Protocol Stack .............................................. 8
2.2 Five Software Artifacts .................................................. 10
2.3 Protocol Inputs and Outputs ............................................ 11

3.1 NER for SMTP ............................................................. 19
3.2 NER Recognizers ......................................................... 19
3.3 Anatomy of a Recognizer ................................................ 20
3.4 Monitoring Infidelities .................................................... 22
3.5 Monitor Placement ......................................................... 24
3.6 Buffers in the DUT ......................................................... 26

6.1 Ping Message Exchange .................................................. 44
6.2 Ping Receiver Specification .............................................. 45
6.3 Ping Sender Specification ............................................... 45
6.4 Ping Monitor Specification .............................................. 45
6.5 A Complete Ping Recognizer .......................................... 48
6.6 Ping Monitoring Modules ............................................... 54
6.7 Ping Modules Description .............................................. 57

8.1 The Verisim Framework .................................................. 77
8.2 Scenario S ................................................................. 83

9.1 Sample SMTP Session .................................................... 90
9.2 Recognizing Emails ...................................................... 92
9.3 Recognizing Emails ...................................................... 95
9.4 SMTP Server State Machine .......................................... 97

10.1 TCP State Machine ...................................................... 111
10.2 TCP Data Transfer States ............................................. 112
10.3 TCP Monitor Channels .................................................. 115

11.1 Co-networked Monitorable Properties .............................. 128
Chapter 1

Introduction

On April 25, 1997, nearly 40 percent of the Internet became inaccessible for periods between 20 minutes and 3 hours. Millions of users were affected. The error was traced to a mis-configured router managed by a small ISP in Virginia [BHW97]. In August 1999, bugs in software supporting a large commercial high-speed network affected 70,000 business customers over a period of 8 days. Among those affected was the electronic trading system of the largest U.S. futures exchange, which was shut down for most of the week as a result of the outages [How]. Such incidents not only revealed the extent to which individuals and businesses have become dependent on Internet services, but also showed up the unreliability of many of these services. This thesis investigates techniques for increasing the robustness of Internet services.

Internet services such as mail delivery, and encrypted data transfer, are provided by network protocols, standardized by the Internet Engineering Task Force (http://www.ietf.org). When the need for a new Internet service is recognized, network device vendors, such as Lucent and Cisco, and operating system vendors, such as Microsoft and Sun, form IETF working groups and design a protocol standard specification, which describes a distributed algorithm that provides the service. The individual vendors then implement these protocols in their hardware and software, and these implementations can inter-operate if they all conform to the standard.

Network protocols are layered. A protocol $P$ depends on the services provided by protocols in a lower layer $L$, and $P$ itself provides a service to a higher layer of users $U$. At the lowest layer, $L$ consists simply of network hardware: the ‘wire’, and the network interface card (NIC). At the highest layer, $U$ may be a human user, typing commands on a terminal. So, the designers of a protocol standard must make assumptions about the behavior of the users ($U$) as well as the correctness of the lower-layer protocols ($L$). The protocol $P$ is designed so that any implementation of $P$ will provide the required service, as long as it is executed in a protocol stack that correctly implements $L$, and has well-behaved users $U$.

Protocol software often malfunctions, failing to provide its service. The effects of a malfunction can range from a system crash, if the malfunction occurs on a desktop operating system, to an Internet-wide outage, if it occurs on a backbone network device. Given the protocol software design cycle outlined above, there are several reasons why an implementation may malfunction.

- The standard describes a faulty algorithm that fails to provide the service in some cases.
- The implementation is buggy, and does not conform to the standard.
- The users $U$ are badly behaved, and produce unexpected input. Such users may either be malicious attackers or malfunctioning higher-layer protocol implementations.
- The lower-layer $L$ malfunctions, and fails to provide the service that $P$ depends on. Lower-layer errors may be due to incorrect implementations, or network mis-configurations.
When a protocol implementation malfunctions, it is important to detect the error, as well as understand its cause. If the implementation is at fault, then a bug report must be made out to the software vendor. On the other hand, if the standard is incorrect, then the standards group must be notified. A mis-behaving user may indicate a malicious attacker from whom private assets must be protected, while network mis-configurations should be forwarded to system administrators for correction.

Incorrect standards have design errors that must be detected and addressed early in the protocol development cycle. Protocol verification has been an effective technique to identify these errors [BOG02]. Bugs in implementations are typically found through protocol testing, using tools such as network simulators. Users and lower-layer protocols, on the other hand, are dependent on run-time factors, such as network configuration and load, and user behavior patterns. This makes them unpredictable at design time, and therefore difficult to test. Incorrect behaviors by users often present a security threat to the network. Such behaviors are classified as attacks and are detected by firewalls and intrusion detection systems. Network mis-behavior or mis-configuration is detected by network management software. In this thesis, we will concern ourselves with the protocol testing problem.

Traditionally, protocols have been tested using active techniques. First, a test network topology is constructed and artificial users are programmed. Then in this controlled network and user environment the protocol implementation is probed with specific sequences of inputs and each resulting sequence of outputs is checked against a pre-generated expected output sequence. Although active testing can be effective in finding errors in early prototype implementations, it suffers from several disadvantages. First, active testing is inadequate for testing an implementation against unpredictable user and lower-layer protocol behavior. Second, the high degree of control over the network required for active testing is available only in expensive network test-beds or virtual simulated environments. So, active techniques are inapplicable for testing implementations "in the field" after deployment.

On the other hand, passive protocol monitoring is used to test protocol implementations at run-time. A passive monitor runs on the same network as the implementation under test and checks it for errors as it executes. So, when an implementation malfunctions, it is possible to find the cause of the error and propose a corrective measure as feedback, instead of the program simply crashing. This technique has been used to good effect on telecommunications software, such as software running on telephone switches. The primary advantage of the passive monitor is that it is unintrusive - it imposes no conditions on the network or user behavior and has no side-effects. So, it may be used in any network be it a testbed, network simulation, or an operational network. Moreover, passive monitoring may be used as an additional module in any test environment, even alongside active techniques.

Although passive monitoring systems have been proposed for a wide variety of tasks from self-checking distributed systems [FP76, SCA] to network intrusion detection [Gro01], there have only been rarely used for protocol testing because of the difficulty of programming such monitors. While active tests simply involve comparing the observed output sequence with an expected output sequence that has been computed beforehand, the passive monitor must compute these expected outputs on-line because it does not control the sequence of inputs. Moreover, the passive monitor has limited visibility - it can see only the low-level messages on the network, not the high-level interactions between the implementation and the user. In some conditions the passive monitor may even experience trace infidelities - the message sequence it sees on the network are not the same as the actual sequence at the implementation under test. Finally, when the passive monitor flags an error it may not be straight-forward to map this error event to a violation of the specification, and then to a flaw in the implementation.

In light of these limitations, we identify four requirements for a passive monitoring system to be effective.

**Expressiveness.** The ability to write a monitor to test a protocol at any level in the protocol stack.
Correctness. Confidence that the errors produced by the monitor indicate violations of the specification.

Diagnostics. Tools and techniques for interpreting errors and guiding the user to flaws in the implementation.

Efficiency. The ability to monitor long executions of several protocol instances on-line in a typical test network.

Previous work on passive testing tried to address the expressiveness and correctness issues by generating monitors from the protocol specification. This ensures that the resulting errors can be mapped to specification violations, but the generated monitors are slow because they have to deal with non-determinism in the specification. Moreover, they provide very little diagnostic information about the cause of the error. As a result, these techniques have not been applied to the testing of full-scale stacks of Internet protocols. Our approach is to allow the user to program protocol monitors directly and then provide tools and techniques to map this monitor program back to the specification.

1.1 Network Event Recognition

We present a programmable passive protocol monitoring system called Network Event Recognition (NER), for analyzing network protocol software. We claim that NER satisfies all the requirements listed above. We demonstrate that NER is an effective tool for finding bugs in protocol code.

We introduce a domain-specific language, NERL, that is used to program run-time monitors based on protocol specifications. A NERL recognizer analyzes an event stream and produces alarm events when deviations from the specification are detected. NERL recognizers can be used to passively monitor protocol implementations in real-time as they execute on a live network. In addition, recognizers can also be used to analyze packet traces captured from a network simulation or live network.

To monitor a protocol at a high layer in the stack, a passive monitor must model all lower layers in order to reconstruct the messages at the protocol of interest. We introduce a module language NERL_MOD that is used to specify and manage stacks of recognizers where each recognizer may be written in NERL or C.

We demonstrate the applicability of NER on three case studies. We first analyze network simulations of a wireless routing protocol: AODV, and find bugs in the AODV implementation. Then we analyze live sessions of Internet mail servers running SMTP and find several errors in popular mail server software. Finally, we use NER to analyze TCP implementations. We demonstrate that the NER framework and tool-set is flexible and effective enough to deal with trace infidelities at the monitor.

1.2 Summary of Contributions

Network Event Recognition Framework We present the first programmable passive monitoring system designed for the testing of network protocols.

NERL We present a strongly-typed domain-specific language NERL to program specification-based run-time monitors for network protocols. Earlier works on event recognition either used ad hoc scripts, which were prone to simple errors, or used specialized pattern languages, which were not expressive enough for protocol specifications.
NERL\textsubscript{MOD} We present a module language NERL\textsubscript{MOD} to program stacks of protocol event recognizers.

**Tools and Techniques** We present several automated tools and analysis techniques that help in writing correct recognizers and in diagnosing implementation errors.

**Protocol Testing** In addition to network event recognition, we also demonstrate new protocol testing techniques for simulation analysis and live monitoring.

**Simulation Analysis** We show how NERL can be used to analyze simulation traces, and find bugs in protocol code. We demonstrate bugs in independently written simulator code for a wireless routing protocol (AODV). We are not aware of any other formal specification-based tool that attempts to find bugs in simulator code for network protocols.

**Dealing with Infidelities** We show how the formal NERL framework can be used to address inconsistencies between the traces observed at a monitor and at the device under test during live monitoring. These inconsistencies have been recognized by previous work as a serious problem for passive monitoring systems.

**Case Studies** We analyze protocols at three different layers in the Internet stack and find errors of interest to protocol developers and implementors.

- **AODV** We use NER to find 5 different flaws in the simulator implementation of AODV. In addition, we find 1 flaw in the AODV (version 0) standard. We demonstrate the efficiency of the NERL language and the effectiveness of the diagnostic tools and analysis techniques.

- **SMTP** We use NER to find 6 different flaws in the SMTP implementations in three different popular mail servers. We demonstrate the value of correctness checking for SMTP monitors.

- **TCP** We use NER to confirm the existence of 1 flaw in two TCP implementations and its absence in a third implementation. We demonstrate the flexibility of the NER architecture and languages.

### 1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 provides a general background to the network protocol design process and conventional techniques for finding errors in protocol implementations. Readers familiar with the protocol design process may skip this chapter. Then, in Chapter 4, we formally define the problem addressed by this thesis and we survey related research that parallels or motivates our work.

The Network Event Recognition framework and the NERL and NERL\textsubscript{MOD} languages are defined in Chapter 6. We define the syntax and semantics for these languages and give examples of their use. Then, in Chapter 7, we describe the NER tool-set. We outline the implementation of the compilers for NERL and NERL\textsubscript{MOD}. In addition, we describe correctness and diagnostic tools that aid in programming recognizers and analysis techniques that are used in analyzing error events. Readers interested in the details of the compiler and tool implementations may choose to read these chapters in sequence. However, the motivations and applications of the tools and techniques will become clear only in the case studies. So, we recommend that the reader skims Chapter 7 and goes on to read the case studies first, referring back to Chapter 7 as new tools and techniques are introduced.
The next three chapters contain case studies. In Chapter 8, we apply NER to simulation traces of a wireless routing protocol, AODV. In Chapter 9, we analyze live networks running the mail delivery protocol SMTP. In Chapter 10, we address the passive monitoring of TCP implementations. These chapters demonstrate the use of several of the tools and techniques described in Chapter 7. Finally, in Chapter 11, we evaluate the results in this thesis and outline future work.
Chapter 2

Background

The Internet is a medium of communication. It provides users with several data transfer services. For instance, the mail transfer service takes an email from a sender and delivers it to a recipient across the Internet. Users never see how this is accomplished; they write and receive emails using friendly mail user agents such as Pine or Outlook. This service is actually implemented by two software mail transfer agents, one at each end-point, that interact with each other to transfer an email.

Implementing communication services uniformly is a challenge, because of the inherent heterogeneity of the Internet. For instance, mail transfer must work even if the sender is using a Windows XP desktop connected via a phone line to AOL and the recipient is reading mail from a Solaris server connected to the Internet through an T1 link. So, the possibly proprietary mail transfer agents included with Windows XP and Solaris need to be capable of inter-operating, across various kinds of links, in order to transfer email.

The Internet’s approach to this problem of interoperability is embodied by the Internet Engineering Task Force (http://www.ietf.org). For each service, the IETF publishes a network protocol standard—a specification document detailing how networked devices must interact with each other to implement the service. For instance, the SMTP standard [Pos82] describes the operation of a client MTA C and a server MTA S, and the format and sequence of messages that are exchanged between C and S, enabling C to deliver an email E to a recipient user@S on the server.

Operating system vendors and network device manufacturers can then produce hardware and software that implements these network protocols. The implementations themselves are allowed to be proprietary and optimized for specific platforms, as long as they conform to the protocol standard. The idea is that standard-conformant implementations are guaranteed to interoperate correctly to provide the required service.

Our interest is in checking whether a network protocol implementation fails to provide the service that it claims. For proprietary implementations, often the only information available about implementation behavior is the sequence of standardized messages exchanged between protocol participants. We aim to analyze such message sequences to discover service failures and to diagnose the faults in the implementation that cause these failures. For instance, the most commonly used SMTP implementation, Sendmail, is known to have several bugs that cause it, in some cases, to fail to deliver email. Given an SMTP message sequence we want to check if such errors have occurred, and if possible to find the bugs in Sendmail that caused the errors, even if the source code is not available.

In this chapter, we will describe the nature of flaws in a protocol implementation, and give a general background of techniques used to identify and protect against these flaws. In the following section, we discuss some commonly used network protocol design principles and outline our research goals. In section 2.2, we classify prevalent testing methodologies for network software,
and in section 2.3 we argue in favor of a passive protocol monitoring approach.

2.1 Network Protocol Design Principles

Several guidelines have influenced the development of Internet protocols. In this section, we first describe protocol layering—the modularity principle that governs how different protocols work together. Then we use a software engineering model, WRSPM, to describe stages in the design of a single protocol layer. We identify vulnerabilities in each stage of protocol design.

2.1.1 Layering

Network protocols provide services. For instance, consider the mail delivery service: deliver email \( E \) to recipient user@domain. This is implemented by the Simple Mail Transport Protocol (SMTP) \[Pos82\]. A user invokes an SMTP client with a destination address and an email file, and the protocol will make the best effort to deliver it. For data transfer, SMTP depends on another network service: reliable in-order transfer of data \( E \) to destination domain, and this service is provided by the Transmission Control Protocol (TCP). In turn, TCP depends on packet delivery: find a path and deliver packet \( P \) to domain, which is provided by IP and its routing protocols, and so on. Therefore, it is useful to think of protocols as software modules, arranged as layers in a stack. Higher-layer protocols depend on services provided by protocols below them.

This architecture has been formalized by the seven layer OSI model \[H.Z80\]. In practice, operating systems rarely implement all seven layers. Moreover, the implementations of several of the more commonly used protocols are often bundled together for efficiency. Nevertheless, logically, a typical Windows or Linux machine contains a network protocol stack, as shown in Figure 2.1, with protocols arranged in several layers. In the figure, each box represents a protocol and arrows represent dependencies on lower-layer protocols. Solid arrows show typical relationships; dashed arrows describe optional layering schemes.

**Link Layer** A network consists of network devices, such as computers and routers, connected by a common link. Broadcast links are shared by many computers, while point-to-point links join two computers. The link layer consists of protocols that provide the service: deliver data \( D \) from network device ND1 to ND2 (through the link L that they share). Network Interface Cards (NICs) and their drivers typically offer this service.

Ethernet has traditionally been a popular link layer protocol for communication across local area networks (LANs) connected through shared cable. So, even newer link technologies, such as bridged LANs and wireless networks, support Ethernet. On the other hand, PPP is commonly used across point to point links such as phone lines.

**Network Layer** An internetwork (such as the Internet) consists of several networks of devices, where some devices (routers) are connected to two or more networks. This leads to a bipartite graph consisting of devices and links. The link layer provides ‘local connectivity’ for devices attached to the same link. The network layer provides connectivity across links attached to the same device. The service provided is: deliver data \( D \) from network device ND1 to network device ND2, where ND1 and ND2 may be connected by a path of two or more links and devices.

In the Internet, the network layer services are provided by the Internet Protocol (IP) and a suite of associated protocols that are used to gather information about network topology and errors. IP routing protocols such as OSPF and RIP try to find paths from one device to another. IP itself runs on routers and end-points, and carries out the actual delivery of data packets along a discovered path: it uses the link layer protocols to transmit packets across a
Figure 2.1: An Internet Protocol Stack
link to the next router on the path. If a router is asked to deliver packets that are too large for the link medium, it breaks up the packet into smaller fragments that are then reassembled at the destination.

In order for IP to use the link layer, device names must be translated between those used by IP and those used by the link layer. ARP is a protocol that does this translation for Ethernet. To communicate network layer connectivity and error information, routers and endpoints use ICMP. For instance, the ping program uses ICMP to check whether a device is willing to provide the network layer service.

**Transport Layer** Internetworks are unreliable. Devices and links may experience crashes, over-load and re-configuration. Consequently, data transmitted across a network using IP may be lost, delayed, duplicated, re-ordered, and even corrupted. IP only promises that it will make the best effort to deliver data. For many applications, this level of reliability is inadequate. Transport Layer protocols attempt to enhance IP delivery to suit different application requirements. So the transport layer provides service such as reliable data transfer, in-order data delivery, confidential data transfer, and authenticated data delivery.

TCP provides reliable, in-order data transfer and protects the network against congestion. TCP simulates a connection (much like a telephone conversation) on top of the connectionless network layer. In addition, it allows several applications to share a single network connection by multiplexing the network access between them. SSL (or TLS) enhances TCP with cryptographic primitives, to provide both confidential and authenticated transfer. We treat SSL as a transport layer protocol, although it runs on top of TCP, because the service it provides is a transport layer service. On the other extreme, UDP is a minimal transport layer protocol—it adds multiplexing to IP and protects against packet corruption.

**Application Layer** Application layer protocols provide high-level Internet services, such as email, remote login, file transfer, and web surfing. Depending on the level of reliability that each service needs, a transport layer protocol is chosen, and the application layer protocol uses it to exchange messages across the Internet to provide the high level service.

For instance, SMTP, TELNET, FTP, and HTTP use TCP to provide email transfer, remote login, file transfer and web surfing. The Networked File System (NFS) protocol provides a virtual file system using the RPC protocol that in turn runs over UDP. Moreover, there may be several layers of applications. For instance, the IMF standard defines the format of emails exchanged over SMTP, and the MIME standard describes the payload of the IMF messages that include attachments. We say that MIME runs on top of IMF that runs on top of SMTP. The application layer experiences the most changes in the protocol stack with new protocols for new services being designed every day. For instance, instant messaging has been gaining in popularity and several new competing protocols provide this service. In addition, traditional protocols such as HTTP and SMTP can now operate over newer transport protocols such as SSL.

This protocol stack is often called the IP stack because all protocols at and above the network layer are based on IP. This gives a reference point at which to start our analysis of protocols in the stack. In this thesis, we will concentrate on the analysis of network, transport and application protocol software. We choose to ignore the analysis of link-layer protocols because of their wide variety and low-level implementations. Each link layer protocol interfaces with different kinds of communication hardware. Therefore analyzing link layer protocol implementations would require the capability to extract raw bits from each kind of link - phone line, cable, wireless. On the other hand, network layer protocols only have to deal with IP packets. Moreover, the link layer is implemented using both hardware and low-level drivers by network interface card manufacturers. In contrast, the network and transport layers are implemented as high level software as
part of each operating system, while application layer protocols are implemented as third-party software and are often executed in user-space.

Designing protocols as layers delivers a clean and modular separation of services. As a result, each individual protocol can be simpler, relying on the correctness of the lower-layer protocols it depends on. It also makes the analysis of protocol software simpler. To check whether a protocol software stack is operating correctly, all we need to check is that each individual protocol layer is correct. In the next section, we will discuss the analysis of single protocols.

2.1.2 Protocol Software Development

A software project generates a variety of artifacts - code, documentation, and orally communicated (or uncommunicated) assumptions. It is helpful to use a ‘reference model’ for these artifacts as a foundation for classification and analysis. Gunter et al have proposed a general model called WRSPM [GGJZ00], as an extension of the work of Jackson and Zave [JZ92, JZ95, ZJ97], and applied it in a network protocol case study [BGG+98]. We use it again in this section as a strategy for characterizing the issues in network protocol design.

The WRSPM reference model consists of five artifacts classified into two overlapping groups as depicted in Figure 2.2. Here W, the ‘world’, describes assumptions about the operational environment, and R represents a set of requirements to be met by a program. The goal of a programming project is to produce a program P that satisfies the requirements R when it is run on a programming platform M in an environment that satisfies the restrictions W. The role of the specification S is to provide enough information for a programmer to build such a program. S tries to guarantee that a program that complies with it also satisfies the requirements R.

For instance, suppose we are designing a protocol NP that must provide a service NS. Figure 2.3 shows the inputs and outputs of a single protocol. The protocol receives commands from the higher-layer protocol that demand a service. The protocol implements the service by sending and receiving messages through the lower layer, and issues responses to the user. To ensure timeliness the protocol must also keep track of timer events managed by the operating system.

Then, W contains assumptions about the behavior of ‘users’ of the service NS - the command sequences that they are allowed to generate. R is a formal description of the service NS itself - the responses that it must produce for each command. S is the protocol standard specification - it describes, in reasonable detail, the algorithms and protocol processes in any implementation of the NP protocol. For instance, it may describe two NP processes: the client and the server, the format of the messages they exchange, and the state machines at both. P is the actual protocol software that implements S - the NP protocol implementation. P depends on the availability of several ‘link’ services provided by the network hardware and lower-layer protocols; all assumptions about these lower-layer services are contained in M. For instance, M may specify that messages sent by the client NP process must eventually reach the server NP process.

The aim of the design process is to ensure that the protocol implementation P is robust: it provides the required service R, and is inter-operable with all other implementations of S. To ensure robustness, we must prove that when higher-layer protocols conform to W, and the lower-layer protocols conform to M, then (1) S guarantees the service described in R (W, P, and S provide R),
and (2) $P$ conforms to the specification $S$ ($W$, $P$ and $M$ satisfy $S$). This indicates several vulnerabilities that may cause a network service to fail.

**Incorrect Design** $S$ fails to guarantee $R$.

**Buggy Code** $P$ fails to conform to the standard $S$.

**Bad User** The user (higher-layer protocol) above the program $P$ violates the assumptions $W$.

**Link Failure** The link (lower-layer protocol) below $P$ fails to provide the services specified in $M$.

For instance, consider the mail delivery service described before. SMTP is a standard specification ($S$) that is said to provide mail delivery ($R$). However, it is possible that the specification does not take into account some series of interactions, or some rare error conditions. That would be an instance of incorrect design. Such errors are typically identified early in the design phase. Sendmail is an implementation ($P$) of SMTP. However, Sendmail fails to conform to SMTP for some kinds of invocations because of bugs in Sendmail code. A Sendmail user is expected to provide a valid email message and a well-formatted envelope containing sender and recipient addresses. A user who violates this assumption ($W$) will cause Sendmail to fail. Finally, SMTP relies on a correct implementation of TCP, and consequently a correct implementation of IP, and also correctly functioning network hardware. A violation of any of these assumptions can cause mail delivery to fail.

If we assume that the lower and higher layer protocol implementations are separately analyzed and are correct, then we only need to check for buggy code and incorrect design. Given a protocol implementation, specification, and requirements we are interested in the following questions:

**Conformance** Does $P$ conform to $S$?

**Functionality** Does $P$ provide $R$?

**Correct Design** Does $S$ provide $R$?

Although functionality is implied by conformance and correct design, we list it separately because it is an essential property of the implementation irrespective of the correctness of the specification. Each of these properties can be checked independently.
Traditionally, protocol testing has been used to answer the conformance and functionality questions. To test a protocol implementation \( P \), we execute \( P \) under several conditions - different kinds of users and different network behaviors - and check whether \( P \) behaves correctly. Protocol verification is used to check whether a design is correct. Although implementations are too complex (have too many states) to carry out a complete proof of conformance and functionality, specifications are often cleaner and easier to abstract, allowing proofs of correct design or generation of counter-examples if the design is incorrect. For instance, Bhargavan et al use protocol verification techniques to develop a formal proof of correctness for the RIP routing protocol [BG02b]. On the other hand, they show that the AODV routing protocol is incorrectly designed, and demonstrate counter-examples as proof.

In this thesis, we are primarily interested in implementation conformance and functionality. But when an implementation is shown to fail to provide the service \( R \), we will also be interested in whether the failure is due to a buggy (non-conformant) implementation or due to incorrect design.

### 2.2 Protocol Testing Techniques

Testing network protocols is quite difficult when compared with sequential program testing. This is because a protocol describes interactions in a concurrent, distributed system, involving participants on multiple machines across a dynamic network. Protocol participants must respond to varied user requests as well as a wide range of network behavior. Designing an adequate test suite is difficult. Furthermore, understanding the results of a test is a challenge. Formal methods have been suggested as solutions to understanding the testing process for network protocols [SCB91].

In this section, we survey several techniques for functionality and conformance testing of protocol implementations and define our methodology. Protocol testing research is fairly large and varied, and most testing techniques are customized to particular protocols. We restrict our attention to general testing frameworks and techniques that can be applied to multiple protocols. In addition, we only look at techniques that can be formally specified and analyzed. Even this restricted body of research has been extensively surveyed [CFP93, LY96, BP94, SCB91].

#### 2.2.1 Formal Verification

When the requirements and specification of the protocol are specified in a formal language, we can attempt to develop a mathematical proof of conformance and functionality. Formal verification techniques attempt to automatically generate such a mathematical proof, or provide a counter-example if the implementation is incorrect. Verification techniques have been successfully applied on telecommunication code using tools such as Verisoft [God97] and FeaVerHolzmannS02.

Although such a complete analysis is attractive, it is typically difficult to accomplish because of state space explosion: the implementation is too complex to analyze automatically [Hol91]. So to apply verification techniques, the implementation must be abstracted to a manageable model. Although automated abstraction is possible for implementations written in a restricted style [GS02], it is not applicable to general Internet protocols implementations.

#### 2.2.2 Live Network vs. Simulation

In a live testing framework, the developer constructs a network topology of several devices running the protocol implementation under test (IUT). Then the implementation is tested by executing it for a sequence of upper-layer commands and checking the generated responses. Sometimes, a device running an IUT may be connected to another running a dummy implementation that sends test message sequences to the IUT.
On the other hand, in a network simulator, although the IUT thinks it is interacting with other nodes over a complex topology, in reality all the network devices are being simulated on a single machine. For instance the network simulator NS [Pro, FV00] takes as input a C++ protocol implementation along with a scenario describing a topology and characteristics of lower and higher layer protocols. NS then generates a randomized simulation of the network and produces a detailed trace of all the events related to the IUT.

The advantage of the network simulator is that it avoids the effort of maintaining several testbeds. Different network configurations can be designed and tested quickly. Prototype implementations can be written for the simulator to test implementation techniques early in the development cycle. Moreover, the simulator gives the user considerable control over the test environment. The user can control the characteristics of the lower layer and higher layer protocols. In addition, the user gets a god’s eye view of the entire network - the simulator logs all the events at each protocol layer as they happen.

The main advantage of live testing is that it checks a real implementation that is going to be deployed, while the simulator implementation is often not executable on a real network device and represents only a prototype. So, some live testing is always included in the final stages of protocol implementation development.

2.2.3 White Box vs. Black Box

White box testing refers to tests in which the internals of the protocol software are available. One can instrument, inspect and execute different parts of the software during the test. This allows the test to extract a considerable amount of information about the behavior of the software. For instance, a network simulator is a white-box testing environment. In black box testing, the protocol software is treated as a “black box” - its internals are invisible and one can observe its behavior only by giving it inputs and inspecting its outputs.

Black box testing is less powerful than white box testing, but is often made necessary by the unavailability of implementation source code, and by the need to uniformly test software written in multiple languages. Black Box testing can be effective only if the software has well-defined input-output specifications. Fortunately, network protocols always have well defined input-output specifications because of the need for standardization.

2.2.4 Active vs. Passive

An active test specifies both inputs to send to the IUT and the outputs that must be produced by the protocol software. For instance, a test may specify sending $i_1$ to $P$, waiting for $o$ and then sending $i_2$ and so on. In passive analysis on the other hand, the test assumes that some other module is providing the inputs and monitors only the outputs coming from the software. In a way, active testing can be thought of as a combination of test suite generation followed by passive analysis of the test outputs [BB89].

The chief advantage of active testing is in its control over the test environment. So it is possible for an active test suite to provide guarantees of coverage - the space of behaviors that it checks. Passive tests cannot provide any such coverage guarantees because they do not control the inputs. On the other hand, passive testing is unintrusive - it is invisible to the IUT - and it can analyze arbitrary traces, unprejudiced by the design of the test. While the vast majority of protocol testing research is on active testing techniques, passive testing is preferred when the developer wants to test the IUT after deployment in a network environment that is beyond his control [LNS+97]. We describe relevant passive testing research in more detail in the next chapter (4.1.2).
2.2.5  Ad Hoc vs. Randomized vs. Specification-based

Initial testing of protocol software is done by ad-hoc test scripts that check that it ‘works’ on a few sample test cases. Randomized testing involves making a model for the inputs to the protocol, and generating randomized input traces for it to respond to. The outputs of the protocol are then checked for incorrect behavior. For instance, a network simulation is a randomized testing technique. In specification-based testing, the tests and test-checkers are automatically generated from formal specifications of the protocol.

While ad-hoc testing is valuable only as long as it finds errors in the IUT, randomized tests can provide some coverage guarantees over the space of possible test traces. When possible, specification-based testing is preferred to both because it can guarantee that various parts of the specification have been checked, and every error that is discovered can be mapped to a property violation in the formal specification.

2.2.6  Statistical vs. Logical

Network protocol implementations often form the bottleneck in a fast network and so they are engineered for high performance. Consequently, most of the testing for core protocols involves statistical analysis: running the protocol for a long time and checking that its performance matches a theoretical profile. Logical analysis, on the other hand, follows the logical behavior of the protocol and compares it with a formal model of how it should behave, producing alarms every time an unexpected event occurs.

Statistical analysis of protocol behavior is useful for identifying several of the problem areas in the IUT and for profiling its expected behavior. Moreover, it can be easy to run fast automated statistical tests. On the other hand, logical analysis is more cumbersome, takes more time, and sometimes provides a lot of extraneous information that the developer needs to wade through.

However, performance measurement does not identify all errors. For instance, Suppose that a routing protocol also has a security requirement that a packet at a node $n_1$ meant for a neighboring node $n_2$ will never be seen by a third node $n_3$. If this property is violated, the hit on performance is likely to be small but one would still like to know if the property is violated. For one, even if the error occurs rarely, its effects might be catastrophic. Second, such a low profile error may become important in other protocol and network configurations. Logical testing is capable of finding all such errors that occur in a run of the protocol, and should always be carried out before or during performance analysis. Even if one were only interested in performance, performance profiles of buggy implementations can be misleading.

2.2.7  Conformance vs. Requirements

Conformance testing checks that protocol software conforms to the standard. It checks that every trace that the software produced could have been generated by the specification. Specifications are typically written as state machine descriptions in a formal language, such as Lotos [Lot87]. Conformance can then be cast as state machine inclusion, and several algorithms have been developed for this framework [LY96].

Requirements (or functionality) testing checks that the protocol software satisfies its high-level requirements and provides the service that it promises. Protocol requirements can usually be expressed as formulae in a temporal logic, such as GIL, or in a real-time algebra. These formulae can then be used to generate tests or to construct test oracles that check if a test trace produced by the protocol software satisfies the requirements.

Testing conformance is essential to ensure protocol interoperability. Moreover, if the specification has been correctly designed, a conformant implementation is guaranteed to be functional. However, functionality testing is valuable when either a formal specification does not exist, or is
too complex for conformance testing. Testing functionality is often simpler, and sometimes it can be used in conjunction with conformance testing to find errors in the specification [BGO00].

2.2.8 Online vs. Offline

In online testing, test results are computed as the protocol is being executed. In offline testing, the protocol responses are logged as the test is carried out, and analyzed later.

Offline analysis is more powerful in that it can carry out multiple passes on the test logs, while online analysis must be done ‘on the fly’. Offline analysis is also more suitable in simulation environments where the timeliness of error reporting is not important. On the other hand, online analysis is more suitable in a deployed network because it can recognize errors as they occur and alert the system administrator. An efficient online analysis tool can also be used offline, but the reverse is not true.

2.3 Passive Protocol Monitoring

The choice of the testing framework for a protocol implementation is subject to a number of external factors, like the layer of the protocol, programming language used for the implementation, modularity of the implementation, availability of source code, and feasibility of formal specification. A general protocol testing framework must be flexible with respect to these criteria.

There are two aspects to any testing framework: generating test cases, and analyzing test results. Although test-case generation is an interesting and worthwhile topic of research, in this thesis we will restrict our attention to test result analysis. In particular we will develop a framework for passive, black box, logical, online, formal specification-based testing for both conformance and requirements. One motivation for this restriction is that the resulting techniques are applicable in all stages of protocol development and even after deployment. Another motivation is that by restricting ourselves to this framework we gain the maximum flexibility in terms of implementations and environments where our techniques are applicable. Our framework will be applied in simulation testing, as well as live run-time network monitoring; to implementations where source code is available and to proprietary protocol implementations for different operating systems. Hereafter, we will refer to this class of testing as passive protocol monitoring. We realize our framework through a technique we call Network Event Recognition, that we will describe in detail in Chapter 6.

A passive protocol monitor is a software module that inspects all the data going into and out from a device running a protocol, and attempts to logically reconstruct the behavior of the protocol implementation. It uses a formal specification of the protocol to carry out this reconstruction and to identify errors in the implementation. The advantages of such an approach are as follows.

- It is unintrusive, which makes it deployable in many situations, and in parallel with other testing techniques. It can be used with simulations, or live network test-beds, or even deployed in a running network to point out flaws as they occur.
- It does not require implementation source code, and so is also applicable to proprietary protocol implementations.
- It runs on a separate machine, and so can monitor several protocol implementations (on different machines) at the same time.
- It provides formal guarantees; if a test is passed, then all the logical properties checked in the test hold for that execution of the protocol. If a test fails, then the failure can be mapped to a violation of the formal specification.
In a live network scenario, the recognizer runs on a separate machine on the same network as the device under test. It attempts to recognize protocol events by looking at data ‘sniffed’ from the network that is going to or coming from the device under test. There are several limitations to this framework that become apparent.

- The recognizer only sees one end of the input-output behavior of the protocol (the lower-end). It can inspect the inputs and outputs between the network and the protocol, but not the commands and responses between the user and the protocol, or the timer events produced by the operating system (Figure 2.3).

- The input-output trace often does not have enough information to characterize the implementation behavior. The implementation may have secret state that is not evident in the messages produced, which would make reconstructing its behavior difficult, if not impossible. This problem can occur if the protocol has ambiguities, causing the specification to be non-deterministic, or if the implementation were using some cryptographic primitives for secrecy.

- The monitor is susceptible to network errors. The monitor is connected to the device through a network link, which makes it vulnerable to packet loss, delay, duplication etc. between the monitor and the device. The packet trace captured by the monitor is then said to contain infidelities.

To address the first limitation we shall develop programming techniques to reconstruct the invisible high-level command and responses from the observed low-level messages. But the second limitation is the most serious. For instance, without the relevant keys, we cannot hope to learn anything by passively monitoring an encrypted packet trace. In this thesis, we will restrict our attention to protocols whose messages contain enough information about the state for us to be able to reconstruct the protocol state, timers, and higher-layer commands and responses. In later chapters, we demonstrate several important protocols for which the messages do contain enough information. We analyze these protocols using passive monitoring and demonstrate its effectiveness. Finally, to deal with network errors, we propose several algorithms and techniques for passive monitoring that attempt to find protocol errors even in traces with infidelities.

2.3.1 Passive Monitoring vs. Network Monitoring

Some tools that are loosely referred to as network monitors are not really related to our notion of passive protocol monitoring. In particular, we distinguish our work from network management systems, network surveillance tools and protocol analyzers.

The goal of network management tools is to monitor a network and gather statistics about the performance of key network and transport protocols. These statistics are then presented in a readable manner to a network administrator who uses them to diagnose the health of the network and to identify problem areas and protocols. The administrator can then change the internal network topology or protocol accessibility accordingly. Network management is not specification based and it carries out statistical (not logical) analyses. The IETF has standardized the Simple Network Management Protocol [CFSD90] that runs on every router and network device and provides a uniform way to query the device for protocol statistics. When enhanced with Remote Monitoring [Wal95], SNMP provides a programming interface that can be used to set up a customized passive monitoring system. However, network management systems are typically used only for the performance analysis of low-level network and transport protocols, and not for detecting implementation errors.

On the other hand, network surveillance tools are used to capture emails and web sessions of users, especially if they violate some network policy. For instance, the FBI’s Carnivore system [SCHP+00] is used to capture emails, web and file transfer sessions of ‘suspects’ for whom
a wiretap warrant has been issued. Such systems do not analyze protocols in detail, they simply capture sessions and filter them with regular expressions.

Protocol analyzers such as Ethereal [Eth] parse packet traces using their knowledge of packet formats for a large number of protocols. They carry out no ‘stateful’ analysis or the protocol event trace. Indeed, when these analyzers claim to understand protocol P, it means that they understand P’s message formats and not its state machine.
Chapter 3

Monitoring Model

3.1 NER Architecture

The goal of a protocol monitor is to recognize protocol events. At the lowest layer, a protocol event is as simple as a packet $P$ has been sent from $A$ to $B$. Higher layer data like emails, however, are broken up into several packets, and some of them may be resent for reliability. So at higher layers, recognizing complex events like an email $E$ has been sent from $U$ to $V$ may involve capturing a number of packets, correlating them and putting the data in these packets together. We call this hierarchical process of recognizing protocol events at different layers Network Event Recognition (NER).

Figure 3.1 shows how network event recognition would work for the SMTP protocol. The sender sends an email to the receiver, which is broken down into SMTP commands, and then into TCP segments and IP fragments. A monitor that spies on this session needs to put together the packets, segments and commands in order to be able to analyze the behavior of the SMTP implementation itself.

Effectively, at each layer in Figure 3.1, the protocols at the sender and receiver sit on an abstract channel over which they send messages. The function of the network event recognizer is to first reconstruct the messages sent over this channel, and then check the messages for conformance with the protocol specification or requirements.

The channel at the bottom is the physical interface with the devices under test (DUTs), which are running the protocol implementations we are monitoring. When NER is used to monitor a live network, this interface is the ‘wire’ on which the monitor is ‘sniffing’ (listening) for interesting packets. On the other hand, when NER is used to analyze network simulations, the interface between the monitor and the implementation is the simulation trace file. The higher-layer channels are reconstructed by succeeding layers of protocol recognizers.

While Figure 3.1 works for checking a single email, ordinarily we will want our monitor to check several SMTP sessions concurrently. So, we want the SMTP monitor to look as in Figure 3.2, where each module at each layer analyzes events corresponding to one connection of one protocol.

3.1.1 Protocol Event Recognizers

Each box in Figure 3.2 is an event recognizer. It must carry out two kinds of tasks: error event recognition - checking the protocol message sequence at the current layer - and meta event reconstruction - reconstructing the messages on the higher-layer channel (Figure 3.3).

The inputs to the recognizer are the messages on the current channel - meta events produced by the lower layer - which it must parse into its internal format to generate primitive events. This parsing step takes the payload the lower layer and imposes some structure on it to extract fields.
specific to the current layer. For instance, TCP ports are embedded in the IP payload and must be extracted by a TCP event recognizer. If the payload is a text string, then the structure imposed may be a regular expression pattern or a grammar. For instance, SMTP commands are embedded as lines of text in TCP streams and must be parsed using a standardized grammar.

At the lowest layer, there are two kinds of primitive events of interest:

**Primitive Network Events** These are identified and delivered by the capture module. For instance, libpcap (http://tcpdump.org) is an open source tool that captures IP packets from Ethernet wires and presents them to requesting programs, and can be used for live network monitoring.

**Timer Events** Another kind of primitive event is the tick generated by a system clock. Many protocols have specifications and requirements that are time-bound. This makes it necessary for NER to keep track of the system clock to recognize events such as timeouts that are often necessary to signify the absence of an event.

Then, we need to put these primitive events together to reconstruct meta events for the next higher layer. Note that the receiver of the packets also needs to carry out this reconstruction - the packets it receives are passed up the protocol stack and events are reconstructed layer by layer. So the event recognizer simply has to mimic the receiver’s state machine in order to reconstruct the meta events as they would have been seen at the receiver. If the sender and the receiver are operating correctly, then these reconstructed events are exactly those that were produced by the sender and accepted by the receiver.

Additionally, to check whether the sender and receiver are working correctly, we check whether
the observed sequence of messages is disallowed by the standard or the requirements. For in-
stance, if we encode the sender and receiver state machines as specified by the standard, a con-
formance violation would be indicated by a message being sent in a state that does not allow it.
A functionality violation would either be expressed as a safety property of the reconstructed state - `state S is never reached` - or a timed property of messages - `message A is always followed
by message B within T seconds`. When conformance or functionality violations are detected, the
event recognizer produces alarms or error events.

One could think of error event recognition and meta event reconstruction as independent ac-
tivities, but there are good reasons for combining them in a single recognizer. The state machine
descriptions for the two tasks are often similar and can be shared. Moreover, we like to think of
an event recognizer as a monitorable version of the protocol specification, which is in itself a com-
bination of both implementation description (this is how you transfer data) and interoperability
restrictions (these are the only messages you can send). As a result, it is natural to write the event
recognizer as a combination of data reconstruction and protocol checking.

For meta events and error events, there are four main recognition techniques. Some of these
have been studied before [LF98] in the context of event management systems.

Filtering  Events have attributes. Filtering recognizes events whose attributes satisfy a predicate.
For instance, recognize all email-related packets, and throw away the rest.

Parsing  Parsing event attributes uncovers hidden structure within its fields. For instance, the TCP
segment event takes an IP packet event and extracts TCP header fields from its payload.

Correlation  Recognizing that events E1 and E2 have both occurred. For instance, recognizing
that a TCP packet contains data D as well as ack A.

Aggregation  Recognizing that a sequence (or pattern) of events has occurred. For instance, a
typical SMTP session involves naming the sender S, naming the recipient R, and sending
the email E. These three events can be aggregated into an EmailSent event containing all the
information sent.

Abstraction  Each layer of event recognizers presents only an abstraction of the events it monitors
to the layer above. For instance, if a recognizer reconstructs the data seen in a TCP session,
the higher layer only sees the abstract TCP_Data events and not the entire sequence of
packets that caused this event to be produced.

The next task is to develop programming techniques for writing event recognizers using the
above techniques. Later in this chapter, we define the syntax and semantics for a domain specific
language, NERL, for writing event recognizers and a modules language, NERL_MOD, for putting
recognizers together to form a monitoring stack.

3.2 Monitoring Environments

In the context of monitoring, the term **fidelity** refers to the closeness with which the sequence of
input and output events seen by the device under test matches the sequence of events observed
by the monitor. A perfect fidelity protocol event recognizer is one that sees exactly the inputs and
outputs of the protocol implementation it monitors. In Figure 3.4, this means that the PSEEN is the
same as the PSENT and the PRECV, and it occurs at exactly the same time (Figure 3.4,1). Fidelity is
important because correct event recognition will be possible only if the captured trace is correct.

Fidelity in the monitoring environment depends on the quality and number of channels be-
tween the implementation under test and the monitor. For instance, if a routing protocol runs on
top of IP, we say that the participants send messages to each other on the IP channel, which pro-
vides very few guarantees about the delivery of the messages. On the other hand, an application
Figure 3.4: Monitoring Infidelities
protocol such as SMTP runs on top of TCP. So we say that processes running SMTP send each other messages on a TCP channel, which guarantees reliable, in-order delivery. The channel is an abstract entity provided to a protocol by the underlying network and lower layer protocols.

Since network event recognition is meant to work on IP packet traces, we first examine the behavior of the IP channel. Then we discuss three monitoring configurations in which we will carry out our passive monitoring analysis.

**IP Channel Characteristics**

In a realistic network, infidelities are introduced at the IP layer because of network errors and router buffering.

**Delay** Packets sent are delayed before or after passing the monitor. PSENT at time $T$ triggers PSEEN at $T'$, which triggers PRECV at $T''$ (Figure 3.4, II).

**Loss** Packets sent are dropped by the network before or after passing the monitor. PSENT never becomes PSEEN, or PSEEN never becomes PRECV (Figure 3.4, III).

**Reordering** Packets sent in one order reach the monitor or receiver in a different order. PSENT($p$) followed by PSENT($q$) results in PSEEN($q$), PSEEN($p$) or PRECV($q$), PRECV($p$) (Figure 3.4, IV).

**Duplication** One packets sent gets converted to two packets before reaching the monitor or receiver. PSENT($p$) results in PSEEN($p$), PSEEN($p$) or PRECV($p$), PRECV($p$).

**Corruption** Packets are modified by the network. PSENT($p$) results in PSEEN($p'$), or PRECV($p'$).

Delay is caused by buffering on the intermediate routers, while packet loss occurs when these buffers spill over. Reordering is caused by the different paths that different packet may take over the internetwork. In rare cases, packets get corrupted and duplicated when links such as wireless networks malfunction.

At the packet layer, the extent of infidelity is determined by the performance of the monitor and its placement in the network. If the packet capture software is too slow to sniff packets off the wire under heavy load, it will miss packets. We build our event recognizers on top of libpcap ([http://tcpdump.org](http://tcpdump.org)), an open-source library that is under active development. We have found that libpcap can easily keep up with an inter-packet arrival time of about 20 microseconds. At these speeds, it is feasible to monitor typical TCP/IP protocol traffic over 100 Mbps Ethernet, without the monitor dropping any packets. Above this library, our event recognizers must also be fast enough and consume the data as it is captured off the wire. We will show that this is feasible for the protocols that we analyze. Network placement however, is critical to monitoring. In Figure 3.5, monitor $M_1$ is co-located with the device under test (DUT); it will enjoy perfect fidelity with respect to the DUT. Monitor $M_2$ is co-networked and monitor $M_4$ is at a bottleneck location; these are particularly useful, as they are able to observe all traffic between the device and the remote host. Monitor $M_3$ will not observe traffic passing through network element $A$. $M_5$ is located at an Internet service provider (ISP), and $M_6$ is located in another service provider’s network. The further the monitor is from the DUT, the more infidelities it will experience.

In the presence of significant infidelities, event recognition quickly becomes infeasible. This is because the protocol event recognizer cannot distinguish between protocol errors and trace infidelities. For instance, if the protocol specification says that message B can only be sent after sending message A, and the monitor sees message B without a preceding message A, it is difficult to tell whether this indicates an implementation error, or whether A was dropped before reaching the monitor, or whether A was never seen because the monitor was not at a bottleneck location.

Higher-layer recognizers may be protected from some of these infidelities by the level of abstraction at which they operate. For instance, IP and UDP employ checksums that protect against
packet corruption, so a recognizer for a protocol that runs above these layers can assume that packet corruption is detected and dealt with at the lower layer recognizer. Similarly, we shall show that recognizers for protocols that run on TCP can ignore loss, re-ordering and duplication. But in general, event recognizers may produce incorrect results in the presence of infidelities.

Trace infidelities have been discussed earlier, in the context of both TCP monitoring [Pax97] and intrusion detection [PN98], and have been known to cause false positives - alarms when none should be triggered. If we want our event recognizers to be correct, they must be able to distinguish between infidelities and implementation errors. We consider three monitoring environments in which event recognition is feasible: co-located monitoring (M1), bottleneck monitoring when the protocol is at a high layer of abstraction (M4), and co-networked monitoring for low layer protocols (M2).

Co-located Monitoring

We define a co-located monitor as one that can observe the input and output actions of the device under test synchronously with the device - it encounters no infidelity whatsoever. This kind of monitor can only be implemented by inserting it into the protocol stack on the device under test. Indeed this intrusive approach is followed by some test systems, such as Orchestra [DJM96]. Such monitors are hard to deploy because of the need to put new kernel-level software on every monitored system.

An alternative strategy is to execute the protocol implementation in a network simulation environment. In a network simulator, all devices, links, and protocol stacks are simulated on a single machine, and every simulation produces a single execution trace for the complete network.

In effect, the simulator monitoring channel provides a co-located view at every simulated node (a god’s eye view), so the resulting simulation trace has perfect fidelity - every event is logged exactly when it happens. Although packets may be lost even in the simulated network, the monitor has a perfect view of the packets that left and arrived at each node. In Chapter 8, we will describe
network event recognition for a routing protocol in a simulation environment.

Bottleneck Monitoring over Reliable Streams

A bottleneck monitor is defined as one that can observe all the inputs and output produced by the device under test, but the monitoring channel is subject to delay, loss, re-ordering and duplication. To effectively combat these infidelities, we will additionally require that the monitoring channel be implemented by a protocol that provides reliable message streams.

A reliable in-order transport protocol, such as TCP, guards against data loss, duplication and re-ordering - it implements a reliable, first-in-first-out, duplex channel between two participants. So a protocol that runs above TCP can ignore lower-layer infidelities. Similarly the event recognizer for a protocol above TCP can assume perfect fidelity, up to some delay, as long as the TCP packets are correctly reconstructed.

To see why recognizers above TCP can ignore infidelities, assume that all layers up to TCP are working correctly at the sender and receiver. Then, reliability implies that there is a time \( T \) in a TCP session when both the sender and receiver agree that data segment \( S \) has been successfully sent and received. So by mimicking the operation of both the sender and the receiver, the event recognizer can conclude at time \( T \) that \( S \) was sent and \( S \) was received at some time before \( T \). This information is adequate to accurately reconstruct the events of layers above the reliability layer, as long as some delay is permitted. A concrete example of this kind of analysis is shown in Chapter 9.

Co-networked Monitoring

A monitor is said to be co-networked with the device under test if they sit on the same physical broadcast network (LAN). In contrast, a bottleneck network sits just outside the gateway of the LAN. Of the two, a co-networked monitor can clearly enjoy better fidelity, as there is no network element (router) between it and the device. Because of the broadcast nature of the LAN, the monitor sees data at exactly the same time as the device does, and current link layer technology minimizes loss, delay and re-ordering. In a real (not simulated) network, or even in a network test-bed, a co-networked monitoring configuration is the best we can hope for. It is important that network event recognition be applicable for such a configuration. Moreover, we hope that some of the techniques we develop for co-networked monitoring can extended to general bottleneck monitoring.

Note that although the infidelities in co-networked (or bottleneck) monitoring do not matter if the protocol under analysis runs over a reliable message stream, these infidelities become significant during event recognition over unreliable channels. So, although a monitor for a protocol that runs over TCP may ignore channel infidelities, a monitor for TCP itself cannot. For instance, if we wish to check that the reliability component of a TCP implementation is correct, then we must take loss and re-ordering into account; indeed these are the errors that TCP is supposed to smooth over.

The primary fidelity challenges in co-networked monitoring arise in dealing with buffering on the device under test. If a protocol \( S \) is being run by the device, then the goal of the monitor \( M \) is to determine if \( S \) is properly implemented. However, this must be done by observing behavior on the network, and there may be input and output buffers between the device and the network. The situation is depicted in Figure 3.6. For instance, the Linux ethernet drivers capture packets from the wire and hand them over to a routine \( \text{netif_rx()} \) that queues the packet in a buffer for later processing. When the CPU next has processing cycles, it processes this packet and passes it up through the protocol stack if necessary. Networks can often produce data too fast for the CPU to handle; this leads to input buffering. In rare cases, the local network is congested, so output packets are also buffered.
When input and output buffers over-fill, packets are silently dropped at the device, leading to monitor infidelities. Note that this over-fill may be caused by packets from some other protocol that the device is engaged in; the monitor may never even look at these packets. But even without input buffer loss, input buffering can cause different perceptions of packet arrival and dispatch at the monitor \( M \) and device \( D \). We describe this phenomena in some detail in the next chapter. To see the issue briefly, note that it is impossible for \( M \) to tell in Figure 3.6 whether output \( d \) of \( S \) was created by the device before or after the device observed \( a \) or \( b \), or even whether \( b \) was dropped before it reached \( S \).

Our experiments showed that while input buffering is common, buffer overfill and input packet loss occurs only rarely and under heavy network load on the device under test. Output buffering and loss are also quite rare and occur only in heavily loaded networks. Moreover, the kernel will ordinarily throttle the protocol implementation to avoid producing outputs that may get delayed or lost in the output buffer. In this thesis, we assume that the monitor is fast enough to keep up with the input-output traffic, and that outputs never get buffered or lost.

For our purposes, we define a \textit{co-networked} monitor as one that can observe all outputs produced by the device under test without delay, and can observe all inputs going toward the device, but these inputs may be buffered and lost at the device under test. Our approach for co-networked monitoring is to reconstruct the possible behaviors of the monitoring channel by modeling the non-deterministic input buffer. In Chapter 7, we describe an algorithm that reconstructs events on the co-networked monitoring channel. Then in Chapter 10, we use this algorithm to execute a co-networked recognizer for TCP and check the correctness of several TCP implementations.
3.3 Strategies for Co-networked Monitoring

When a monitor can accurately see what messages the device under test is sending and receiving, it is quite simple to write a network event recognizer to check that the device is operating correctly. In Chapter 6, we described how NERL and NERLMOD can be used to program such a recognizer for the ping protocol. However, we also observed that this kind of perfect fidelity between the message stream observed by the monitor and the message stream observed by the device is difficult to achieve. For instance, if we fail to place the monitor in a bottleneck location, it will miss some packets that take an alternative path.

But even the packet stream seen by a bottleneck monitor can contain infidelities: packets may be delayed and lost on the way from the device to the monitor, or from the monitor to the device. In this section, we investigate strategies for combating such infidelities for a special case of the bottleneck monitor - the co-networked monitor. These strategies are a first step toward understanding the effectiveness of passive monitors in a general bottleneck monitoring environment.

A co-networked monitor is one that sits on the same broadcast network (LAN) as the device. So it sees every link layer “frame” as it is generated and received by the network interface card (NIC) on the device under test. The only source of infidelity in this environment is that there is a packet buffer between the NIC and the protocol stack. When a packet arrives at the NIC it is not immediately consumed; instead it is stored in the packet buffer until the device has enough resources to process it.

In a moderately loaded network, the packet buffer will often contain two or three packets as the device processes them one by one. But even in a lightly-loaded network, if two packets from different sources arrive too close together, one of them will be buffered. Under heavy load - when the device gets a large burst of packets - the packet buffer may get filled, leading to new packets getting dropped. This happens rarely - typical packet buffers can store more than a hundred packets - but we have observed one or two occurrences in some long streams.

To fix terminology, we shall refer to the packets received by a device as its inputs, and we shall refer to the packet buffer as the input buffer. When an input get processed by the protocol stack, we say that it has been consumed. If a new input gets dropped by the packet buffer (because of overflow), we say that the input has been lost.

A network event recognizer attempts to reconstruct the current state of a protocol at the device under test and raises error events if the message trace it observes violates the protocol specification. But an event recognizer that does not account for input buffering and loss will have an incorrect view of protocol state, which may lead to superfluous error events (“false positives”). Effectively there is an additional layer in the protocol stack; the new bottom layer is a non-deterministic, first-in-first-out (FIFO), lossy buffer that feeds packets to the IP layer. A correct event recognizer must model the behavior of this buffer and distinguish actual errors in the protocol from behaviours that are caused by the input buffer.

In the rest of this section, we describe issues in programming such network event recognizers. In the next section, we show how to annotate NERLMOD programs with information about buffers at the device under test. Then we formalize the effect of these buffers on the monitoring task. Finally, we present a general algorithm that transforms a recognizer that assumes perfect fidelity to one that accounts for input buffers.

Output buffering and Bottleneck monitoring In general, even output packets generated by the device may get buffered and lost under heavy network load, but we do not detect any such occurrence in our co-networked monitoring experiments. For one, the network is usually much faster than the device and it is unlikely that the device will produce packets faster than the network can consume them. Second, if the network is heavily loaded and cannot accept packets, this information is usually fed back to the kernel which then throttles the protocol stack to slow down output.
To relate the strategies we develop for co-networked monitoring to general bottleneck monitoring, note that an internetwork behaves much like a non-deterministic, non-FIFO, lossy buffer. So an effective bottleneck monitor must account for input and output buffers that additionally allow packet re-ordering. It is reasonable to assume that strategies for programming recognizers for bottleneck monitoring will build on our strategies for co-networked monitoring. Several of the features of our co-networked recognizer.

### 3.3.1 Co-networked Monitor Model

In this section, we present a model for the sequences observed by a co-networked monitor. This model was earlier published in [BCMG01]. We first describe input buffering, and in the next section we extend the model to account for input loss.

Assume that we want to monitor a protocol with one input channel and one output channel (like the ping signature above). At each step, either it consumes an input, or produces an output. Outputs may be produced in reaction to inputs, or in reaction to events such as timer expirations, that are not visible to the outside. The event recognizer for this protocol must check that the implementation follows the specification $S$. Our model of the protocol is similar to a deterministic I/O Automaton [LT89]; it is a machine with a (possibly infinite) state space carrying out parameterized input and output actions.

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

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

We introduce three new tokens. $iq_a$ corresponds to enqueing of an input symbol $a$ in queue $I$. $id_a$ corresponds to deletion of $a$ from head of $I$ and simultaneous input into device. $od_a$ corresponds to output of $a$ from device. Note that $M$ gets to see only tokens $iq$ and $od$. We also assume that observation of $iq_a$ at the monitor is simultaneous with enqueing of $a$ into $I$ and observation of $od_a$ at the monitor is simultaneous with its output from the device.

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

$$iq_a iq_b iq_c id_a id_b od_d id_c$$

This sequence represents the following events: $a$, $b$, and $c$ got enqueued in the input queue $I$; the device consumed $a$ from the head of the queue; the device consumed $b$ from the head of the queue and then produced output $d$; and, finally, the device input $c$. The monitor sees the sequence

$$iq_a iq_b iq_c od_d$$

and we will define the language recognized by the monitor in terms such sequences. We first introduce a notion of admissible execution sequences under the buffering restrictions. The co-networked recognizer for $S$ accepts all admissible executions that are allowed by $S$.

**Definition 3.1 (Admissibility)** $\omega$, a string of $iq$, $id$ and $od$ tokens, is said to be an admissible execution sequence with respect to $M(S, m)$, if the following are true:

- **FIFO Input**: if $iq_a$ occurs before $iq_b$, then $id_a$ occurs before $id_b$.
- **Causality**: the $k$'th $id$ token can only occur after the $k$'th $iq$ token, and
- **Input Buffer Limit**: in any prefix of $\omega$, the number of $iq$ tokens minus the number of $id$ tokens must not exceed $m$.
According to the admissibility model, there are three sequences that a co-networked recognizer needs to be aware of

- $\tau$: the $iq/od$ sequence observed by the monitor,
- $\sigma$: the $id/od$ sequence that actually occurred at the device, and
- $\omega$: the $iq/id/od$ sequence that represents the execution of the device-monitor system and contains both $\tau$ and $\sigma$ as projections.

Since the monitor can only see $\tau$, it must reconstruct the system execution $\omega$ and extract the device execution sequence $\sigma$. Admissibility defines the set of possible system execution sequences ($\omega$) that are consistent with the observed trace and input buffering parameters. Now we have a way of defining the language that must be recognized by $M$. Essentially, a string $\tau$ belongs to $L_M$, if there is some admissible sequence whose $iq/od$ projection is $\tau$, and whose $id/od$ projection belongs to $L_S$. Formally:

**Definition 3.2** The co-networked monitor language accepted by $M(S, m)$ is defined as $L_M = \{ \tau \mid \exists \omega: \omega_{iq, od} = \tau \land \omega_{id, od} \in L_S \land \omega \text{ admissible w.r.t. } M(S, m) \}$.

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

**Example.** Consider the following string in $S$:

\[
i_a \ i_b \ o_c \ i_d \ o_e \ i_f \ i_g
\]

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

\[
A : iq^1 id^1 iq^2 id^2 od^1 iq^3 id^3 od^2 iq^4 id^4 iq^5 id^5
\]

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

\[
B : iq^1 iq^2 id^1 id^2 od^1 iq^3 id^3 od^2 iq^4 id^4 iq^5 id^5
\]

\[
C : iq^1 iq^2 iq^3 id^1 id^2 od^1 id^3 od^2 iq^4 id^4 iq^5 id^5
\]

\[
D : iq^1 iq^2 iq^3 id^1 id^2 iq^4 od^1 iq^5 id^3 od^2 id^4 id^5
\]

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

\[
E : iq^1 iq^2 iq^3 iq^4 id^1 id^2 od^1 iq^5 id^3 od^2 id^4 id^5
\]

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

\[
A, B : iq^1 iq^2 od^1 iq^3 od^2 iq^4 iq^5
\]
The maximum number of paths in an above the major diagonal (0, 0) to (n, n) without crossing the major diagonal (x = y) because then the number of id tokens will exceed the number of iq tokens. The maximum number of paths in an n x n grid from (0, 0) to (n, n) without crossing the major diagonal is the maximum number of paths in an n x n grid from (0, 0) to (n, n) without crossing the major diagonal.

Lemma 3.5 If \( \omega \) is admissible w.r.t \( L(S, m) \), then it is admissible w.r.t \( L(S, m') \) where \( m' > m \).

Proof. This lemma follows directly from the clauses of the definition for admissibility. The only clause affected is the input buffer limit which has been increased.

Finally, since \( L_M \) depends on the set of \( \omega \) that are admissible with respect to \( M(S, m) \), we are interested in the maximum size of this set.

Theorem 3.6 Suppose that the monitor observes a trace \( \tau \), and the input buffer has size \( m \). Then the total number of \( \omega \) that can be constructed from \( \tau \) (\( \omega_{id, od} = \tau \)) such that they are also admissible with respect to \( M(S, m) \) is less than \( C(N_{iq}) \times (m + 1)^{N_{od}} \), where \( N_{iq} \) is the number of inputs in \( \tau \), \( N_{od} \) is the number of outputs in \( \tau \), and \( C(n) \) is the \( n \)th Catalan number:

\[
\binom{2n}{n} \quad \frac{n + 1}{n+1}
\]

Proof. Note that every \( \omega \) that is admissible w.r.t \( M(S, m) \) is constructed by inserting id tokens between the iq and od tokens. First, each \( \omega \) must follow the FIFO clause - the number of iq tokens is the same as the number of id tokens. This number is defined as \( N_{iq} \) in the theorem. Then note that \( \omega \) must also follow the causality clause - so the number of id tokens can never exceed the number of iq tokens in any prefix of \( \omega \).

So one can view the inputs in \( \omega \) as a path in a two-dimensional grid from (0, 0) to \( (N_{iq}, N_{iq}) \), where the x-dimension refers to iq tokens and the y-dimension refers to id tokens. This path must never go above the major diagonal (x = y) because then the number of id tokens will exceed the number of iq tokens.

The maximum number of paths in an n x n grid from (0, 0) to (n, n) without crossing the major diagonal is the maximum number of paths in an n x n grid from (0, 0) to (n, n) without crossing the major diagonal.
diagonal is known to be given by the \( n^{th} \) Catalan number. So the number of input sequences themselves are bound by \( C(N_{iq}) \).

Then we must place the \( od \) tokens between the inserted \( id \) tokens. Since the number of \( id \) tokens in a row cannot exceed the size of the buffer (each \( id \) represents a buffer delete), so each \( od \) may have to be placed between \( m \) \( id \) tokens, which adds an additional factor of \( (m + 1)^{N_{od}} \).

In the above theorem we ignored the input buffer limit clause, which implies that the paths that are allowed from \((0, 0)\) to \((N_{iq}, N_{iq})\) must stay in the band defined by \( x \geq y \) and \( x \leq y + m \). By ignoring the upper limit of \( x \), our upper limit was a little loose. But even if \( m = 1 \), the number of such paths is \( 2^{N_{iq}^2} \). So even with an input buffer of size 1, the number of \( \omega \) the co-networked monitor needs to reconstruct may be as great as \( (2^{N_{iq}^2} + 2^{N_{od}}) \) or \( O(2^{|t|}) \).

### 3.3.2 Dealing with Loss

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

**Example.** Consider the following sequence:

\[
\text{iq}_a \ il_a \ \text{iq}_b \ \text{id}_b \ \text{od}_d \ il_c
\]

In this sequence, \( S \) observes (produces) the following string:

\[
i_b \ \text{od}_d
\]

Whereas, the monitor observes the following tokens:

\[
iq_a \ il_b \ il_c \ od_d
\]

Inputs \( a \) and \( c \) are lost in the loss unit.

We assume that loss bursts are bound: suppose there at most \( l \) input tokens can be lost in succession. Then the co-networked model is \( M(S, m, l) \) with input buffering limit \( m \) and input loss limit \( l \). We can then extend the definition of admissibility as follows:

**Definition 3.7 (Admissibility)** \( \omega \), a string of \( iq \), \( id \), \( il \) and \( od \) tokens, is said to be an admissible execution sequence with respect to \( M(S, m, l) \), if the following are true:

**FIFO Input:** the sequence of \( id/il \) tokens in \( \omega \) is a prefix of the sequence of \( iq \) tokens,

**Causality:** the \( k \)’th \( id/il \) token can only occur after the \( k \)’th \( iq \) token,

**Input Buffer Limit:** in any prefix of \( \omega \), the number of \( iq \) tokens minus the number of \( id/il \) tokens must not exceed \( m \), and

**Input Loss Limit** in any prefix of \( \omega \), the number of consecutive \( il \)’s without an intervening \( id \) token must not exceed \( l \).

The definition of \( L_M \) also changes accordingly:

**Definition 3.8** The co-networked monitor language accepted by \( M(S, m, l) \) is defined as \( L_M = \{ \tau \mid \exists \omega : \omega_{iq, od} = \tau \land \omega_{id, od} \in L_S \land \omega \text{ admissible w.r.t } M(S, m, l) \} \).
Example. Consider the previous string in S. In addition to $A - D$, we can arrive at several other admissible sequences with respect to $M(S, 3, 2)$ ($l = 2$), by accepting inputs that might get lost:

$$F : iq^0 il^0 iq^1 id^1 iq^2 id^2 od^1 iq^3 id^3 od^2 iq^4 id^4 iq^5 id^5$$
$$G : iq^0 iq^1 iq^2 il^0 id^1 id^2 od^1 iq^3 id^3 od^2 iq^4 id^4 iq^5 id^5$$

However, not just any number of lost inputs can be introduced. The following string is not admissible because the number of elements lost cannot exceed 2.

$$H : iq^{-1} iq^0 iq^1 iq^2 iq^3 id^2 il^{-1} il^0 id^1 id^2 od^1 iq^5 id^3 od^2 id^4$$

As before, any observed trace that does not belong to $L_M$ indicates an error in the device under test:

**Theorem 3.9** Suppose a device with an input buffer of size $m$ and a loss module that can lose up to $l$ packets in a row produces an execution sequence $\sigma$, and the trace observed at the co-networked monitor is $\tau$. If $\sigma \in L_S$, then $\tau \in L_M$.

The proof of this theorem follows immediately from the definition of admissibility as before.

By adding the loss module, we increase the number of device execution sequences that the monitor cannot distinguish between, approximating the properties even further:

**Definition 3.10** Device execution sequences $\sigma_1$ and $\sigma_2$ are said to be buffer-loss-equivalent with respect to an input buffer of size $m$, loss module with loss-burst $l$, and observed trace $\tau (\sigma_1 \sim_{\tau,m,l} \sigma_2)$ if $\{ \omega \mid \omega_{id,od} = \sigma_1 \land \omega_{iq,od} = \tau \land \omega \text{ admissible w.r.t. } M(S,m,l) \} = \{ \omega \mid \omega_{id,od} = \sigma_2 \land \omega_{iq,od} = \tau \land \omega \text{ admissible w.r.t. } M(S,m,l) \}$.

The safety of loss over-estimation if formalized by the following lemma:

**Lemma 3.11** If $\omega$ is admissible w.r.t $L(S,m,l)$, then it is admissible w.r.t $L(S,m',l')$ where $m' \geq m$ and $l' \geq l$.

**Proof.** This lemma again follows directly form the clauses of the definition for admissibility. The only new clause to check is the input loss limit.

The loss module increases the overall non-determinism in the system. As a result, the monitor needs to keep track of even more admissible $\omega$ than before.

**Theorem 3.12** Suppose that the monitor observes a trace $\tau$, the input buffer has size $m$, and the loss module has a limit $l$. Then the total number of $\omega$ that can be constructed from $\tau (\omega_{iq,od} = \tau)$ such that they are also admissible with respect to $M(S,m,l)$ is less than $2^{N_{iq}} \ast C(N_{iq}) \ast (m + 1)^{N_{od}}$, where $N_{iq}$ is the number of inputs in $\tau$, $N_{od}$ is the number of outputs in $\tau$, and $C(n)$ is the $n$th Catalan number:

$$C(n) = \frac{2n \choose n}{n + 1}$$

**Proof.** Since now we have a loss module, only a subset of the $iq$ tokens translate into $id$ tokens, the rest become $ils$. Ignoring the input loss limit, the maximum number of such subsets is given by $2^{N_{iq}}$.

After choosing a subset, we need to construct $\omega$ exactly as it there were only an input buffer but treating $il$ and $id$ tokens as identical. This means that the remaining factors in the bound remain the same.

The bound expressed above is again somewhat loose, because we ignore the input loss limit. But even if the loss limit were 1, one in every two packets may be lost. So the number of subsets is bound by $2^{N_{iq}/2}$. So even with a small buffer and loss limit, the number of $\omega$ that the monitor needs to keep track may still grow to $O(2^{r/2})$. Since this is already too large, the precise number of possible $\omega$ sequences is not that relevant. Hereafter we shall only refer to an “exponential” sized admissible set.
3.4 Motivations for a new Monitoring Language

3.5 Tuning

The main aim of a NERL analysis is to be able to point out flaws in the protocol implementation. However, when there are close to a million error events, it is extremely difficult to understand why these errors took place. We can use techniques such as output event filtering to reduce the number of output events reported; it does not provide any insight into their cause. Event tracing indicates the packet events that caused the error, which is very useful, but does not by itself pinpoint the possible bugs in the implementation under test.

One approach to protocol bug-hunting is the naive “repair first bug” approach. This assumes that the protocol implementation code is available, for instance if it is a simulator implementation. We first run the implementation followed by the NERL analysis. We look at the output event trace and try to find the bug in the code that caused the first output event. We repair this bug, and then re-run the implementation and NERL analyses, and repeat. This is clearly inapplicable to live network monitoring. Even in the case of a network simulation, each re-run may take a few hours to complete. Clearly, it would be more efficient to attempt to find the most bugs in the same simulation or packet trace. We define a technique, called tuning, that attempts to do this.

A NERL recognizer specifies a reference state machine that the implementation is supposed to follow. If there is an output error event, then it is likely that the implementation has performed an incorrect action. By looking at the output event, we can try to guess what the incorrect action was. We can then modify our reference state machine written in NERL to also perform this incorrect action. We call this modification tuning. The tuned NERL recognizer mimics the implementation. We then re-run the NERL analysis with the tuned recognizer. If the number of errors is reduced significantly, our guess was probably right. Moreover, if the guess was right, then the new output event trace contains independent errors, not tainted by the incorrect action that we have masked by tuning.

The intuition for the name, tuning, is that we consider error events as noise, cause because of the inconsistency between the NERL recognizer and the implementation. By tuning, we attempt to bring the NERL state machine closer to that of the implementation, to test our guess of where the error is, as well as to find new errors in the implementation.

3.6 Fault Origin Adjudication

NERL can be used to express high-level protocol requirements $R$, and then analyze protocol event traces and identify failures of these requirements, indicating faults in the implementation. This could be because the implementation $P$ deviated from the standard $S$. But it is also possible that the specification $S$ was incorrect because it failed to ensure the high-level user requirements $R$. In this section, we introduce a technique to determine which of these two possibilities obtains, assuming that high-level requirements have been properly expressed and a deviation from them has been found. We call this process Fault Origin Adjudication (FOA) [BGO00].

Suppose that we first check a protocol event trace $T$ produced by $P$ against a NERL program representing the requirements $R$. We call such a program the functionality checker. Then we check the same trace against a NERL program representing the specification $S$. We call this program the conformance checker. A combination of answers from the conformance checker and the requirements checker enables us to reason about the fault origin. Table 3.1 describes the four possible outcomes and their interpretations.

Notice the error case A; it indicates a design error in the standard, even though we never analyzed the standard directly. FOA therefore can potentially point out a design error for free, in addition to the implementation errors that NERL points out anyway.
Table 3.1: FOA outcomes and their interpretations

<table>
<thead>
<tr>
<th>Region</th>
<th>Interpretation and remedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Everything OK.</td>
</tr>
<tr>
<td>D</td>
<td>Incorrect implementation of the standard. Correct the program.</td>
</tr>
<tr>
<td>C</td>
<td>Incorrect implementation of the standard. Correct the program.</td>
</tr>
<tr>
<td>A</td>
<td>Incorrect standard. Revise the standard and the program accordingly.</td>
</tr>
</tbody>
</table>

We shall demonstrate the use of FOA in the AODV case study (Chapter 8). In the case of AODV, we were able to produce traces that fell in each of the categories A, C and D (most traces fell under D). The A trace indicated that the AODV standard itself was faulty, and allowed loop formation in some cases. This error in the standard had been found earlier by us as part of a protocol verification project [BOG02].
Chapter 4

Passive Protocol Monitoring

4.1 Related Work

In this section, we survey tools and techniques that share our goals, methodology or case studies. We compare these research works to ours and argue that none of them solve the problem we have set out to address. First we look at existing studies that have tried to analyze the kinds of protocols we have chosen as our case studies. Then we look at passive protocol testing techniques that are closely related to our methodology. We then survey work in run-time monitoring of software that uses similar methodology in a different domain. Finally we look at relevant work in specification-based network intrusion detection where the methodology is related to ours but goals and case studies differ.

4.1.1 The Case Studies

TCP. For important protocols such as TCP, the IETF has documented the various testing tools that represent the collective wisdom of implementors and network administrators [PS98]. For instance, a particularly sophisticated approach is taken by the Orchestra tool [DJM97, DJM96] that uses fault injection to test a TCP implementation under different network error conditions. Orchestra has been used to analyze six commercial TCP implementations and find various errors in them. Orchestra showcases the advantages of the active testing approach - it uses a high degree of control over the test environment to find errors. On the other hand, this approach is too intrusive - it needs a new layer between TCP and the network layer on every machine - and so is unusable after deployment in an operational network, or in a test network with a large number of nodes.

A passive testing approach is taken by Paxson’s tcpanaly [Pax97]: a tool that analyzes captured network traces of packets going to and coming from a TCP implementation, and attempts to find errors in the implementation. However, the tcpanaly tool does not completely succeed in this attempt because of limitations in the packet capture and protocol analysis technology available at the time. Paxson cites packet capture errors: drops (tcpanaly does not see packets that the device under test does), additions (tcpanaly sees extra packets which never make it to the device under test), and re-sequencing (tcpanaly sees packets in an order different from the device under test). In addition, Paxson argues that different TCP implementations differ from each other to such an extent that a generic TCP correctness specification is not feasible. So instead tcpanaly is hard-programmed with intimate knowledge of the expected TCP behavior for each operating system. This compromise is somewhat successful and many of the errors found by tcpanaly have been documented by the IETF as typical TCP implementation problems [PAD+99]. In this thesis, we aim to overcome the limitations that Paxson cites and provide solutions for some of them. Our aim is to carry out generic TCP monitoring without looking inside individual implementations.
Routing. Testing routing protocols is harder because they involve many participants distributed over an unpredictable network topology. Since routers are an important component of the network infrastructure, any active testing must be done before deployment by constructing realistic test network configurations [HLSV00]. However, since predicting realistic Internet topologies is difficult and since Internet traffic is not very well understood, such testing is not adequate and passive monitoring of routers after deployment has become a popular technique. Malan and Jahanian [MJ98] describe windmill: a passive monitoring architecture that has been used for measuring routing protocol performance. Protocol modules for several layers are written in C and the windmill architecture describes how they extract information from each other. Monitors are then distributed through the network where they capture and analyze routing protocol messages. The goal of windmill is primarily performance measurement, and the architecture makes no attempt to guarantee monitor correctness or to provide error diagnostics. On the other hand, windmill is geared toward highly efficient execution. Other passive monitoring tools, such as RouteMonitor [MF99] or PITS [ZWY02], are customized for specific routing protocols and do not provide generic support for new protocols. In contrast, we aim to create a flexible, general system that can be applied to even new protocols, such as AODV, with support for ensuring monitor correctness and providing error diagnostics.

An alternative to a distributed passive monitoring architecture is to use network simulations of routing protocols. Simulations are cheap to set up and provide a complete ‘god’s eye’ view of the network. Several network configurations can be tried quickly and the hassle of maintaining distributed monitors eliminated. Simulators, such as NS [Pro], take a protocol implementation and execute it over a virtual network configuration producing an event trace. However, since simulator implementations are carried out primarily for prototyping, there have not been many studies to check the correctness of these implementations. Helmy and Estrin describe a method called Systematic Testing of Robustness by Examination of Selected Scenarios (STRESS) [HE98, HEG98] that enhances NS simulations with support for specification based test scenario generation and output trace analysis. They show how to carry out STRESS for a multicast routing protocol. The user specifies the routing protocol using FSM descriptions which are then used to generate scenarios and test sequences that inject faults into the simulation and check the protocol implementation for robustness. The aim of STRESS is to actively test the robustness of the protocol specification and not to passively check the correctness of the implementation. In contrast, our work analyzes routing protocol implementation correctness through simulation trace analysis.

SMTP. Implementations of application protocols, such as SMTP, are usually tested by commercial software companies or by users in the field using ad-hoc active testing techniques. They are not subject to much research. However, bugs in SMTP implementations are considered serious security risks and are frequently posted on security websites such as SANS [SAN]. Surveillance tools, such as Carnivore [SCHP*00], passively monitor SMTP implementations but do not attempt to analyze their behavior. Network intrusion detection systems also sometimes have SMTP modules to detect attacks on the mail server, but these do not check the correctness of the server.

4.1.2 Passive Testing

While the majority of protocol testing research is confined to active testing, passive testing is preferred for detecting errors that may make themselves felt only over a long period of time, or only in unexpected network environments. Active testing can be thought of as a combination of test case generation and test result analysis. However, it would be incorrect to assume that we can extract a passive tester from every active testing technique. This is because each active test sequence is written in a notation such as TTCN [TTC98] which contains inputs to send to the IUT followed by the output expected. In contrast, a passive tester does not know the inputs that will be sent and consequently the outputs that should be produced, it must be ready to analyze any sequence of inputs and outputs. So designing passive testers is not a trivial task.
Lee et al [LNS+97] define passive testing as observing the input/output behavior of a system in its natural operation for the purpose of detecting faults. The expected behavior of the network is represented using FSMs and they describe algorithms to check if a captured input/output trace violates the FSM specification. This method has been used to check Internet routing protocols [ZWY02] as well as SS7 routing protocols [LNS+97]. Tabourier, Cavalli and Ionescu show how to extend the algorithms so that they can work for protocol specifications written as Extended Finite State Machines (EFSMs) and apply passive testing to a GSM signalling protocol [TCI99]. The main theme of this line of work is of using the input/output trace to reconstruct the (finite) state of the protocol implementation and check that the implementation is not in an ‘error’ state. The main disadvantage of these methods is the lack of expressiveness. Protocols can have an infinite (or at least a very large) number of states and are organized hierarchically in layers, which they cannot handle. In addition, while algorithms for trace analysis are presented, there is no support for error diagnostics - errors show up as inconsistencies in a trace and the whole trace must be manually analyzed to find the root of the error.

A second line of research involves the design of test oracles: programs that take a protocol specification and a trace produced by the implementation under test and indicate whether the test was passed or not. There are several ways in which the protocol specification may be written. Richardson and Dillon [DY94, Ric94, ORD96, DR96, RAO92] describe ways of using formulae written in their graphical temporal logic, GIL, to generate test oracles. Jagadeesan et al [JPO95, JPP+97] use safety-property temporal-logic specifications of telecommunication software to generate oracles. While it is easy and intuitive to write real-time properties of protocols in a logic like GIL, temporal logics are not expressive enough to represent complete protocol state machines, or to set up multiple layers of protocol properties.

Callahan et al [CS96, CEM98] use Promela and LTL to represent protocol specifications and use model-checking techniques to generate test oracles. The limitation of their method is that the entire protocol trace must be represented as a single Promela process which is infeasible if the trace has more than a hundred thousand events.

Bochmann et al have investigated test result analyzers for protocol specifications written using the formal description techniques LOTOS, SDL and Estelle [BDZ89, BB89, BDD+90, EB95, PYB96]. These languages are powerful, expressive and come with several correctness and diagnostic tools. However, the standard compilers for these languages do not support passive test result analysis and so new algorithms are presented. The Test Result Analysis (TETRA) [BB89] tool includes an offline trace analysis algorithm that takes an observed trace, translates it to a LOTOS process and checks it against a LOTOS specification by using depth-first search techniques. A disadvantage of this method is that if the trace consists of millions of events, then translating it into a LOTOS process will be infeasible. In addition, it cannot be used for online analysis as the trace is being generated. The Tango tool [EB95] generates online trace analyzers from specifications written in Estelle. However, the performance of their analyzers is quite poor - ranging from 10 packets per second for a complex (800 transition) protocol to 250 packets per second for a small protocol.

A key theme of this entire line of research involves dealing with non-deterministic protocol specifications. All the algorithms used need to determinize them ‘on-the-fly,’ and the performance of the oracle is dependent on the number of choices it can make at each step. The main reason for starting from non-deterministic specifications is that these tools have been only applied on OSI protocols for whom LOTOS or Estelle specifications are already available. This makes it incumbent on any testing tool to treat these specification as starting points of the analysis. We believe that much better performance can be achieved by allowing the programmer to rewrite the specification as a deterministic monitor. For instance, the Tango trace analyzer for a transport protocol can at most analyze a few hundred packets every second. In contrast, our deterministic monitor for AODV analyzed tens on thousands of packets every second. The memory hit of determinization possibly accounts for this disparity. While the TETRA and Tango work has been quite successful in finding errors in small traces of ISO protocols, they have not been used
to analyze Internet protocols, for whom formal specifications are rarely available. We believe that our methodology of programming passive monitors directly and then checking them against the protocol specification are more suited and efficient in the Internet protocol world.

4.1.3 Run-time Monitoring

It has long been recognized that software testing does not guarantee correctness. While the guarantees provide by test coverage metrics might be adequate for application software, critical software such as an operating system or software that runs telephone switchboards often needs to be monitored at run-time. A run-time monitor could check that an implementation is in a dangerous state and alert an administrator or call corrective programs. The earliest proposed application of run-time monitoring to networks that we know of is the Overseer [FP76]: an active monitor that uses a specification of network behavior presented as program graphs to monitor and control access to key resources in an early version of the Internet. More recently, Smith et al [SCA] proposed to use active network technology to implement the overseer functionality. Unfortunately neither of these systems was implemented.

Savor and Seviora [HKL+95, SS96, SS97] advocate a monitoring approach called *software supervision* to find bugs in telecommunication software. They use a formal specification written in SDL to check the inputs and outputs of a software component. Their approach has been applied to network protocols with results and limitations similar to those of Bochmann et al desribed in the previous section.

Diaz et al [DJC94] propose to design self-checking distributed systems. In their work, the software is developed in terms of a *worker* and an *observer*, where the observer executes concurrently with the worker and checks for deviations from a formal specification written using Petrinets. They have extended this work to network protocols and checked some low level OSI protocols.

Recently, several groups interested in protocol verification have used verification techniques to monitor and check single executions of software. Some of the algorithms they use are applicable in passive protocol testing, as are the FSM and EFSM algorithms developed by many of the groups mentioned above. In addition, the run-time verification research pays more attention to model-checking and slicing tools that can be used to diagnose errors. However, the major thrust of this research is in making internal events observable - for instance, instrumenting the monitored software to produce external events whenever a particular state is entered. For network protocols, this is unnecessary because the protocol standard strictly describes the set of observable actions.

Lee et al. [LKK+99, KVBA+98, KVBA+99] describe a general architecture, MaC, for the Monitoring and Checking for software, including network protocol software. They describe a tool Java-MaC for checking Java software. Java-MaC instruments Java code to produce low-level events, and then analyzes these events using a high-level event language, MEDL. MEDL has been applied to the runtime monitoring of routing protocol simulations [BGK+02]. However, MEDL itself is not expressive enough to represent protocol events, protocol state, or multiple layer of protocols. Its implementation is in Java and is not efficient enough to handle large traces of packets. Moreover, it does not provide any diagnostics on why an event occurred. However, we believe MEDL can be extended to suit passive protocol monitoring and we shall describe an extended version in the next chapter.

4.1.4 Network Intrusion Detection

Network Intrusion Detection Systems (NIDSs) are used to monitor network traffic to look for the ‘bad users’ in the network. Although NIDS have not been used for correctness testing, several of the techniques they use are applicable to passive protocol monitoring.

In order to identify malicious behavior, NIDSs need to reconstruct protocol implementation behavior and check it against an expected profile. There are several ways in which such an expected
profile can be constructed, and this gives rise to different kinds of NIDS [Gro01, Axe99, Kva99].

**Attack Signatures** are well-known patterns to watch for in network traffic. For instance, a port-scan attack can be defined as a series of TCP SYN packets sent to all the ports on a destination. A signature-based detection system for this attack inspects all SYN packets and flags an alarm if many ports on a single destination are scanned from the same source. Signature-based detection is the most popular method in current NIDS, primarily because several organizations maintain databases of all known attacks on a network, and very few of the attacks on a network are new.

**Traffic Anomalies** represent deviations from a traffic profile. Anomaly-based detection systems use the fact that intrusions are unusual or anomalous events, and should be detectable as such. Such NIDSs first have a learning phase when they construct a model of *usual* behavior of users on the network. Then, in the detection phase, any behavior by a user that violates this usual behavior is flagged as a possible intrusion. The advantage of this technique is that it can potentially catch a number of new attacks, and identifies many intrusions that cannot be cast as attacks. For instance, a user logging in at 2AM may not count as an attack but will be flagged as anomalous behavior for the user. On the flip side, there is no guarantee that such a NIDS would catch implementation errors.

**Specification-based** intrusion detection is a relatively new technique. It takes a specification of how users should behave, and uses it to identify errors in a network stream. For example, any email user must first log in to the mail server, transfer incoming or outgoing email, and then log out. This specification can be represented as a state machine for each user, every event detected causes a transition in the machine, and the condition checked is that no bad state is ever reached. While a specification-based system has the advantage of being more expressive than signature-based detection, it is also less efficient than because of the state that it must maintain.

Of these, the specification-based NIDS have an approach that is closest to ours - they check protocol implementations for deviations from an attack specification.

Echmann, Kemmerer and Vigna use state transition diagrams to describe intrusions and to generate detection modules [VK99, EVK00, EVK01]. The state diagrams they use can be seen as formal specifications of user and network behavior. The diagrams in themselves are not expressive enough to represent protocol behavior and so cannot be used for protocol testing. On the other hand, this research demonstrates an architecture for executing state transition diagrams at network speeds that can be reused for passive monitoring.

The Bro NIDS consists of an event engine and a policy script interpreter [Pax99]. The event engine contains protocol modules written in C++ that can reconstruct protocol events from packet streams. For instance, the TCP event engine takes a packet stream and produces events such as `connection_established`. The policy language is used to specify intrusions: it is a domain specific language with just enough constructs to write event handlers that identify attacks and log them. While the Bro policy language in itself is quite powerful and one can express state machines in it, it is expected that all protocol analysis be done in the event engine layer. For intrusion detection, only a few low-level protocols need to be analyzed, and so this architecture is adequate. However, one should not view this as suitable for passive testing because writing new protocol monitors in C++ and maintaining them will not be a simple task. We would need tools to ensure monitor correctness. Also, Bro provides diagnostics through hard-coded “trace” functions that log events as they occur. We would like some better diagnostics that log only those events that cause an error.

It is also instructive to look at the limitations that have been discovered for NIDS because some of these limitations may be applicable to passive protocol monitoring as well. Ptacek and Newsham [PN98] describe limitations in intrusion detection techniques that stem from the fact that an
intruder can often fool the NIDS either because of errors in the protocol monitor, or because of ambiguities in the protocol. In addition, Paxson [Pax97, Pax99] shows that in the presence of network errors, NIDS cannot distinguish between the errors and attacks and so can produce false positives. Handley et al. [HPK01] describe an approach to resolve the problems due to specification ambiguities of network protocols. They advocate placing a ‘normalizer’ in the path of the packet stream, before it reaches the intrusion detection system or the device under test. The normalizer takes the stream of packets and normalizes them to resolve ambiguities. A similar approach is advocated as ‘protocol scrubbing’ in [MWJH00]. Since these are intrusive active techniques that modify the packet stream in the network, we do not adopt them for our purposes. Instead, we attempt to solve the problems by proving our monitors correct by model-checking them and by developing strategies to deal with network errors and specification ambiguities.

4.2 Our Earlier Work

This thesis is founded on earlier work with Carl Gunter and Davor Obradovic as part of the Verinet project (http://www.cis.upenn.edu/verinet). We first used the WRSPM model to describe a telecommunication protocol in [BGG+98]. The WRSPM model exposes several vulnerabilities in protocol software: the program may violate the specification, the specification may fail to satisfy the requirements, and assumptions made on users and the network may prove to be incorrect. In [BOG02], we show how protocol verification techniques can be used effectively to prove correctness of specifications with respect to requirements. However, current verification technology is inadequate to protect against the other vulnerabilities in protocol implementations. We propose to use a passive protocol monitoring framework to address these other issues.

The immediate inspiration for the thesis comes from our work on Verisim [BGK+02, BGK+00], a system for checking simulation traces for conformance with the standard, and for satisfaction of the requirements. Verisim fits neatly with the conventional testing technique of simulation, and adds a dimension of formal trace analysis to find bugs in simulator code. Several techniques are proposed for dealing with scalability issues, and to reduce false alarms. When both conformance and requirements tests are carried out for a protocol, and a fault is discovered, it may be possible to adjudicate whether the fault lies in the implementation (a bug in \( P \)), or whether the specification is inadequate (a design error in \( S \)). We describe a technique for this Fault Origin Adjudication in [BGO00].

In order to apply our monitoring and analysis techniques in live network conditions, we need to address some of the packet capture issues that plague passive network monitoring. Some of these problems are discussed and addressed in [BCMG01]. Requirements for a new passive monitoring language are detailed in [BG02b]. Finally an application of passive monitors in SMTP surveillance is described in [BG02a].
Chapter 5

Problem Description and Related Work

We first describe in detail the problem that this thesis attempts to address. Then we survey related research and compare it with ours in terms of our goals, methodology, and experiments carried out in the form of case studies.

5.1 Goal

The goal of this thesis is to develop a general, flexible, passive protocol monitoring system and apply it to the analysis of a variety of network protocol implementations.

Protocols. We want to be able to analyze a large class of protocols. While it is important to check infrastructural protocols, such as routing and transport protocols, it is also interesting to check popular application protocols, such as Internet messaging or file transfer. Similarly, while finding errors in established protocols is likely to have a wider effect, inserting our testing technique into the development cycle of newer protocols is also important. So we require that our passive monitoring system be able to handle the class of protocols characterized by the following:

IP Stack network layer, transport layer, and application layer protocols based on the Internet Protocol.

Variable Participants multi-party protocols as well as two-party protocols

Clear-text Messages protocols that exchange un-encrypted messages, or encrypted messages where the keys are provided to the monitor

Formal Specification protocols that have unambiguous (formal) specifications or for whom such a specification may be derived from the standard

Properties. For this class of protocols our system should be capable of checking traces produced by an implementation for deviations from both the requirements and the specification. The class of properties checked by our passive monitoring system is limited by the following:

Observable: properties of protocol state, timers, and higher-layer events that can be reconstructed from the messages

Safety: properties whose violations can be detected over a finite trace
We will demonstrate several significant properties in our case studies that are both observable and safe. For instance, while our system cannot check for liveness properties (a response is eventually produced for every command) most protocol liveness requirements are typically time-bound (a response is produces in 5s) and so can be cast as safety properties. For a more formal treatment of observability, the reader is referred to the notion of fault observability in fault secure distributed systems [DJC94]. Run-time monitoring research has discovered that only recursive, safety properties can be monitored [Vis00].

**Monitoring Environment.** Finally, we require our passive monitoring system to be flexible enough to be applicable in the following monitoring environments

- **Offline Trace** Packet traces are captured by a monitor and analyzed offline.
- **Simulation** Event traces are generated by network simulations and analyzed immediately after.
- **Online Monitoring** Packets are captured from a live network and analyzed online.

### 5.2 Methodology

We have chosen to use a programmable monitoring architecture, Network Event Recognition, that we describe in detail in the following chapter. We design domain specific languages for programming the passive protocol monitor. The monitor program can be seen as a monitorable rendition of a formal specification of the protocol. We then develop tools and techniques for finding errors in packet traces and mapping them to deviations from the specification and flaws in the implementation.

We evaluate our monitoring system and others on the following criteria

- **Expressiveness** Can we easily express all the safe, observable properties corresponding to protocol specifications and requirements? Can we describe several layers of protocols and monitor them at the same time?

- **Correctness** Is the protocol monitor correct? Does it reconstruct higher layer events correctly? Does it generate any *false positives*: errors when none exist?

- **Diagnostics** Can we provide detailed information about the cause of each error? Can we use the error to pinpoint a bug in the protocol software?

- **Efficiency** How many protocol participants can the system analyze? For offline analyses, how large a packet trace can be feasibly analyzed? For online analyses, how many packets per second can we analyze?

### 5.3 Case Studies

To experimentally validate our system we carry out case studies that demonstrate NER for a number of protocols and monitoring environments. We have chosen the following three case studies that exercise several of the features and tools that we have developed.

- **Routing Protocol Simulations**
  - **Protocol**: AODV - a new, network layer, multi-party protocol
  - **Properties**: Conformance, Functionality
  - **Monitoring Environment**: NS network simulations
**Internet Mail Forwarding**
- Protocol: SMTP - an established, application layer, two-party protocol
- Properties: Conformance
- Monitoring Environment: Live test network

**Reliable In-order Transport**
- Protocol: TCP - an infrastructural, transport Layer, two-party protocol
- Properties: Conformance
- Monitoring Environment: Offline packet traces
Chapter 6

Programming in NERL

6.1 Recognizer Language: NERL

In this section we will describe a language, NERL, for writing event recognizers. We first present an example recognizer for a simple ping protocol. In the following sections we describe the syntax and semantics of NERL in more detail.

The ping protocol is inspired by the ICMP Echo feature [Pos81]. When a user at node A needs to know whether node B is alive, node A sends a request to node B and if node B is alive, it sends back a reply to node A. When this reply reaches A, it produces an IsAlive event and sends it to user. The message exchange is shown in Figure 6.1. 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 process at node B can be represented by the state machine in Figure 6.2- it gets an EchoRequest and replies with an EchoReply that contains the same sequence number and data. In the figure, input events are suffixed by “?” while output events end in “!”.

Figure 6.1: Ping Message Exchange

Figure 6.2: State Machine for Node B
On the other hand, the process at 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 6.3.

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

1. Check that the receiver follows the state machine in Figure 6.2. In particular, we want to check that the receiver produces EchoReply(S,D) only if the sender earlier sent it an EchoRequest with the same sequence number and data.

2. Reconstruct the IsAlive higher-layer event. We want the monitor to generate a meta-event indicating that the ping sender A must have produced an IsAlive event. The data contained in the IsAlive event may be used by some higher layer recognizer.

The first task corresponds to error-checking, while the second reconstructs messages on the abstract channel at the next higher-layer.

Consequently, our ping monitor will implement a state machine as shown in Figure 6.4. Notice how 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

Figure 6.2: Ping Receiver Specification

Figure 6.3: Ping Sender Specification

Figure 6.4: Ping Monitor Specification

45
on the wire, such as the one in Figure 6.1. 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 BadEchoReply error-event is generated. Note that the states OK and ERR are only transient states between the monitor receiving an input event and producing an output event.

We can use this state machine to program a ping recognizer in NERL. First, we need to define a variable to store the state of the monitor. We represent it by a variable status that can have the values {CLEAR, WAIT, DONE}. In addition, we need variables to store the sequence number and data that were seen in the last EchoRequest:

```c
#define CLEAR 0
#define WAIT 1
#define DONE 2
int status;
int icmp_seq;
string icmp_data;
```

In general, we can also declare variables with type bool or double, or compound types such as records and arrays.

We also need to declare the inputs and outputs of the recognizer. There are two primary inputs to the ping recognizer. We declare them in NERL as follows:

```c
input event ty_echo EchoRequest;
input event ty_echo EchoReply;
```

Each NERL event must be given a type for its attributes. For instance, in this event declaration, the input events EchoRequest, EchoReply have attributes defined by the ty_echo type that represents ICMP echo packets. This type is defined by the following C-style type declaration

```c
typedef {
    int ip_src;
    int ip_dst;
    int icmp_type;
    int icmp_code;
    int icmp_id;
    int icmp_seq;
    string icmp_data;
} ty_echo;
```

The ping monitor described in Figure 6.4 also has two outputs:

```c
output event string IsAlive;
output event string BadEchoReply;
```

Here, the string attribute of the IsAlive event contains the icmp_data exchanged, and the string attribute of BadEchoReply contains an error message.

Then, we program each state transition shown in Figure 6.4. When an EchoRequest is seen in state CLEAR, we update the icmp_seq and icmp_data variables and the state changes to WAIT. The transition is written in NERL as follows:

```c
transition EchoRequest(e)
    OccurredWhen (status == CLEAR) -> {
        status = WAIT;
        icmp_seq = e.icmp_seq;
        icmp_data = e.icmp_data;
    }
```
The transition keyword introduces the state transition, the expression to the left of \( \rightarrow \) is the guard or trigger - it must be true for the transition to be executed. This transition will be executed when the input event EchoRequest occurs and the monitor is in the CLEAR state. The variable \( e \) refers to the attributes of the EchoReply event. Note that since the attributes of the EchoReply event have a record type \( ty\_echo \), the different fields are accessed by record projection on \( e \). The right hand side of the transition (after \( \rightarrow \)) contains a statement that transforms the state by assigning values to variables. In general the statement can contain if conditionals, and while loops as well, using a C-like syntax.

Next, when the EchoReply input event is seen, we first check whether it contains the same sequence number and data as the request. If it does not, then we flag an error event BadEchoReply. The NERL syntax for this error event definition is as follows:

```perl
event EchoReply(e)
    OccurredWhen ((status != WAIT) ||
    (e.icmp_seq != icmp_seq) ||
    (e.icmp_data != icmp_data)) \rightarrow BadEchoReply(b)
    WithAttributes
    {b= "Incorrect Echo Reply"}
```

The syntax of event definition is similar to that of state transitions. The event keyword introduces the event definition, and the guard expression to the left of the \( \rightarrow \) must be true for the event definition to be executed. Here, the event definition will be executed only when the EchoReply input event has occurred (with attributes \( e \)), and either the monitor is not in the WAIT state, or the sequence number and data attributes \( (e.icmp_seq, e.icmp_data) \) of the EchoReply event are different from the stored sequence number and data seen on the last EchoRequest. If the expression to the left of \( \rightarrow \) is true, then the event to right is generated. In this case, the BadEchoReply (error) event is generated. The WithAttributes construct lets us attach a string attribute \( b \) to this event indicating the cause of the error.

On the other hand, if the reply is correct, then the monitor must generate the IsAlive meta-event that node A would have produced:

```perl
event EchoReply(e)
    OccurredWhen ((status == WAIT) &&
    (e.icmp_seq == icmp_seq) &&
    (e.icmp_data == icmp_data)) \rightarrow IsAlive(a)
    WithAttributes
    {a = e.icmp_data};
```

In this case, the WithAttributes construct is used to send the data content of the reply to a higher-layer recognizer, which may use it to analyze some higher-layer feature of the ping program. For instance, the icmp_data often contains a timestamp that can be used to compute the round trip time between ping sender and receiver.

This completes the specification of the NERL recognizer for the simple ping monitor shown in Figure 6.4. However, while this simple event recognizer will work for the single connection between nodes A and B, we would like to have one such recognizer for each possible ping connection between every two nodes on the network. So in order to identify this recognizer, we use the IP addresses of A and B and the ICMP attribute icmp_id that identifies the connection.

The complete event recognizer for ping includes these variables, and an Init input event that initializes them. We also include an additional error check for EchoRequest events. The complete NERL specification of the ping monitor is shown in Figure 6.5. The recognizer is enclosed by the Recognizer and EndRecognizer keywords and preceded by the type definitions. Having written the ping recognizer, we can now use it to monitor ping sessions. For instance, consider the
Recognizer Ping =

typedef {
    in ip_src;
    in ip_dst;
    in icmp_type;
    in icmp_code;
    in icmp_id;
    in icmp_seq;
    s rin icmp_data;
} e o;

typedef {
    in ip_src;
    in ip_dst;
    in icmp_id;
} in ;

input event in Init;
input event e o EchoRequest;
input event e o EchoReply;

output event s rin IsAlive;
output event s rin BadEchoRequest;
output event s rin BadEchoReply;
output event oo Done;

in ip_src;
in ip_dst;
in icmp_id;

#define CLEAR 0
#define WAIT 1
#define DONE 2
in status;
in icmp_seq;
s rin icmp_data;

transition Init(i) > (ip_src = i.ip_src;
                   ip_dst = i.ip_dst;
                   icmp_id = i.icmp_id;
                   status = CLEAR);

event EchoRequest(e)
   OccurredWhen (status != CLEAR) > BadEchoRequest(b)
   WithAttributes
      {b = "Multiple Echo Requests"};

transition EchoRequest(e)
   OccurredWhen (status == CLEAR) > (icmp_seq = e.icmp_seq;
                                         icmp_data = e.icmp_data;
                                         status = WAIT);

event EchoReply(e)
   OccurredWhen ((status == WAIT) &&
                  (e.icmp_seq == icmp_seq) &&
                  (e.icmp_data == icmp_data)) > IsAlive(a)
   WithAttributes
      {a = e.icmp_data};

event EchoReply(e)
   OccurredWhen ((status != WAIT) ||
                  (e.icmp_seq != icmp_seq) ||
                  (e.icmp_data != icmp_data)) > BadEchoReply(b)
   WithAttributes
      {b = "Incorrect Echo Reply"};

event (IsAlive | BadEchoReply | BadEchoRequest) > Done(d)
   WithAttributes {d = true};

transition (IsAlive | BadEchoReply | BadEchoRequest) > (status = DONE);

EndRecognizer;

Figure 6.5: A Complete Ping Recognizer
ping message sequence shown below. This sequence was generated by using the ping command
on a Linux box. The ping is being sent from 158.130.012.217 to 158.130.012.004.

This ping packet trace was captured by our packet capture module. There are three pairs of events,
each amounting to one ping. Each line is prefixed by a timestamp indicating the time at which the
event was seen by the monitor, then the input events for ping are indicated with the relevant IP
and ICMP fields. We do not show the icmp_data field here, since it is unprintable.

When this ping packet trace is fed to the ping recognizer we have programmed, the recognizer
produces an output trace of the form

```
14:22:37.338712 IsAlive Event <ip_src:158.130.012.004,ip_dst:158.130.012.217,
icmp_id:13043>
14:22:38.337671 IsAlive Event <ip_src:158.130.012.004,ip_dst:158.130.012.217,
icmp_id:13043>
icmp_id:13043>
```

The ping recognizer for this particular connection, identified that there were three correct replies
that must have resulted in three IsAlive events at the sender. We also print the connection for
which this event was recognize.

On the other hand, suppose one of the replies was incorrect - the reply had an incorrect se-
quence number:

```
14:22:37.337706 EchoRequest Event <ip_src:158.130.012.217,ip_dst:158.130.012.004,
icmp_id:13043,icmp_seq:0>
14:22:38.337671 EchoReply Event <ip_src:158.130.012.004,ip_dst:158.130.012.217,
icmp_id:13043,icmp_seq:1>
14:22:39.337647 EchoReply Event <ip_src:158.130.012.004,ip_dst:158.130.012.217,
icmp_id:13043,icmp_seq:2>
```

Then the ping recognizer would produce an output trace of the form

```
    <ip_src:158.130.012.004, ip_dst:158.130.012.217, icmp_id:13043>
```

The print functions for each output event are automatically generated by the NERL compiler.

We have not really described how the ping recognizer is initialized, or how the packet capture
module interacts with the ping recognizer. We will address these aspects when we describe the
module language, NERL_MOD, later in this chapter. Now that we have described how to program
and execute a simple recognizer, let us look at the NERL syntax and semantics in more detail.
6.1.1 NERL Syntax

The NERL language provides mechanisms for event filtering, correlation, aggregation and abstraction. However, we do not include parsing functions because we believe that string parsing and packet parsing are better carried out by specialized languages such as Lex/Yacc and PacketTypes. Moreover, we are particularly interested in reasoning about event recognizers, and parsing is an operation that is difficult to formally analyze.

NERL recognizers deal with four types of values: booleans, integers, doubles (floating point numbers) and strings. Time is represented as a double and not treated specially. The standard arithmetic operations are allowed on integers and doubles - operations involving doubles always result in doubles. Strings are variable sized blocks of bytes (not necessarily printable characters) and can be assigned, concatenated, and checked for equality. Arithmetic and string comparisons result in boolean values that can be combined using standard boolean operators.

There are three ways to create complex data structures: records, arrays and variable sized arrays. Records are defined by C-style typedefs as we have seen in the ping recognizer (Figure 6.5). Arrays are as in C, but variable sized arrays can be extended at the end by the push primitive. For instance, push(a,n) adds n cells to the end of the array a. The dual pop operator deletes elements from the end of a variable sized array. For instance, pop(a,n) deletes n cells from the end of the array a. Memory management is handled by the NERL compiler. For convenience, we allow type variables that abbreviate other type expressions. Recursive types are not allowed.

There are two kinds of variables that represent the state of the recognizer. The persistent state is stored in state variables and temporary state is stored in local variables. In addition, event attributes provide a form of transient state that is created and destroyed in a single execution of the recognizer. All state variables and events must be declared along with their types before they are used.

An event recognizer consists of event definitions and state transitions. Each execution of a recognizer can be considered a “round”. In each round, an input event arrives, triggering state transitions and generating events that trigger more transitions and so on. In each round, at most one instance of each event may be produced and each state variable may be modified at most once. At the end of the round, the recognizer is in a new state and produces some output events. We disallow circular dependencies between event definitions and state transitions, to avoid deadlock.

Statements are used to modify state and assign attributes to events. Most of the expressions, l-values, and statements will be familiar to C programmers. We describe the more unusual NERL constructs in more detail.

Primed L-values

During a single round, it is often useful to be able to refer to the old value at a location as well as the possibly new value. Primed L-values ($L'$) refer to the new value assigned to the location $L$ in this round, while $L$ itself refers to the old value. Only persistent state variables can be primed, temporary variables such as local variables and event attributes cannot.

Each state variable maintains its old and new values through the round and at the end all the changes are committed (new values replace old). For instance, consider the statements

```plaintext
icmp_seq = e.icmp_seq;
if (icmp_seq != icmp_seq') then new_seq = true
else new_seq = false;
```

Here, the first assignment changes only the primed variable and not the unprimed one, so the if condition in the next statement makes sense - it checks if the value of the variable has changed in the current round. At the end of the round, an implicit assignment will take place.

```plaintext
icmp_seq ← icmp_seq'
```
Note that if a variable is modified twice in the same round, we can no longer talk unambiguously about the new and old values - there would be two new or two old values at some point in the round. To avoid such ambiguity, and to retain a unique meaning for ‘the new value’, we restrict variable assignments so that a location can be assigned to at most once in each round.

Event Expressions

Event expressions are used as guards in transitions and event definitions. An event expression takes events, denoted by event variables, and combines them to produce a new event. Each event can be thought of as a combination of a boolean flag that denotes its occurrence and attribute fields that carry additional information. So an event expression that denotes an event is a combination of

- a boolean expression that denotes whether or not the event has occurred, and
- a variable binding that allows access to the event’s attributes.

So for instance, the event denoted by the event expression \( X(x) \) occurs whenever the event \( X \) has occurred, and the temporary variable \( x \) is bound to \( X \)’s attributes. The special input event \( \text{Init}(x) \) occurs only when the recognizer is initialized; its attributes \( x \) are used to initialize the recognizer state. The timer event \( \text{Tick}(t) \) occurs at every round and its attribute tells the time \( t \). The special output event \( \text{Done}(x) \) is used by the recognizer to indicate that it has finished its analysis and can be deleted.

\( Ev_1 \& Ev_2 \) occurs when both \( Ev_1 \) and \( Ev_2 \) occur, and it binds temporary variables in \( Ev_1 \) and \( Ev_1 \) to the corresponding event attributes. \( Ev_1 \mid Ev_2 \) occurs when one of the two events occurs, but it does not bind any temporary variables.

Finally, the \( \text{OccurredWhen} \) construct is used to filter and aggregate events. In \( Ev \text{OccurredWhen } E \), \( E \) expresses a predicate on the event attributes in \( Ev \) and on the current state. This event expression has occurred when \( Ev \) has occurred and the predicate \( E \) is satisfied in the environment that includes the temporary variables bound on \( Ev \).

For instance, the event expression below occurs when \( \text{EchoReply} \) has occurred, and when its attributes \( e \) and the state variable \( \text{status} \) satisfy the predicate after \( \text{OccurredWhen} \).

```
EchoReply(e)
OccurredWhen ((status != WAIT) ||
(e.icmp_seq != icmp_seq) ||
(e.icmp_data != icmp_data))
```

Note that the sub-expression \( \text{EchoReply}(e) \) binds the temporary variable \( e \) to the record attribute of the \( \text{EchoReply} \) event, and the fields in \( e \) can then be used in the expression after \( \text{OccurredWhen} \).

Commands

A NERL recognizer executes a sequence of commands, each of which could be an event definition or a state transition. While event definitions use existing events to generate new ones, state transitions are used to modify state variables. Both these kinds of commands are guarded by an event expression \( Ev \) - when \( Ev \) occurs, the temporary variables in \( Ev \) are bound to the corresponding event attributes and the state transition is executed or a new event is generated.

State transitions are written as transition \( Ev \rightarrow S \), where \( S \) is a statement that modifies variables and is executed whenever \( Ev \) has occurred. Here \( S \) can contain assignments, if-then-else conditionals and while-loops, and can modify compound data structures.
The event definition \( \text{event } E_v \rightarrow X(x) \text{ WithAttributes } S \) generates the event \( X \), if it already does not exist, whenever \( E_v \) has occurred. The attributes of \( X \) are then initialized and assigned by the statement \( S \). Although \( S \) will typically only contain assignments to attribute fields, in some cases it is useful to allow complex statements such as while-loops in \( S \) for assigning to complex attributes such as arrays.

**Recognizer**

A recognizer is simply a collection of type definitions, variable and event declarations, event definitions and state transitions. Each recognizer defines the behavior of its instances. An instance of a recognizer can be thought of as a process that waits for input events, each event triggers one round of execution, it modifies state and produces output events.

In addition to the syntax rules, a well-formed recognizer must follow the following rules:

- If two event definitions produce the same event, then only one of them may be executable in a round.
- If two statements modify the same state variable, then only one of them may be executable in a round.
- It should be possible to order the list of commands such that each command only refers to events that are defined before it in the list, and only refers to primed variables that are modified by transitions before it in the list.

Of these, the first two avoid non-determinism - after a sequence of commands is executed there is a unique value for each state variable and event attribute. The last condition ensures that there is a deadlock-free execution sequence of commands. In addition to these rules, NERL recognizers must obey the scoping and typing rules we define later.

**Syntactic Sugar**

For convenience, we allow some relaxations of the syntax rules in the previous section. Semicolons are allowed at the end of type definitions and statements (C-style). If conditionals can be written without the attached else (an implicit skip). Also, local variables in statements can be written with implicit scope till the end of the statement block as in C. For instance

\[
\{ \text{int } i = 0; \\
\quad \text{while } (i<10) \text{ do } \{ i = i + 1; \} \\
\quad j = 1; \}
\]

is equivalent to

\[
\{ \text{int } i = 0 \text{ in } \\
\quad \text{while } (i<10) \text{ do } \{ i = i + 1; \} \\
\quad j = 1; \}
\]

In event expressions, we allow \( X \) without the attached attribute variable, to mean that the attributes are thrown away (a dont-care pattern). We often use \text{when} as an abbreviation for \text{OccurredWhen}, and \text{with} as an abbreviation for \text{WithAttributes}. In addition, we allow the copying of complex event attributes. That is, in event definitions, the attribute variable of the generated event could be directly assigned to another event attribute variable:

\[
\text{event } X(x) \rightarrow Y(y) \text{ WithAttributes } \{ y = x \}
\]
### Table 6.1: NERL Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrBndErr</td>
<td>Array bounds violation</td>
</tr>
<tr>
<td>EmptyArr</td>
<td>Deleting elements from an empty array</td>
</tr>
<tr>
<td>WriteErr</td>
<td>Writing to a location twice in one round</td>
</tr>
<tr>
<td>RepeatedEvt</td>
<td>Generating two instances of an event in one round</td>
</tr>
<tr>
<td>BadLocation</td>
<td>Accessing a location that does not exist</td>
</tr>
<tr>
<td>Error</td>
<td>Unexpected error</td>
</tr>
</tbody>
</table>

This is a common kind of attribute assignment and avoids having to write the detailed sequence of assignments it implies. Moreover, in the implementation it can be efficiently translated to a pointer assignment; this works only when both the left and right hand sides are event attributes.

### Run-time Errors

In addition to these judgments, every construct may evaluate to an “error” which can be thought of as a run-time exception. The exceptions raised by NERL are shown in Table 6.1.

When the recognizer cannot be evaluated, it raises an error. For instance, an error would be raised if an expression tries to access an array index beyond the range of the array (ArrBndErr). Similarly, we cannot evaluate statements that attempt to delete elements from an empty array (EmptyArr), or which try to modify a location that has already been modified in this round (WriteErr). Event definitions can generate new events as long as they have not already been generated in this round (RepeatedEvt). An implicit error is divergence - statements with while loops might execute indefinitely. Finally, the BadLocation error and the generic Error should never be raised by a well-typed program, they indicate flaws in the assumptions made on locations in the store. Our evaluation rules for NERL ensure that in each round of execution, either the recognizer returns with a modified state and event environment, or it diverges, or it raises exactly one of these errors.

While most of the evaluation rules are quite straight-forward, we describe some of them in more detail in the following sections.

### 6.2 Modules Language: NERL\textsubscript{MOD}

Earlier in this chapter, we described an event recognizer for the ping protocol. However, in itself this event recognizer description cannot be executed because it does not specify where the input events come from or where the output events are sent. In this section we describe a language for specifying how to ‘close’ recognizer descriptions to form an executable monitor. While the syntax of this language will be reminiscent of the recognizer language, it is important to realize that these languages are independent. In particular, our modules language will work even if the individual recognizers are written in C or Lex or Perl, as long as they obey the input-output event interface.

To build an executable monitor, the ping event recognizer must be put together with a packet capture module. When the packet capture module sees an ICMP packet, it sends it to the ICMP parsing module to extract the ICMP fields from the payload. When the ICMP parser discovers that the packet contains an Echo Request or Reply, it must send the packet to the Ping recognizer for the corresponding ICMP connection. This hierarchy is depicted in Figure 6.6.

NERL\textsubscript{MOD} is a language for formally describing such interactions between network event recognizers. It is reminiscent of several object-oriented languages, although it is simpler in practice.
In order to actually specify the monitoring stack in Figure 6.6, we first declare the input-output signature of each box. Then we specify how and when each recognizer instance is generated and initialized. Finally, we specify how the outputs of one box are fed as inputs to another.

The signature of an event recognizer consists of all its input and output event declarations. For instance, for the ping recognizer we described earlier in this chapter, the signature is declared as follows:

```plaintext
signature Ping_Sig = {
    input event ty_ping Init;
    input event tyEcho EchoRequest;
    input event tyEcho EchoReply;
    output event string IsAlive;
    output event string BadEchoRequest;
    output event string BadEchoReply;
    output event bool Done;
}
```

Here, the Init input event is executed only once - when the recognizer instance is first initialized. The Done output event indicates that the recognizer instance has completed its analysis and can be deleted. The rest of the input and output events represent the actual analysis of the ping protocol.

Similarly, the signature of the packet capture recognizer can be declared as follows:

```plaintext
signature PCap_Sig = {
    input event string Init;
    output event ty_ippkt IP;
}
```

Since this is at the bottom of the monitoring stack, there are no inputs to it. The Init event initializes the device from which it must read packets. The packet capture module we use in this section reads packets from a file, so the attribute of the Init event is the name of the file.

Finally, the signature of the ICMP packet parsing module can be described as follows:
signature ICMP_Parse_Sig = {
    input event bool Init;
    input event ty_ippkt IP;
    output event ty_echo ICMP_Echo;
}

This parsing module takes IP packets as input and produces ICMP packets. It is a stateless recog-
nizer, and the Init event takes a dummy boolean attribute. Note that although the packet capture
and parsing modules are written in C, these signatures provide all the information that NERL*MOD
needs to know in order to instantiate them and control their interaction with other recognizers.

Having declared the signatures, we must bind the recognizers to them:

Recognizer PCap_Sig PCap;
Recognizer ICMP_Parse_Sig ICMP_Parse;
Recognizer Ping_Sig Ping;

The primary reason for separating the declarations of the recognizer and signatures is that we
might want to maintain several versions of a recognizer - all with the same signature. For in-
stance, suppose we developed a more sophisticated ping recognizer PingMultiple that allowed
a sequence of EchoRequests followed by a sequence EchoReply events and matched them one-
to-one, then this new recognizer would also be declared as

Recognizer Ping_Sig PingMultiple;

So as long as recognizer have the same signature, they are equivalent as far as the modules lan-
guage is concerned.

Although the stack contains only one instance each of the PCap and ICMP_Parse recognizer,
there may be several instances of the Ping recognizer - one for each each connection. We declare
recognizer instances as follows:

instance PCap P WithAttributes
{init = "icmp.pcap");
instance ICMP_Parse I WithAttributes
{init = true);
instance Ping E[ty_ping c]
WithAttributes
{init = c);
This definition is executed whenever the output event ICMP_Echo, with attributes e, is generated by the instance I, and the type field of the ICMP packet contains 8 - the value corresponding to Echo Requests. When this expression to the left of \( \rightarrow \) is true, the event to right is generated. Here, the input event EchoRequest in the Ping instance E\[x\] is triggered. The WithIndex construct tells us which ping instance this packet should be forwarded to. Here, we specify that an ICMP Echo with attributes e should be forwarded to the instance monitoring the connection defined by \(<e.ip_src,e.ip_dst,e.icmp_id>\). Finally, the WithAttributes behaves the same as in NERL - it assigns attributes to the input event.

The outputs produced by the ping instances are then forwarded to an inbuild PRINT module that logs these events:

\[
\text{event E}[x].\text{IsAlive}(a) \rightarrow \text{PRINT} \\
\text{event E}[x].\text{BadEchoReply}(b) \rightarrow \text{PRINT}
\]

We have already shown the output of this monitoring stack earlier in this chapter. The complete NERL\_MOD program for ping monitoring is shown in Figure 6.7.

6.2.1 NERL\_MOD Syntax

The objective of the modules language is to describe a stack of recognizers and their interconnections. As such, the NERL\_MOD program is a sort of main function or class that takes other recognizers as components and produces an executable monitor.

While a number of constructs in NERL\_MOD are reminiscent of NERL, it is important to realize that the recognizer modules might be written in any language. Indeed, we have written event recognizers in C, Lex, and NERL and used NERL\_MOD to glue them together. A NERL\_MOD program is used to describe three main components: signatures, instance initializations and event forwarding.

**Signatures**

The signature declarations provide a way of uniformly defining the input-output characteristics of a recognizer. Irrespective of which language the recognizer is written in, it must obey a signature written in NERL\_MOD. A signature is defined by the construct

\[ \text{signature } T = \{ D_1; D_2; \ldots; D_m \} \]

Here each declaration \( D_i \) is either a type definition, input event declaration, or output event declaration. Events form the interface between recognizers, and signatures define their types. Every signature must contain the Init input event. When the signature contains only this input event, it specifies a capture module.

For instance, consider the simple ICMP parsing module that takes ICMP packets and produces ICMP echo requests and replies. Such a module can be easily written in C, or in a packet processing language such as PacketTypes. The signature imposes that the function that implements the
typedef {
    in int ip_src;
    in int ip_dst;
    in int icmp_id;
} in;

signature PCap_Sig = {
    input event s rin Init;
    output event i IP;
}

signature ICMP_Parse_Sig = {
    input event oo Init;
    input event i IP;
    output event e o ICMP_Echo;
}

signature Ping_Sig = {
    input event in Init;
    input event e o EchoRequest;
    input event e o EchoReply;
    output event e o IsAlive;
    output event s rin BadEchoRequest;
    output event s rin BadEchoReply;
    output event oo Done;
}

Recognizer PCap_Sig        PCap;
Recognizer ICMP_Parse_Sig  ICMP_Parse;
Recognizer Ping_Sig        Ping;
Recognizer Ping_Sig        Ping_Multiple;

instance PCap        P WithAttributes
                      {init = "icmp.pcap"};
instance ICMP_Parse   I WithAttributes
                      {init = true};
instance Ping        E[  ip src] WithAttributes
                      {init = c};

event P.IP(p) OccurredWhen (p.ip_p == ICMP) ->
    I.IP(q) WithAttributes
                      {q = p};

event I.ICMP_Echo(e) OccurredWhen (e.icmp_type == 8) ->
    E[x].EchoRequest(p) WithIndex
        x.ip_src = e.ip_src;
        x.ip_dst = e.ip_dst;
        x.icmp_id = e.icmp_id
    WithAttributes
                      {p = e};

event I.ICMP_Echo(e) OccurredWhen (e.icmp_type == 0) ->
    E[x].EchoReply(p) WithIndex
        x.ip_src = e.ip_src;
        x.ip_dst = e.ip_dst;
        x.icmp_id = e.icmp_id
    WithAttributes
                      {p = e};

event E[x].BadEchoRequest(b) > PRINT;
event E[x].BadEchoReply(b) > PRINT;
event E[x].IsAlive(a) > PRINT;

end

Figure 6.7: Ping Modules Description
recognizer, say ICMP_Parse, takes one input event containing IP packets and produces one output event containing ICMP echoes. So the signature of this module can be written as follows, with the required initializing event.

```plaintext
signature ICMP_Parse_Sig = {
  input event bool Init;
  input event ty_ippkt IP;
  output event ty_echo ICMP_Echo;
}
```

**Instance Definition and Initialization**

We have seen a recognizer language in this chapter where recognizers are required to have an `Init` event that contains initial parameters for the recognizer state. `NERL_MOD` provides a mechanism for defining how modules are to be initialized. There are two kinds of recognizer instances: single instances and instance tables.

Single instances are defined as

```
instance R X WithAttributes S
```

Here `X` is defined to be an instance of the recognizer `R` (`R` is defined elsewhere, possibly as a `NERL` recognizer). The statement `S` assigns parameters to the implicitly bound variable `init`. This variable `init` is sent as the attribute of the `Init` event when `X` is initialized.

Instance tables are defined as

```
instance R X[ν x] WithAttributes S
```

Here we define an entire group of instances `X[ν]` each of which behaves like the recognizer `R`. We think of `X` as referring to a hash table of instances, where the key is of type `ν`. `ν` is restricted to be a type that has a fixed size; it only consists of base types and records. When `X[ν]` is initialized, the statement `S` is executed to assign parameters to the variable `init`, which is then sent as the attribute to `X[ν]`’s `Init` event. Note that `S` could depend on the index `ν` of the instance it is initializing.

Instances are initialized on-demand - when an event arrives that must be forwarded to an instance that does not exist yet. Also, if no event arrives for an instance in a really long while, it may be deleted. However, semantically, it is useful to think of all possible instances being initialized in the beginning of the execution and existing till the end.

**Event Forwarding**

After initialization, all the work in the `NERL_MOD` program is done by event forwarding. If the individual instances are thought of as concurrent, layered processes, then event forwarding takes events from a lower layer and sends it to a higher layer. The simplest event forwarder is written as follows:

```
event X₁.O(o) → X₂.I(i) WithAttributes {o = i}
```

Here module `X₁`’s output event `O` is piped to `X₂`’s input event `I`. In general we can choose which output events to forward using the `OccurredWhen` construct and we can modify the corresponding input event’s attributes.

When the lower and higher layers both have instance tables, the event forwarder looks as follows:

```
event X[x].O(o) → Y[y].I(i) WithIndex S₁ WithAttributes S₂
```

Here, whenever the output event `O` is generate by any `X[x]` in the lower layer, the event `I` in the higher layer instance `Y[y]` is triggered. `S₁` defines the upper layer instance `y`, while `S₂` defines the attribute `i` of the event `I`. 

58
6.3 Extensions for Co-netoworked Monitoring

6.3.1 Input and Output Channels

We refer to the channel from the monitor to the device as the input channel, and the channel from device to monitor as the output channel. In a co-networked channel, input channels may be buffered. As a first step to programming a co-networked recognizer, we identify the channels and buffer parameters and annotate the signature with this information. For instance, if we wish to monitor the ping receiver in a co-networked environment, the the signature is annotated as follows:

```
signature Ping_Sig = {
ichannel  ITo[5,2];
ochannel  OFrom;

input event Ty_ping Init;
input event ITo Ty_echo EchoRequest;
input event OFrom Ty_echo EchoReply;

output event Ty_echo IsAlive;
output event string BadEchoRequest;
output event string BadEchoReply;
output event bool Done;
}
```

Here we declare that ping recognizer monitors two channels: an input channel called ITo that can buffer up to 5 packets and lose up to 2 packets in a row, and an output channel called OFrom. Since we are monitoring the ping receiver, the EchoRequest event is on the input channel and the EchoReply event is considered an output.

A recognizer implementing this signature must allow up to 5 EchoRequest events to be buffered. So while the sequence `EchoRequest(0); EchoRequest(1); EchoReply(0)` (numbers in the parentheses refer to sequence numbers) would flag an error event in the ping recognizer we described earlier, it should be allowed by a co-networked ping recognizer. This exemplifies the approach to co-networked event recognition: relax the monitored property to allow for buffering and loss.

To see a second example of channel annotations, if we were monitoring the ping sender then the channels would reverse:

```
signature Ping_Sig = {
ichannel  IFrom[7,1];
ochannel  OTo;

input event Ty_ping Init;
input event ITo Ty_echo EchoRequest;
input event OFrom Ty_echo EchoReply;

output event Ty_echo IsAlive;
output event string BadEchoRequest;
output event string BadEchoReply;
output event bool Done;
}
```
Here we specify that the ping sender may buffer 7 packets but cannot lose 2 packets in a row. This is to illustrate that the buffer parameters for different devices might be different. Generally, we start with smallest possible values \([0,0]\) and increase them as we find superfluous error events until we are satisfied that the errors produced indicate real errors in the protocol. This works quite well as long as the buffer parameters are small. An alternative is to accurately measure buffer sizes and loss bursts by carrying out experiments and use those as parameters to begin with.

Note that the ping recognizer we defined before monitors both sender and receiver at the same time, and so cannot be directly used in the above manner. In Chapter 10, we shall show how to modify and annotate such recognizers.
Chapter 7

NERL Language Implementation

For network event recognition to be successful, we need several kinds of tools that aid the user in programming protocol event recognizers, executing them, and diagnosing any errors that they may indicate in a protocol implementation. In this chapter, we describe several such tools for the NERL and NERL\textsubscript{MOD} languages.

**Execution** tools for NERL recognizers and monitor stacks.

- A NERL compiler that translates recognizers to C functions.
- A NERL\textsubscript{MOD} compiler that translates monitor stack descriptions (main modules) to an executable C program.

**Correctness** tools to check NERL recognizers for errors.

- NERL and NERL\textsubscript{MOD} type-checkers implemented as part of the compilers.
- A NERL model-checker that translates recognizers to Promela processes that can be model-checked in SPIN.
- A slicing tool for NERL that is used to automatically produce recognizers specialized for a particular output event.

**Diagnostic** tools that help in analyzing and understanding NERL error events.

- Event tracing: an enhancement of the NERL compiler that tags output events with the sequence of packets that caused it.
- Co-networked monitoring: a modification of the NERL\textsubscript{MOD} compiler that filters out errors caused by a co-networked channel.

**Analysis** techniques used in NERL protocol analyses.

- Tuning: a technique for analyzing NERL output events to pinpoint bugs in the implementation.
- Fault Origin Adjudication: a technique for deciding whether a NERL error indicates a flaw in the implementation or a flaw in the standard.
7.1 Compiling NERL Recognizers

In Chapter 6, we described the NERL language, with its typing and operational rules. In this section we describe how the language is implemented. The NERL compiler translates recognizers written in NERL to recognizers in C. The benefits of this approach are threefold:

1. The compiler does not need to concern itself with machine-code generation. It can rely on optimizations in the C compiler for efficient code generation.
2. Since the output of the compiler is readable C code, we can debug the compiler by inspecting this generated code and by using C-based tools such as lint.
3. If the performance of the recognizer is found to be inadequate, we can directly hack the C recognizer. For instance, if the NERL data structures are found to be inadequate, we can replace them by more efficient data structures using the generality of C.

Given the above considerations, one might ask why we do not write our recognizers directly in C to begin with. The reason is that for recognizers written in the restricted NERL language, we can develop several checking tools and transformations that would be difficult to develop for general C programs. So our translation strategy gains the benefits of using a domain specific language, while retaining the power of a general programming language.

7.1.1 Parsing and Type-Checking

The compiler is written in the functional language OCaml. We find that using this higher-order, strongly-typed language significantly simplifies the implementation. The NERL parser is written in OCamlYacc/OCamlLex and is translated from the syntax presented in the previous chapter (Table ??). NERL programs are converted into abstract syntax trees represented by OCaml data types.

The type-checker was useful in pointing out simple errors in NERL programming. For instance, in the ping recognizer, suppose we made some errors in the BadEchoReply event.

```
81: event (EchoReply(e) & MyEchoReply)
82:    OccurredWhen
83:    ((status != WAIT) ||
84:     (e.icmp_seq != icmp_seq) ||
85:     (e.icmp_data != icmp_seq)) -> BadEchoReply(b)
86:     WithAttributes {
87:       b="Incorrect Echo Reply"
88:     };
```

Here, there are two errors: First, on line 81, e should refer to the attributes of EchoReply and not to MyEchoReply. The type-checker points out this error as a type error when e is first accessed on line 84

```
ping.nerl:84.13: Warning: Not a record, cannot project
```

The second error is that in cutting and pasting line 84, we wrongly wrote `(e.icmp_data != icmp_seq)` when we meant `(e.icmp_data != icmp_data)` The type-checker indicates this error at the comparison on line 85 (the two variables have incomparable types).

```
ping.nerl:85.13: Warning: Not Arithmetic Type.
```
7.1.2 NERL-to-C Translator

The NERL-to-C translation is based on the big-step semantics for NERL - NERL constructs are translated to C program fragments with equivalent semantics. The details of the translation for many important constructs are shown in Appendix ???. We write the translation functions as $[R]_c$.

The values in NERL - bools, ints, doubles, strings - are translated to their C equivalents, but variables are stored differently because we need to maintain both old and new values. The operational semantics describes a simple implementation of cells involving partially committed assignments that get committed at the end of a round. Instead, the compiler uses round numbers to avoid the commit phase at the end of a round. Each variable is translated to a cell that contains a value, an oldvalue and an integer changed indicating when the value was last changed. For instance, the storage type double of variables is translated as follows

$$\text{double}_c = \text{struct}\{\text{int changed; double value; double oldvalue;}\}$$

If this cell is modified in round number $N$, then the value field is first copied into oldvalue, and changed is set to $N$. Then all subsequent accesses to this cell first check when it was last changed - if it was changed in this round then the old value is in oldvalue, otherwise it is in value. Other basic types are translated similarly.

NERL records translate naturally to C structs. Arrays are implemented by a separate array library:

$$\tau[n]_c = \text{array} /* \text{created by create_array(sizeof(\tau)_c, n) */}$$

This is so that run-time errors such as array-bound violations can be detected. The array library also implements variable-sized arrays.

Once the type translations are defined, it is straightforward to translate the type and state variable declarations

$$\text{typedef } \tau \nu \quad \text{typedef } [\tau]_c \nu$$

$$\text{state } \tau x = [\tau]_c * x$$

$$\text{event } \tau X = \text{struct}\{\text{bool flag; } [\tau]_c * \text{ attrib}\} X$$

These declarations define the signature of the recognizer, which is represented as a C-struct of the form:

$$\text{typedef struct}\{$$
$$\text{struct}\{I\} * \text{ inputs;};$$
$$\text{struct}\{O\} * \text{ outputs;}$$
$$\text{struct}\{S\} * \text{ state;}$$
$$\text{struct}\{L\} * \text{ locals;}$$
$$\} R_{\text{sig}};$$

For instance, here $I$ contains all the input event declarations, and $S$ contains all state variable declarations. More precisely, this type definition contains more than just the signature. It defines the type of the control block of the recognizer - a data structure that contains its inputs, state and outputs. Each instance of a recognizer $R$ can be represented by a single control block with type $R_{\text{sig}}$.

The recognizer itself is translated to a function that takes a pointer to such a control block that has exactly one input event enabled, and modifies the state and output events. The logical structure of the translated C function is as follows:
void R_recognize(R_sig * R_cb){
    if (Init.flag == true) R_init(R_cb);
    R_clear_locals(R_cb);
    R_clear_outputs(R_cb);
    R_cb->state->thisround ++;
    [C1]c . . .;
    [C2]c . . .;
    . . .
    [Ck]c . . .;
}

In the initial round, the control block is first initialized using R_init - memory is allocated for the recognizer state (R_cb -> state) and local events, and all state variables are initialized. In every round, the local and output events are cleared, then all the state transitions and event definitions are executed in some order. The variable R_cb -> state -> thisround contains the round number and is used to update the changed attributes of store cells.

Given the operational rules, the translations of the L-values, expressions and statements are straightforward. Indeed, the original NERL syntax and operational semantics were themselves C-like for these constructs.

Events are represented in C by boolean flags and attached attributes. The flag indicates that the event has occurred, and if it has, then the attrib field points to its attributes.

[event \tau X]c = \text{struct}\{\text{bool flag;}[\tau]c * \text{attrib}\} X

The translation of event expressions is higher-order - it produces a function.

State transitions are translated by applying such an event function to the state modification statement, and event definitions apply such event functions to an event generation statement.

One important decision to be made while compiling recognizer commands is their order of evaluation. We use the command ordering described in the operational semantics, formalized by the following rules.

\begin{align*}
\text{event } Ev_1 \rightarrow X(x) \quad \text{WithAttributes } S_1 < \text{transition } Ev_2 \rightarrow S_2 \quad \text{if } Ev_2 \text{ contains } X \\
\text{event } Ev_1 \rightarrow X(x) \quad \text{WithAttributes } S_1 < \text{event } Ev_2 \rightarrow X(x) \quad \text{if } Ev_2 \text{ contains } X \\
\text{transition } Ev_1 \rightarrow S_1 < \text{event } Ev_2 \rightarrow X(x) \quad \text{WithAttributes } S_2 \quad \text{if } Ev_2 \text{ or } S_2 \text{ reads } L' \\
\text{transition } Ev_1 \rightarrow S_1 < \text{transition } Ev_2 \rightarrow S_2 \quad \text{if } Ev_2 \text{ or } S_2 \text{ reads } L' \\
\text{and } S_1 \text{ might modify } L' \\
\text{and } S_1 \text{ might modify } L'
\end{align*}

The recognizer translation rules sort the command sequence according to this ordering and produce the corresponding C function. To illustrate the translation rules we have described, we include the C recognizer generated from the ping NERL recognizer in Appendix A.2. For readability, we cleaned up the formatting and indentation of the generated code using the Emacs C package.

Run-time Checks

In addition to the above translation rules, we adopt some dynamic error checks so that the recognizer does not crash unexpectedly.

\begin{itemize}
    \item All array accesses are guarded by an array bounds check; if it fails, an error event is triggered and the recognizer is shut down.
    \item All array pop operations are guarded by an empty array check; if it fails, an error event is triggered and the recognizer is shut down.
\end{itemize}
If an event is generated twice, an error event is triggered and the recognizer is shut down.

If a state variable is modified twice, an error event is triggered and the recognizer is shut down.

These rules in conjunction with type soundness should mean that the translated recognizers never crash, they either shut down gracefully or they execute indefinitely.

### 7.1.3 Implementation Status

We have written a fully functioning NERL compiler in OCaml, it was used to generate code for all the recognizers describe in this thesis. However, it is still under development and occasionally produces code that is inefficient. One source of inefficiency is in the treatment of primed variables. In current version of the compiler, whenever a cell is modified, its current value is stored in oldvalue. Although this is necessary if the old value is going to be needed later in the same round, we find that many variables are read and updated exactly once, and so this update operation is redundant. As a result, we find that the generated code is much faster if we manually delete this oldvalue copy for such variables.

Another performance limitation is imposed by the simplistic data structures in the NERL language. While variable sized arrays are simple to implement and powerful enough to represent most protocol data structures, there are often more efficient ways of storing protocol data. In such cases, we find that the generated code can be sped up considerably by hacking the C code and replacing a generated data structure with a more efficient one. It might be useful to add a generic data structure interface to NERL that would allow a programmer to add an arbitrary C data structure implementation to NERL programs.

On the other hand, note the ubiquitous use of pointers in the translated code. Using code generation is particularly useful in avoiding pointer errors. We earlier wrote some protocol recognizers directly in C and had to spend many hours dealing with segmentation faults, which are almost entirely avoided when using the NERL compiler.

### 7.2 Compiling NERL\textsubscript{MOD} Programs

The NERL\textsubscript{MOD} compiler takes a recognizer stack description and produces the corresponding main function that describes how the complete monitor stack is to be executed. The NERL\textsubscript{MOD} compiler assumes that each individual recognizer has been compiled to a C function of the form:

```c
void R.recognize(R.sig * R.cb);
```

Here the signature \(R.sig\) is of the following form, where the internal structure of the state or local variables is not available.

```c
typedef struct{
    struct{T} * inputs;
    struct{O} * outputs;
    void * state;
    void * locals;
} R.sig
```

Recognizer instances are declared to be of the type of their signatures. While this is straightforward for single instance, instance tables are represented by hash tables and instance indices are used as keys.
\[
[\text{instance } R X[\nu x] \text{WithAttributes } S]_c = \{\text{HashTable } X\_table; \\
R\_sig \ast X\_cb; \\
X\_table = \text{new_hash_table}(\nu\_hash, \nu\_cmp);\}
\]

We use a popular open-source hash-table library written in C (glib).

The instance initialization commands are translated to C program fragments that set up the parameters and call \text{R\_recognize} with the \text{Init} event. While the initialization of single instances occurs at the beginning of execution, instances that are part of a table are initialized only when an event forwarding definition generates an event for it. Translating event forwarding is straightforward and is similar to the translation of event definitions in NERL.

Finally, the NERL\_MOD program is compiled to a \text{main} function that runs an infinite while loop, executing modules from bottom to top as events become available. Since a capture module does not depend on any other instance, we treat a capture instance \text{X} as equivalent to an event forwarding definition of the form \text{event ()} \rightarrow \text{X}. Then the order of execution is determined by a dependency tree of event forwarding definitions generated from the causal dependency relation between instances.

- The bottom (root) of the tree consists of capture module instances (\text{event ()} \rightarrow \text{X}).
- A forwarding definition of the form \text{event} \ldots \rightarrow \text{Y.I} has as its children all forwarding definitions of the form \text{event} \text{Y.O} \rightarrow \ldots
- A forwarding definition of the form \text{event} \ldots \rightarrow \text{Y[y].I} has as its children all forwarding definitions of the form \text{event} \text{Y[z].O} \rightarrow \ldots

This tree reconstructs the monitor stack and converts it into a tree-copying forwarding definitions if necessary. As a sample, we include the C translation of the ICMP monitor stack in Appendix A.4.

The translation of the main module executes instances as a depth-first traversal of this tree - first one capture module is executed, then its first child and so on. Detailed rules for the translation are provided in Appendix ??.

7.2.1 Implementation Status

We have written a compiler for NERL\_MOD in OCaml, and this compiler shares a lot of code with the NERL compiler. The compiler parses NERL\_MOD programs and type-checks it according to the typing rules. It then generates the \text{main} function as described above. We have used this compiler to generate most of the module stacks used in the case studies.

It must be noted, however, that translated NERL\_MOD programs are often quite simple and can be easily written by hand. In some cases, we find it easier to directly write and modify the main module in C. One advantage of programming it in NERL\_MOD is that we can then impose our typing restrictions. More importantly, we can control when recognizer instances are generated, and in what order they are executed. As we shall show for co-networked monitoring later in this chapter, this control over the execution sequence proves crucial.

7.3 Model-checking NERL

To model-check NERL programs, we translate them to Promela - the modeling language for the SPIN model-checker. There are several benefits to adding a model-checking component:

- We can check dynamic well-formedness properties of the recognizer. For instance, we can check whether a recognizer will ever violate the bounds of an array resulting in an \text{ArrBndErr} exception.
We can check that the NERL recognizer is correct with respect to the specification –

- it produces meta-events only if a correct protocol participant would, and
- it produces error-events only if the protocol specification is violated.

To see an example, for the ping recognizer we designed before, we would want to prove the following meta-event property.

**Property 7.1** If a ping sender process correctly sends an EchoRequest message to a ping receiver that correctly responds with an EchoReply message, then the ping recognizer produces an IsAlive event.

In addition, we would prove that the ping recognizer does not generate any false alarms.

**Property 7.2** If an ping sender process correctly sends an EchoRequest message to a ping receiver that correctly responds with an EchoReply message, then the ping recognizer does not produce any BadRequest or BadReply event.

To prove these properties, we first translate the NERL recognizer for ping to a Promela model. Then we model the ping sender and the ping receiver in Promela. The complete system is then handed over to SPIN along with the above properties expressed as Linear Temporal Logic formulae. The SPIN model-checker analyzes the system to find any executions that may violate the properties. The Promela models and LTL properties for have been included in Appendix A.5. To limit the state space, our ping sender chooses between 10 sequence numbers and sets an empty data field before sending the EchoRequest. SPIN found no errors in our ping recognizer. To check the first property SPIN analyzed 6299 states and 23780 transitions. For the second property, SPIN checked 6347 states and 8807 transitions. So, even for simple recognizers and properties, there may be several cases that need to be analyzed. It is useful to add an automated checking component that avoids the manual checking of each case.

### 7.3.1 NERL-to-Promela Translator

The Promela translation is similar to the C translation in many ways. One important difference is that there are no pointers in Promela, so data types must be translated to pointer-less types. Moreover, there are no floating-point or string values in Promela, so we abstract both these types by integers. For instance, doubles translate to:

\[
\|\text{double}\|_p = \text{struct}\{\text{int changed}; \text{int value}; \text{int oldvalue}\};
\]

Doubles are rounded to the nearest integer, and strings are represented by their lengths. So the Promela models represent abstractions of the NERL recognizer.

Arrays are translated to Promela arrays, and variable sized arrays are simply large arrays of size \(AMAX\) with an attached length attribute. The dynamic checks that we inserted in the NERL compiler are inserted as assert statements that can be checked by the SPIN model checker.

Event variables are represented by Promela channels and their attribute types by channel types:

\[
[\text{event } \tau X]_p = \text{chan } X = [1] \text{ of } [\tau]_p,
\]

Here, \(X\) is defined to be a channel with buffer size \([1]\), and values of type \([\tau]_p\) can be sent and received on the channel. So event generation translates to send operations on channels \((X!)\) and to check if an event has occurred, the channel must be polled \((X?![x])\). Because of restrictions in Promela syntax, the dynamic channel initialization in above translation is not entirely correct. Instead, we need to define table of channels in the beginning and index into them.

An event expression translates, as before, to a context and an attached function on Promela statements \(S\) - the function takes the promela statement and embeds it in new environment.
Here we first check if there is any input on the $X$ event channel; if there is, then we extract the buffered event attributes into $x$, and then execute $S$.

The recognizer control block is stored globally in an array indexed by the identity of the recognizer process. The recognizer itself is translated to a proctype or Promela process that runs concurrently with other recognizers; it waits for inputs on channels, modifies state and produces outputs on channels. Details of the translation are given in Appendix ??.

### 7.3.2 Implementation Status

We have implemented a preliminary NERL-to-Promela translator in OCaml - it shares the structure and several functions from the C translation. This translator was used to model-check recognizers in all the case studies. In particular, in the SMTP case study (Chapter 9), we used it to prove that several popular SMTP monitor specifications are incorrect.

The primary challenge for a good Promela translation is that the resulting state space should be small enough for SPIN to model-check effectively. Although NERL recognizers translate naturally to Promela, the translated models are often too large to be model-checked for interesting properties. In such cases, we need to modify the translated Promela model and carry out several abstractions in order to limit the state space. We believe that it should be possible to use properties of NERL recognizers to automatically carry out some of these abstractions, but no such strategies have been currently implemented.

### 7.4 Event Slicing

When we write a NERL recognizer for a protocol, we attempt to encode all the observable, safety properties that correspond to the specification and requirements. In addition, we attempt to reconstruct all the command and responses of the higher-layer so that we can place this recognizer in a monitoring stack. However, there are several cases when using this complete recognizer is not a good idea:

- We are interested only in reconstructing higher-layer events and not in checking for errors (for instance, this layer is known to be correct).
- We are interested only in a specific error, not all errors.
- We have found an error $E$, and we want to know the preceding events that could have caused $E$.
- We want to use different monitoring algorithms for different properties.

Event slicing is a technique that takes a set $O$ of output events and *slices* away all those transitions and event definitions that cannot affect events in $O$. It proves useful in all the cases mentioned above:

- The slice corresponding to higher-layer or error events efficiently generates only those events.
- The slice corresponding to error $E$ tells us which events and state transitions could have caused it.
The different slices corresponding to different events are all recognizers in their own right and may be executed independently.

The event slicing algorithm itself is a version of standard static program slicing [Tip95, BG96], with NERL event definitions and transitions considered as atomic statements. The causal dependencies between commands is represented by the $\prec_c$ relation. The corresponding slicing algorithm is shown in Table 7.1.

Table 7.1: NERL Static Event Slicing

<table>
<thead>
<tr>
<th>Long-term Causal Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>event $E v_1 \rightarrow X(x)$ WithAttributes $S_1$ $\prec_c$ transition $E v_2 \rightarrow S_2$ if $E v_2$ contains $X$</td>
</tr>
<tr>
<td>event $E v_1 \rightarrow X(x)$ WithAttributes $S_2$ event $E v_2 \rightarrow X(x)$ if $E v_2$ contains $X$</td>
</tr>
<tr>
<td>transition $E v_1 \rightarrow S_1$ $\prec_c$ transition $E v_2 \rightarrow S_2$ if $E v_2$ or $S_2$ reads $L$ or $L'$ and $S_1$ might modify $L'$</td>
</tr>
<tr>
<td>transition $E v_1 \rightarrow S_1$ $\prec_c$ transition $E v_2 \rightarrow S_2$ if $E v_2$ or $S_2$ reads $L$ or $L'$ and $S_1$ might modify $L'$</td>
</tr>
</tbody>
</table>

**Event Slice Algorithm**

$$\text{Slice}(\text{recognizer} \rightarrow \text{event} \rightarrow \text{recognizer})$$

Let $R = \{D_1; D_2; \ldots; D_m; \}$, $V_1; V_2; \ldots; V_n; \$ T_1; T_2; \ldots; T_o; \$ \text{EndRecognizer}$

$$\text{ESlice}(V_1; V_2; \ldots; V_n; T_1; T_2; \ldots; T_o)(X) \{$$

0. Input = $\{V_1; V_2; \ldots; V_n; T_1; T_2; \ldots; T_o; \}$; Output = $\{}$

1. For all $V_i$ in Input such that $V_i = \text{event } E v_1 \rightarrow X(x)$...

1.1 Remove $V_i$ from Input and add it to Output

2. Repeat until Output does not increase in size

2.1 For all $V_i \in$ Output

2.1.1 For all $V_j$ in Input such that $V_j \prec_c V_i$

2.1.1.1 Remove $V_j$ from Input and add it to Output

$$\text{ESlice}(V_1; V_2; \ldots; V_n; T_1; T_2; \ldots; T_o)(X) \}$$

### 7.4.1 Implementation Status

The slicing tool is implemented as a compile-time feature of the NERL compiler. For instance, to slice for the BadEchoReply output event in the ping recognizer, we invoke the compiler as follows

```
nerl -slice BadEchoReply ping.nerl > ping.c
```

The resulting C program only generates the BadEchoReply event, and does not generate IsAlive or BadEchoRequest. For the simple ping recognizer, the slice does not significantly reduce the
size of the program. However, in the TCP case study, we shall see how the slicing tool can be used to specialize the recognizer for specific properties. In addition, in the SMTP case study, we just need one portion of the TCP recognizer. So there again, we use the slicing tool to generate an efficient subset of the TCP recognizer.

7.5 Dynamic Event Tracing

Diagnostic tools are essential for a monitoring system. NERL events have attributes containing information that can be used to interpret them. 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 user providing error information. In this section, we describe a powerful diagnostic tool that can automatically supply diagnostic information for all NERL events.

NERL monitors are executed on packet traces that can contain several million packets. So when an error is discovered at time T, the user must go through the packet trace backward starting at T and attempt to find out why the error occurred. Clearly, it would be very useful if along with the error event we could produce the sequence of packets that caused it. This sequence is in effect a dynamic slice of the packet trace [Tip95, BG96]. The sequence is also the smallest sub-trace (counter-example) that can be extracted from the packet trace to cause the error.

We propose to add an event tracing ability to NERL programs. As the recognizer executes, every state variable keeps track of the primitive events that led to its current value. Then as each output event is produced, the recognizer also indicates the sequence of events that triggered it. This tracing algorithm is a version of forward dynamic slicing [KY94] and is similar to event trace slicing [SK00].

To implement event tracing we must modify the NERL compiler, and to a smaller extent the NERL MOD compiler. First, each packet that enters the system is given a unique integer identifier. Then, to each state variables and event we add a depends attribute that contains a sequence of identifiers corresponding to the packets that affected its value; olddepends records the sequence that affected the old value. For instance:

```plaintext
[double]i = struct { int changed; double value; double oldvalue;
                      IntSet depends; IntSet olddepends; }
```

There are two kinds of dependencies that we must keep track of: control dependencies and data dependencies. For instance, consider the if statement:

```plaintext
if (status == CLEAR) then {icmp_seq = e.icmp_seq; }
```

Here the statement after “then” is control-dependent on the variables in the “if” clause; icmp_seq is control-dependent on status. So all the packets that status depends on must be added to the depends attribute of icmp_seq. Event expressions that are used as guards for statements also generate similar control-dependencies. On the other hand, the new value in icmp_seq clearly depends on e.icmp_seq. This is a data-dependency, and again the packets that e.icmp_seq depends on must be added to icmp_seq->depends. The modified NERL compiler generates code by identifying control and data dependencies in this manner.

For instance, during variable assignment, if \( \Delta \) contains the control dependencies of the current statement and \( ([E]_\varGamma \; D \; \Gamma \; \Lambda) \) contains the dependencies of variables in \( E \), then the assignment is translated as follows:
Here the conditional checks whether the location has already been changed in this round. If not, then the current value and dependencies are backed up in oldvalue, olddepends. Then, the current value is modified. The last few statements add dependencies; for each variable \( v_i \) that \( L \) depends on, \( v_i \rightarrow \text{depends} \) is added to \( ([L]_c \rightarrow \text{depends}) \).

More details on the modifications to the NERL compiler for event tracing are listed in Appendix ???. The NERL\textsubscript{MOD} compiler does not have to be changed, except that when it copies output events to input events, it must also copy the dependency attributes.

### 7.5.1 Implementation Status

We have implemented the tracing feature as a modification of the NERL and NERL\textsubscript{MOD} compilers. The new compilers translate recognizers to C programs that have the tracing code enclosed in \#ifdef CTRACE \ldots \#endif. So to turn the tracing on, the C recognizer is compiled with the \texttt{-DCTRACE} directive.

When tracing is turned on, we find that the performance of the recognizer dips considerably. Since the number of packets that a variable can depend on increases with the length of the session, we find that tracing can be infeasible for long protocol sessions. The main reason is that we are tracing the dependencies of every variable in the program. We find that often in NERL programs, there is a group of variables that is always updated together. In such cases it is more efficient to have a single \texttt{depends} attribute for the entire group. Currently, the programmer must modify the generated C code to achieve this grouping, but it should be possible to modify the compiler to detect such groups automatically.

On the whole, we find that event tracing works quite well and produces invaluable diagnostic information. We shall see concrete examples of these in the case studies later in this thesis. To see an example of the output of a traced execution, when the ping recognizer is executed with tracing turned on, then the output trace looks as follows:

```
2: IsAlive Event <ip_src:158.130.012.004,ip_dst:158.130.012.217,  
icmp_id:13043>
2: Depends on:<1,2>
4: IsAlive Event <ip_src:158.130.012.004,ip_dst:158.130.012.217,  
icmp_id:13043>
4: Depends on:<3,4>
```

Here each line is prefixed with the current packet number, and the “Depends” clause shows the packets that the current output event depends on.

Sometimes, we are only interested in the data dependencies of an event - for instance, when we do not care about all the packets that went into setting up a session and only want to look at the
packets that contained data. For such cases, we have also implemented a modified slicing tool that maintains only data dependencies for variables. For instance, in the ping recognizer the IsAlive event contains icmp_data as an attribute, and this is copied from the EchoReply event. So since the IsAlive event does not have any data dependency on the earlier request, if we execute the NERL recognizer with only data tracing on (-DDTRACE), we get:

```
2: Depends on:<2>
4: Depends on:<4>
```

This tells us that the icmp_data attribute in the IsAlive at packet 2 only depends on the attributes of packet 2.

### 7.6 The Trace-search Algorithm

Having defined the language $L_M$ that a co-networked monitor must accept, we would like to describe strategies for programming such recognizers in NERL. Recall that the monitor will observe a trace $\tau$ of iq and od events. The device is not incorrect as long as $\tau \in L_M$: we can exhibit a sequence $\omega$ which is derived from $\tau$ by the addition of id and il tokens and which is consistent with the following set of conditions:

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

Suppose that $g$ is the colocated recognizer for $S$. So $g$ checks $S$ on a sequence $\alpha$ of id and od tokens and tells us whether the sequence is in $L_S$; We write the query to $g$ in the form $\alpha \in g$. One strategy for writing the co-networked recognizer for $S$ would be to transform $g$ automatically to a recognizer that takes buffering and loss into account. Such a transformation would provide a general technique and allow the NERL programmer to ignore all co-networked issues.

So our problem is to construct a function $F(g, \tau, m, l)$, that given a trace $\tau$ of iq and od events collected by a co-networked monitor and a co-located recognizer $g$, tells us whether the trace corresponds to some proper execution with respect to $S$. A non-deterministic algorithm for $F$ is straight-forward. Given a $\tau$, guess a sequence $\omega$ admissible with respect to $M(S, m, l)$, such that $[\omega]_{iq,od} = \tau$. If $[\omega]_{id,od}$ satisfies $g$, $\tau$ is OK. Otherwise, report failure.

A deterministic version of the above algorithm is brute-force search: simply construct all possible $\omega$’s from $\tau$, checking each admissible $\omega$ against $g$, until a match is found or all possibilities are exhausted. Additionally, we would like $F$ to be computable on-line, meaning that it should make only one pass over the input $\tau$. We give such a brute-force breadth-first-search algorithm in Table 7.2. We call this algorithm the trace-search algorithm: $BF(g, \tau, m, l)$.

In effect, the $BF$ algorithm models the input buffer and loss module to generate every admissible $\omega$ and checks it against $g$. The sequences it allows are exactly those in the co-networked monitoring language:

**Theorem 7.3** If $g$ is a co-located monitor for $S$, then $BF(g, \tau, m, l)$ produces an error if and only if $\tau \notin L_M(S, m, l)$.

This theorem follows from the following lemma:
**Data Type.** $\Omega$ is a set of triples (admissible string, input buffer, input loss streak)). Initially, $\Omega = \{ (\epsilon, \epsilon, 0) \}$

**Event Handlers.** On receiving

- $iq_x$:
  1. $\forall (\omega, b, s) \in \Omega$:
     - delete $(\omega, b, s)$ from $\Omega$;
     - check-add $(\omega :: iq_x, b :: iq_x, s)$ to $\Omega$;
  2. iterate until no more additions to $\Omega$:
     - $\forall (\omega, iq_y :: b, s) \in \Omega$:
       - check-add $(\omega :: id_y, b, 0)$ to $\Omega$;
       - check-add $(\omega :: il_y, b, s + 1)$ to $\Omega$

- $od_x$:
  1. $\forall (\omega, b, s) \in \Omega$:
     - delete $(\omega, b, s)$ from $\Omega$;
     - check-add $(\omega :: od_x, b, s)$ to $\Omega$.
  2. iterate until no more additions to $\Omega$:
     - $\forall (\omega, iq_y :: b, s) \in \Omega$:
       - check-add $(\omega :: id_y, b, 0)$ to $\Omega$;
       - check-add $(\omega :: il_y, b, s + 1)$ to $\Omega$

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

- $|b| \leq m$, and
- $|s| \leq l$, and
- $[\omega]_{id, od} \in g$

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

---

**Table 7.2:** Trace-search Monitoring Algorithm
Lemma 7.4 If g is a co-located monitor for S, then $BF(g, \tau, m, l)$ produces a set $\Omega$ containing exactly all the strings $\omega$ that satisfy the following:

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

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

C3 $\omega$ is admissible (with respect to $M(S, m, l)$).

Proof. We use induction on the length of $\tau$. For empty strings, the set $\Omega$ is empty and so the lemma is true. In the inductive step, $\Omega$ contains all strings that satisfy the conditions C1, C2, C3 up to the n’th token in $\tau$.

If the $n + 1$’th token in $\tau$ is $iq_a$, then the algorithm first adds this token to every $\omega$ and every simulated buffer b. This preserves the property C1. Then to generate all admissible sequences the algorithm allows $iq_a$ to be unbuffered and consumed by the device or lost. Finally, before adding any sequence to $\Omega$, it is checked against $g$ and against the buffer and loss restrictions. This preserves C2 and C3.

If the $n + 1$’th token is an output od, then since outputs are not buffered or lost it is added immediately to every $\omega$. This preserved C1. Then all possible buffer-loss sequences are generated. As before, sequences are checked against $g$ and for admissibility before adding them to $\Omega$. This preserved C2 and C3.

The main source of inefficiency in $BF(g, \tau, m, l)$ is in the possibly large number of plausible sequences that it maintains. As we mentioned before it may have to maintain $O(2^{|\tau|})$ such sequences. On the other hand, if $g$ is very strict, it may eliminate a large number of these sequences. In general, while the trace search algorithm is useful for autatically generating co-networked monitors from co-located monitors, it may be much more efficient to use the particular features of a co-located recognizer to manually transform it to run in a co-networked environment.

7.6.1 Implementation Status

The co-networked algorithm is implemented as a modification of the NERL_MOD algorithm. Currently, it supports a monitoring stack with a single recognizer running above the packet capture module. To use it, one must first annotate the input events in the recognizer signature with channel names as we have already described.

Given these channel annotations, the modified NERL_MOD compiler automatically generates code to execute the co-networked algorithm. First, instead of a single control block, instances now have a list of possible control blocks. Instance declarations are translated as:

\[
\text{[instance } R X \text{ WithAttributes } S]_{c} = \{R_{\text{entry}} \ast X_{.cb};\}
\]

Where $R_{\text{entry}}$ is defined for each recognizer:

```c
typedef struct {
    ITo_queue ibuf;
    ITo_streak is;
    Ping_Sig* cb;
    struct Ping_entry* next;
} Ping_entry;
```

Here, the Ping_entry list is essentially the set $\Omega$ in the monitoring algorithm: ibuf contains the buffer of inputs and is indicates the current the loss streak. Accordingly, the modified compiler treats input events (EchoRequest) as $iq$’s and inserts them in the input buffer ibuf of each instance. Output events (EchoReply) are treated as $od$ events and immediately fed to every instance.
The co-networked NERL\textsubscript{MOD} compiler has been used for TCP monitoring, but is still under development. Since the co-networked monitor must maintain several copies of every instance, memory-saving optimizations become very important. For now, we optimize the generated code directly in C. Moreover, the current implementation only supports one layer of instances in the monitor stack. This is because even monitoring a single layer in a co-networked environment is resource intensive, as we shall see in Chapter 10. To monitor multiple layers, we shall need more optimizations.
Chapter 8

AODV: Wireless Routing Protocol Simulations

An internetwork can be viewed as a bipartite graph consisting of nodes representing routers and networks, and edges representing interfaces. A host attached to a network sends a packet with a destination network address to a router on its network. This router cooperates with other routers to determine a path for moving the packet toward its destination. A routing protocol is an algorithm used by routers to determine such a path.

Ad-hoc On-demand Distance-vector Routing (AODV) is a new routing protocol that has been designed to operate on wireless networks of mobile nodes. AODV has been developed as part of the MANET working group at the IETF and is considered one of the leading contenders to become the standard routing protocol for mobile, ad-hoc networks. To evaluate and compare the performance of such routing protocols, the CMU Monarch group implemented versions of several wireless routing protocols in the network simulator NS and carried out several simulations [BMJ98]. Their simulation results were used to fine-tune protocol parameters, as well as argue the relative merits of different routing protocols. Because of the impact of such studies, it is important to know that the simulator implementations used did not have any errors.

In this chapter, we use NERL recognizers to analyze NS simulation traces for the AODV simulator implementation. The chief characteristics of this case study are as follows

- Network layer routing protocol
- Unbounded number of participants executing concurrently
- New, untested protocol
- Simulator implementation: Monarch/NS
- Monitoring environment: co-located “god’s eye” view

Since AODV is still a new protocol with only prototype implementations, previous work on AODV testing consists of simulation studies [BMJ98, LH98, YEG00]. We know of no previous work to test these implementations for correctness. However, protocol bundled with the network simulator, typically come with validation test-suites, so that modified versions of these protocols can be validated to preserve some properties. These tests compare the performance of a modified protocol with a pre-computed expected performance chart for the scenario.

Testing based only on performance measures is less than one would like for careful analysis of a protocol. For one, such an analysis may not be able to detect certain kinds of bugs in the simulator code that do not manifest themselves as performance problems: security-property violations

76
for example. Moreover, it is desirable to have more diagnostic support than just performance statistics for finding implementation flaws. In this chapter, we show how the logical analysis by recognizers written in NERL is supplemented by diagnostic tools and techniques to discover flaws in simulator implementations.

We have earlier introduced this simulation analysis framework, Verisim, in [BGK+02]. The previous implementation of Verisim was based on a monitoring language called MEDL [LKK+99]. In this chapter, we use NERL to realize Verisim. The Verisim framework is shown in Figure 8.1. The AODV protocol implementation, written in C++, is simulated by NS to produce a simulation trace text file. The AODV specification is used to write the NERL protocol event recognizer. The NERL recognizer is executed against the simulation trace and checks for any deviations from AODV specification and requirements.

Next, we describe the wireless routing protocol AODV. Then, we show how AODV properties are represented using NERL recognizers. We execute the recognizers on large simulation traces and describe the results, comparing them with previous results using MEDL. Finally, we describe our analysis techniques for dealing with large output traces, and mapping output error events (alarms) back to bugs in protocol code.

### 8.1 AODV Routing

This section describes the AODV routing protocol [Per97, PR99] that we use in our case study. The first part provides a short description of the protocol. The second part discusses some of its requirements—properties that are expected to hold in AODV implementations.

#### 8.1.1 AODV Protocol

The Ad Hoc On-Demand Distance Vector (AODV) routing protocol is used in packet radio networks. A packet radio network consists of a collection of mobile nodes whose link connectivity frequently changes due to the node movement. A mobile node is willing to forward packets between two nodes that are within its wireless range, but cannot directly talk to each other. AODV is a protocol that used such forwarding nodes to establish routes (communication paths) between nodes `on-demand` (that is, only when they are needed).

A route to a destination \(d\) contains the following fields:

- `next_hop_d`: Next node on a path to \(d\).
- `hop_cnt_d`: Distance from \(d\), measured in the number of nodes (hops) that need to be traversed to reach \(d\).
seq_no_d: Last recorded sequence number for d.

d: Remaining time before route expiration.

The purpose of sequence numbers is to track changes in topology. Each node maintains its own sequence number. It is incremented whenever the set of neighbors of the node changes. When a route is established, it is stamped with the current sequence number of its destination. As the topology changes, more recent routes will have larger sequence numbers. That way, nodes can distinguish between recent and obsolete routes.

When a node s wants to communicate with a destination d, it broadcasts a route request (RREQ) message to all of its neighbors. The message has the following format:

\[
RREQ(d, \text{hops\_to\_src}, \text{dest\_seq\_no}, s, \text{src\_seq\_no}).
\]

Argument hops_to_src determines the current distance from the node that initiated the route request. The initial RREQ has this field set to 0, and every subsequent node increments it by 1. Argument dest_seq_no specifies the least sequence number for a route to d that s is willing to accept (s usually uses its own seq_no_d for this purpose). Argument src_seq_no is the sequence number of the initiating node.

When a node t receives a RREQ, it first checks whether it has a route to d stamped with a sequence number at least as big as dest_seq_no. If it does not, it rebroadcasts the RREQ with incremented hops_to_src field. At the same time, t can use the received RREQ to set up a reverse route to s. This route would eventually be used to forward replies back to s. If t has a fresh enough route to d, it replies to s (unicast via the reverse route) with a route reply (RREP) message which has the following format:

\[
RREP(\text{hops\_to\_dest}, d, \text{dest\_seq\_no}, \text{route\_lifetime}).
\]

Arguments hops_to_dest, dest_seq_no, and route_lifetime are the corresponding attributes of t’s route to d. Similarly, if t is the destination itself (t = d), it replies with

\[
RREP(0, d, \text{big\_seq\_no}, \text{MY\_ROUTE\_TIMEOUT}).
\]

The value of big_seq_no needs to be at least as big as d’s own sequence number and at least as big as dest_seq_no from the request. Parameter MY_ROUTE_TIMEOUT is the default lifetime, locally configured at d. Every node that receives a RREP increments the value of the hops_to_dest field and forwards the packet along the reverse route to s. When a node receives a RREP for some destination d, it uses information from the packet to update its own route for d. If it already has a route to d, preference is given to the route with the bigger sequence number. If sequence numbers are the same, the shorter route is chosen. This rule is used both by s and by all of the intermediate forwarding nodes.

The above preference rule is important for propagating error messages. In addition to the routing table, each node s keeps track of the active neighbors for each destination d. This is the set of neighboring nodes that use s as their next_hop on the way to d. If s detects that its route to d is broken, it sends an unsolicited RREP message to all of its active neighbors for d. This message contains hops_to_dest = 255 (infinity), and its dest_seq_no is one more than the previous sequence number for that route. We call such an RREP message an RERR because it indicates a route error. Such artificially incremented sequence number forces the recipients to accept this ‘route’ and propagate it further upstream, all the way to the origin of the route.

8.1.2 AODV Properties

Our study focuses on analyzing correctness of AODV implementations. This can be studied from two angles: correctness with respect to the requirements (R) and conformance with respect to the standard (S).
A common requirement for a routing protocol is \textit{Loop Freedom}: Computed routes never contain loops. It turns out that in the case of AODV it suffices to prove a simple invariant in order to guarantee loop freedom [BOG02]. The loop freedom invariant is described in Table 8.1.

\begin{table}[h]
\centering
\caption{AODV Requirement: Loop Freedom}
\begin{tabular}{|l|}
\hline
\textbf{Loop Invariant:} Along every AODV route to a destination \(d\), pair \((-\text{seq}_d, \text{hop}_d\text{cnt}_d)\) strictly decreases in the lexicographic ordering. \\
\hline
\end{tabular}
\end{table}

The AODV protocol standard describes what kinds of events can occur and how nodes are supposed to handle them. Network protocols represent reactive systems, which means that every action is carried out in response to an event. Although the standard is written in natural language, one can typically extract the state machine that it is trying to express. For example, the state machine corresponding to an AODV process is shown in Appendix B.1. To monitor conformance with such a state machine, we convert it to a protocol event recognizer that gets triggered every time a network event of interest happens. The recognizer attempts to keep track of the state of the protocol, and checks that the events generated by the protocol are correct with respect to the state machine.

Table 8.2 shows some of the properties that test adherence to the AODV standard. These properties were generated from the state machine description in Appendix B.1. Notice how each property contains an event in its description (denoted by a phrase of the form \textit{when...} or \textit{if...}). We should point out that the set of standard properties listed in Table 8.2 is not complete—satisfying all of the properties still does not guarantee adherence to the standard. In particular, there are a number of properties about the timing of protocol events that our state machine, and consequently these properties, does not express.

\begin{table}[h]
\centering
\caption{Properties from the AODV Standard Specification}
\begin{tabular}{|l|l|}
\hline
\textbf{Property Name} & \textbf{Property Description} \\
\hline Monotone Sequence Numbers & A node’s own sequence number never decreases. \\
Destination Stops & When a packet (RREQ, RREP or data) reaches its destination, it should not be forwarded. \\
Correct Forwarding & If a packet addressed to \(d\) (RREP or data) is forwarded, it is forwarded along the best unexpired route to \(d\) seen so far. \\
Destination Reply & When the destination replies to a route request, the value of the \text{hops}_\text{to}_\text{dest} field of the reply should be 0. \\
Node Reply & When a node sends a route, it sends the best unexpired route seen so far. \\
RREQ Sequence Number & When a node initiates a route request for a destination \(d\), the requested sequence number should either be 0, or the last sequence number recorded for \(d\) (\text{seq}_d). \\
Detect Route Error & If a node detects a broken route, it should use \text{dest}_\text{seq}_\text{no} = 1 + (\text{its own}) \text{seq}_d\text{ in the unsolicited RREP.} \\
Forward Route Error & When a node forwards an unsolicited RREP, it should forward the same sequence number that it received. \\
\hline
\end{tabular}
\end{table}
8.2 AODV Recognizer in NERL

The AODV recognizer must keep track of the state of every router in the network. So, first we encode the AODV state machine (Appendix B.1) in NERL. The state of the recognizer consists of the routing tables at each node. The routes are represented by three arrays:

```plaintext
int seq_no[NODES][NODES];
int hop_cnt[NODES][NODES];
int next_hop[NODES][NODES];
```

The first index indicates the node and the second indicates the destination. For instance, `seq_no[i][j]` contains the current `seq_no` at node `i`.

We then define state transitions that modify the routing tables by to mimicking the AODV nodes being monitored. For instance, every time a better route is seen for destination `j` at node `i`, the routing table is modified. The state transition for this is as follows:

```plaintext
transition Pkt(p) & Recvbetter(rb) -> {
    seq_no[rb.at][rb.dst] = p.src_seq;
    if (p.src_hc == INFINITY) then {
        hop_cnt[rb.at][rb.dst] = INFINITY
    } else {
        hop_cnt[rb.at][rb.dst] = p.src_hc + 1;
    }
    next_hop[rb.at][rb.dst] = p.prev;
}
```

Here the packet `p` is received at `rb.at`, and it advertises a better route to the destination `rb.dst`. The statement on the right updates the sequence number to the advertised sequence number and increments the advertised hop count (bounded above by `INFINITY`). The new route points to whichever node sent the route - `p.prev`. The state machine is encoded using three such state transitions and some auxiliary events like `Recvbetter`.

We then translate the properties described in the last section in terms of NERL. Many of these properties simply forbid deviations from the state machine. To check these properties we define alarms: output events that are triggered whenever deviations from the state machine are detected. However, some properties, such as the Loop Invariant, express additional conditions on the values of state variables. We show how to cast failures of the Loop Invariant in terms of NERL alarms.

Consider three different nodes: `at`, `nxt` and `dst`. Assume that the node `at` has a route to `dst` through its neighbor `nxt`:

```
next_hop_dst(at) = nxt.
```

Let \((s(at), h(at))\) be the sequence number and the hop count that node `at` has for the destination `dst` (similarly \((s(nxt), h(nxt))\) for the node `nxt`). The Loop Invariant property says:

\[(s(at) \leq s(nxt)) \land (s(at) = s(nxt) \Rightarrow h(at) > h(nxt)).\]

Therefore, the property is violated exactly when the following holds:

\[(s(at) > s(nxt)) \lor (s(at) = s(nxt) \land h(at) \leq h(nxt)).\]

The following NERL event definition detects a violation of the Loop Invariant property by checking it every time a packet is seen from a node `at` to a destination `dst`. 

80
When the invariant is violated, we attach the current values in routing table as attributes to the error event.

Every time a better route is received, the recognizer changes the routing table and the loop invariant must be checked. Here, the invariant is being checked at the node rb.at, for the destination rb.dst. So nxt = next_hop[rb.at][rb.dst]. The NERL event, LoopInvFails, simply compares the routing tables at nodes Recvbetter.at and nxt to check if the loop invariant is being violated.

Appendix B.2 gives the complete NERL script for AODV. Table 8.3 shows the AODV properties and their corresponding NERL alarm names.

<table>
<thead>
<tr>
<th>Property</th>
<th>NERL alarm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monotone Sequence Numbers</td>
<td>NotSeqMono</td>
</tr>
<tr>
<td>Destination Stops</td>
<td>DestForwards</td>
</tr>
<tr>
<td>Correct Forwarding</td>
<td>BadFwd</td>
</tr>
<tr>
<td>Destination Reply</td>
<td>BadDestRep</td>
</tr>
<tr>
<td>Node Reply</td>
<td>BadNodeRep</td>
</tr>
<tr>
<td>RREQ Sequence Number</td>
<td>BadRReq</td>
</tr>
<tr>
<td>Detect Route Error</td>
<td>BadRerr</td>
</tr>
<tr>
<td>Forward Route Error</td>
<td>BadRerrFwd</td>
</tr>
<tr>
<td>Loop Invariant</td>
<td>LoopInvFails</td>
</tr>
</tbody>
</table>

8.2.1 Correctness

The NERL type checker was a great help in avoiding simple mistakes while programming these properties. In fact, it helped us find errors that we had missed in the earlier recognizer for AODV that we had written in MEDL.
Once the NERL recognizer is written, we would like to check if it accurately captures the AODV standard. To this end, we translate the recognizer into Promela for model-checking in SPIN. For an earlier verification project [BOG02], we wrote a Promela model for an AODV network. We compose the translated recognizer with this AODV network in a way that it can see all the packets exchanged between all nodes. Then we check that the monitor is correct - the recognizer does not raise any suprfluous errors:

Property 8.1 In a network of correct AODV nodes, there is no execution that results in the recognizer producing an error event.

This property ensures that there will be no false-positives generated by the recognizer. In addition, we would also like to check whether the monitor is maintaining the correct routing tables:

Property 8.2 In a network of correct AODV nodes, the routing table at node $i$ is identical to the routing table for $i$ reconstructed at the recognizer.

Model-checking large AODV networks is infeasible because of state space considerations. So we verify the translated recognizer for these properties in a 5 node network with one destination.

8.3 Monitoring Environment: Simulation Traces

We consider an implementation of AODV written by the CMU Monarch Project (http://monarch.cs.cmu.edu) for the network simulator NS. This code was used primarily for performance analysis of AODV in comparison with other routing protocols for mobile, ad hoc networks [BMJ+98]. In order to carry out this comparison, a number of large random scenarios were constructed as well.

The Monarch implementation is based on the first version of AODV [Per97], and is known to have bugs—because of incomplete specification in the standard, and due to programmer errors. The code is already instrumented to produce a packet trace for every packet generated, forwarded and dropped by the protocol.

We use NERL to analyze NS simulations of this code. To carry out the simulation, we must specify the simulation scenario. First, we define the network topology - how many nodes are there and how are they connected. Then we define the traffic model - which nodes will send data to which destinations. The routing protocol will then attempt to find routes between these sources and destinations. Finally, we define the parameters that the AODV implementation must use - timeout values for instance.

We first carry out the simulation for a small network scenario $S$ with 5 nodes, as shown in Figure 8.2.

Topology: There are 5 nodes initially arranged as in Figure 8.2 (Phase I). Then node 5 starts moving away from the network, causing the wireless links to break after 2.5s (Phase II). 30s into the simulation, node 5 heads back towards node 1. At 55s it is within the range of node 4 (Phase III), at 70s it is in the range of nodes 2,3, and 4 and finally it is in the range of 1,2, and 3 (Phase IV).

Traffic Model: Nodes 1,2 and 3 are constant bit rate (CBR) sources for node 5. They send a total of 1000 packets of size 512 bytes each, one packet every 0.1s.

AODV parameters: We use the optimal AODV configuration computed by the Monarch group. The configuration involves parameters like route timeout intervals and the number of times a request should be re-tryed.
When the AODV protocol is simulated on scenario $S$, NS generates a trace $T$. The initial fragment of a typical trace is shown in Table 8.4. When a packet send or receive event happens at a node $N$, there is a line in the trace with the format:

\texttt{<send/recv> <time> \_N\_ RTR --- <Link Layer info> ------- <IP info> <AODV info>}

For instance, the third line of the trace tells us that at time 0.0, node 3 broadcast an AODV REQUEST, for destination 5, with hop count 0 and broadcast id 1. Moreover, node 3’s current sequence number is 1, and the last sequence number it heard from the destination (5) is 0. This request eventually reaches the destination 5, through node 4. The last line of the trace is node 5’s REPLY to the request which it unicasts to node 3, via node 4.

### 8.4 Analysis

We then execute the NERL recognizer on such a trace ($T$); the recognizer checks if $T$ satisfies the AODV properties $\phi$, and produces a meta-trace $T^\phi$ of any property violations (error events) it finds. Each error event probably indicates a bug in the protocol code.

We executed NERL on a number of traces for the simple 5-node scenario, and found several errors. Table 8.5 shows statistics for the alarms produced by NERL for one of these simulation traces, with 49542 events.

#### 8.4.1 Destination Advertises Incorrect Hop-count

We note that most of the alarms are of the same kind (BadFwd), but that the first alarm in the produced meta-trace is actually a BadDestRep error event at packet number 19:
Table 8.4: AODV Simulation Trace

<table>
<thead>
<tr>
<th>Pkt No: 19</th>
<th>Event: 1, Pkt: 2, Atnode: 5, Fordest: 5, Src: 5, SrcSeq: 2, SrcHops: 1, Dest: 3, DestSeq: 0, Bcastid: 0, Prev: 0, NextHop: 4, TTL: 32, Life: 600</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.003450:</td>
<td>Event BadDestRep has occured - BadDestRep: (at: 5 dst: 5)</td>
</tr>
<tr>
<td>0.003450:</td>
<td>Depends on: &lt;19&gt;</td>
</tr>
</tbody>
</table>

Here in addition to the error event, we have also printed out the fields of the AODV packet for illustration. The BadDestRep error indicates that the destination sent an incorrect RREP. Indeed, when we look at the fields we see that SrcHops is set to 1, when it should be set to 0.

Having found this error, we have two options:

- we can inspect the simulator code for AODV and correct this error, and generate a new trace,
- or
- we can attempt to find more errors in the same trace.

We call the first approach repair-first-bug (RFB). Clearly, the AODV code for the destination is not implemented correctly. If it is implementing a completely different state machine from what we expect, then we cannot hope to analyze the rest of the trace. In that case, RFB is the only realistic approach, even though it implies running simulations again and again.

On the other hand, maybe we can simply guess the bug in the code. In all 3 BadDestRep error events, we find that the SrcHops field is consistently 1. So maybe, there is a simple off-by-one bug in the RREP generation code. To confirm this guess, we modify the BadDestRep event to check whether a destination ever emits a hop-count other than 1 (instead of checking for 0). Lets call this new NERL program $\phi_1$. When we run this recognizer on the same simulation trace, all

Table 8.5: AODV Property Violations Detected by NERL

<table>
<thead>
<tr>
<th>Meta-trace</th>
<th>BadDestRep</th>
<th>BadFwd</th>
<th>BadNodeRep</th>
<th>BadRerr</th>
<th>LoopInvFails</th>
<th>Total alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0^0$</td>
<td>3</td>
<td>2807</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>2830</td>
</tr>
<tr>
<td>$T_0^1$</td>
<td>0</td>
<td>2807</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>2827</td>
</tr>
<tr>
<td>$T_0^2$</td>
<td>0</td>
<td>16</td>
<td>9</td>
<td>4</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td>$T_0^3$</td>
<td>0</td>
<td>16</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
</tbody>
</table>

84
the BadDestRep events disappear. This validates our guess and identifies the first bug in the code. Now we can use this modified recognizer to look for more errors in the trace.

We call this process of modifying the specification to conform to the implementation tuning. The idea of tuning is that if we can guess what the implementation is doing and modify the recognizer accordingly, there should be no error events. Often, one has to try a few guesses before hitting the right answer.

8.4.2 Nodes Ignore Hop-count in Routes

The remaining errors are shown in the second row of Table 8.5. We find that most of the errors are BadFwd, and moreover all of these are at the same node 3 for the same destination 5. The first instance of this error is at:

```
Pkt No: 82
Event:1, Pkt:0, Atnode:3, Fordest:5, Src:3, SrcSeq:0, SrcHops:0, Dest:5, DestSeq:0, Bcastid:0, Prev:0, NextHop:1, TTL:32, Life:0
0.398146: Event BadFwd has occurred - BadFwd:{at:3 dst:5}
0.398146: Depends on: <80,81,82>
```

This error indicates that the node 3 is using node 1 as its next hop towards 5 when it should use some other node. The “depends on” clause indicates that we should also look at the events in packets 80 and 81. Note that since there are already 80 packets in the trace, this diagnostic significantly reduces the events we need to look at.

Packets 80 and 81 contain two RREPs, the first from node 1 with hop count 4 and the next from node 2 with hop count 3. According to the AODV state machine, node 3 should pick the second route because it is shorter. So 3 should be forwarding packets through node 2, not node 1.

After a couple of wrong turns, we guessed that maybe the AODV implementation ignores hop counts and only looks at the sequence number of a route. So we tuned the recognizer by modifying the Recvbetter event, instructing it to ignore hop-counts in routes. We name this thirs version of the recognizer \( \phi_2 \). On running \( \phi_2 \) on the simulation trace \( T \), we find that most of the BadFwd errors disappear, validating our guess. Note that the LoopInvFails error disappeared as well. Error events are often inter-related in this way, the condition causing the LoopInvFails event was related to the same bug that caused the BadFwd event, so when we tune one error out, the other disappears as well.

8.4.3 Node Sends Incorrect RERR

Of remaining errors in the meta-trace (third row of Table 8.5 the first is a BadREr event at packet 3592:

```
Pkt No: 3592
Event:1, Pkt:2, Atnode:4, Fordest:5, Src:5, SrcSeq:3, SrcHops:255, Dest:2, DestSeq:0, Bcastid:0, Prev:0, NextHop:0, TTL:1, Life:0
49.593437: Event BadRerr has occurred - BadRerr:{at:4 dst:5}
49.593437: Depends on: <27,3592>
```

Node 4 is sending an RERR packet to node 2 saying that node 4 can no longer reach node 5. To see why this packet caused an error, we must look at packet 27 as indicated by the depends clause.
At Packet 27 destination 5 sent an RREP to node 4 and gave it a route (seq.no = 3, hop.cnt = 1). Node 4 retains this route all the way to packet no 3592 when it decides that the route is no longer available and sends an RERR. However, the AODV standard says that the RERR should have a sequence number that is one more than the previous sequence number - in this case 4. Instead, the RERR at packet 3592 has seq.no = 3, and this is what caused the error.

Clearly, the implementation of RERR packets is faulty. We guess that it always uses the old sequence number instead of incrementing it, and tune the recognizer to produce $\phi_3$. Indeed, the fourth row of Table 8.5 indicates that all BadRerr error events disappear.

8.4.4 Nodes Send Expired Routes

At this point there are only 25 errors remaining, and we can manually inspect all of them to check what errors they indicate. Carrying on with the analysis, we find a few more bugs in the code.

The remaining BadFwd events are triggered when nodes send RERR packets to node 0 instead of sending them to a valid address. This may indicate a bug in the implementation, or simply that some RERR features have not been implemented.

Finally the BadNodeRep errors are a by-product of the BadRerr errors we saw earlier. Since nodes do not update their tables correctly when they send an RERR, they end up in a state where they reply to requests event though they have no route. When we tune the recognizer to simulate these errors, all the error events disappear. We have analyzed several simulation traces in this manner, and these bugs are all we found.

8.5 Fault Origin Adjudication

In the previous section, when we first analyzed the simulation trace we generated both the LoopInvFails error - indicating a violation of the requirements - and a group of other error events indicating violations of the standard. So we say that the trace $T$ falls in the category C in the FOA table (Table 3.1 introduced in Chapter 7).

Recall that a trace in category C is one that violates both the standard and requirements. It indicates that the implementation is incorrect and must be fixed. We generated several other traces, and analyzed them in the same way as we analyzed $T$. All traces generates by the simulator implementation violated the standard because of the bugs we described, although not all produced the LoopInvFails event. So these traces were either in category D or C. However, after we fixed the bugs, we ran several simulations, and the resulting traces did not generate any error events (category E).

We then noticed that the Monarch AODV implementation never deletes routes even though the standard allows deletion after a configurable BAD_LINK_LIFETIME timeout. So we modified the AODV code to allow this deletion and reran the simulations. This time, we found a trace that generated only LoopInvFails events and no other errors. For instance, in the 5-node network, the following error was produced.

```
Pkt No: 3672
    Event:1, Pkt:2, Atnode:1, Fordest:5, Src:5, SrcSeq:3, SrcHops:4, Dest:1, DestSeq:0, Bcastid:0, Prev:1, NextHop:3, TTL:32, Life:0
53.142582: Event LoopInvFails has occured -
    LoopInvFails:{node:1 next:2 dest:5 seq_at_node:3 hc_at_node:4 seq_at_next:0 hc_at_next:255}
53.142582: Depends on: <3594,3598,3672>
```

---
This error event indicates that node 1 thinks it has a route to node 5 through node 2, but the routing table entries at nodes 1 and 2 fail the loop invariant - the sequence number at node 2 is 0. On tracing back the event sequence indicated in the depends clause, we find that node 2 prematurely deleted its route to node 5, and the error message did not get through to node 1. Now node 2 is in danger of accepting a route from node 1 - causing a routing loop between 1 and 2. Indeed later in the trace a loop is formed.

This trace was in category A - it violated the requirements but conformed to the standard - and indicated an error in the AODV standard. When routes get deleted too early, even a standard-conformant implementation of AODV may form routing loops.

We had found this bug in the standard earlier using protocol verification techniques [BOG02]. However, while the verification effort demonstrated an error in an abstract model of the AODV standard, this FOA case study demonstrates that, under some conditions, a real implementation of the standard may also realize the bug.

We have since informed the protocol authors of the error and our recommendations have been included in the current version of AODV.

### 8.6 Scalability

In order to see how well our techniques scale up to the large simulations typically carried out for performance measurements, we applied our techniques to the largest trace made available by the CMU Monarch group [BMJ 98]. This ‘Off-The-Shelf’ (OTS) trace was generated by AODV simulation on a site of size $1500 \times 300$ meters with 50 nodes constantly moving at 20 meters per second. There were 150 data connections transmitting four 64 byte packets every second for a total of 1000 seconds. The simulation and our NERL analyses of the trace were carried out on a dual Pentium-III 550Mhz 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, and the simulation itself required about 5220 seconds to complete and generated 6,446,316 events.

We have earlier tried to analyze this trace using MEDL [BGK 02], which implements event recognizers in Java. A naive effort to use MEDL to check Monotone Sequence Numbers, a relatively simple property, on this trace was prohibitively time-consuming. We estimate that the time required to check this property, at each node and for each destination (2500 relations), after each of the 6,446,316 input events, is more than 100 days based on extrapolating a 4-day run of the MEDL analysis. After using a number of optimizations, such as using `grep` to specialize the trace to only 5 nodes (25 relations), we could bring the analysis time for Monotone Sequence Numbers down to 51 seconds. We failed to carry out the analysis for all 50 nodes together, but we estimate that we could have carried out several 5-node analyses to check Monotone Sequence Numbers for the complete trace in around 50,000 seconds. This kind of piecemeal analysis would not, however, work for more complex properties, such as Loop Freedom, where more than 5 nodes may interact to cause an output event.

We re-ran NERL on this trace, and we could process the complete trace for all 50 nodes, for all the AODV properties in 675 seconds. We found instances of all errors except for DestForwards. There were 708727 unique errors: 402613 BadFwds, 225305 BadNodeReps, 61528 BadRReqs, 9316 BadDestReps, 7868 BadRerrs, 1910 LoopInvFails, 154 BadRerrFwds, and 33 NotSeqMonos. These errors were due to the various bugs that we have already described.

Our NERL analysis time for this large simulation trace indicates a performance improvement of several orders of magnitude over MEDL. We can think of several reasons for this remarkably better performance.

1. MEDL was not designed for networking applications - it lacks arrays and event attributes. Encoding these in MEDL led to a big performance hit, because it caused every event to be encoded as around 2500 independent events.
2. MEDL was designed to monitor Java programs. As a result it was implemented in Java, and expected input events as Java objects passed using a socket. We were instead trying to use MEDL for speedy text analysis, an application for which it was not optimized.

3. While the performance gap between Java and C might be narrowing, we believe that our choice of C as an execution language made a big difference in the relative speeds of the recognizers.

8.7 Results

Errors. In total, we have detected at least 5 separate bugs in the AODV implementation:

1. When a destination sends a route, it sends a hop count of 1 for itself (instead of 0).
2. A node does not accept a new route if it has the same sequence number as the current route (it accepts only higher sequence numbers).
3. A node that detects a route breakage, forgets to increment the sequence number for the route.
4. A node that detects a route breakage does not send the route error to the correct destination (it sends it to 0)
5. A node that has an expired route that has not yet been deleted replied to incoming route requests with this route (instead of acknowledging that it does not have a route).

The combination of these bugs produces several strange conditions in a long trace, including routing loops. The first 3 of these bugs were also found through our earlier analysis using MEDL. We confirmed the existence of these bugs in the Monarch implementation, and informed the developers. These bugs have been fixed since.

In addition, we identified one error in the AODV standard using the fault origin adjudication technique.

- When a node that has an expired route deletes the route too soon, a routing loop may be formed.

This error has also been acknowledged and fixed in the latest version of the AODV standard.

Performance. We found that our NERL recognizer was quite efficient and scalable. The recognizer monitored 50 routing nodes at the same time and could process packets at 9550 packets per second. The time taken for the analysis, 675 seconds, was small compared to both the simulation time, 5220 seconds, and the “real time” in the simulated network, 1000 seconds. This indicates that our NERL recognizer for AODV is probably fast enough to monitor live AODV networks.

Analysis Techniques. This case study demonstrates the value of the tuning technique. Every run of the NERL recognizer took only a few seconds, much less compared to the simulation time, which was several minutes even for a simple scenario. Repair-first-bug would have needed to run the simulation at least 5 times, in addition to the time required to find and correct the bug in the C++ AODV implementation. Instead, we modified our readable NERL specification and reran the test in seconds.

Diagnostics. We also used the event tracing feature extensively to track the meaning of error events. As we saw in the BadRerr event, even for a simple 5-node network, two events that are related might be more than 3000 packets apart. Alarm tracing allows us to ignore all intermediate packets.
Chapter 9

SMTP: Internet Mail Forwarding

Email has for many years been one of the most prevalent services on the Internet. As a result, the Internet Mail Architecture has been closely studied and quite heavily engineered. To use email, a sender writes a message, addresses it to a recipient, and hands it over to a Mail Transport Agent (MTA) such as Sendmail. Once an email has thus entered the mail system, the system becomes responsible for delivering the message to the recipient or returning an error message (via email) to the sender. Senders and recipients are users (or administrators) of mail servers, and are often represented by email addresses of the form user@domain, where domain may be a mail server anywhere in the Internet.

The actual transfer of email across the Internet is carried out by the MTAs. An MTA is given a message, and an envelope that contains the sender and recipient email addresses, say S@domainA and R@domainB. The MTA then attempts to deliver the message to the MTA at domainB. If there is no direct way to contact the MTA at domainB, the message may be delivered to an intermediate relay server that would later forward the email to its destination.

The protocol that runs between MTAs in order to carry out the transfer of email is called the Simple Mail Transport Protocol (SMTP) [Pos82, Kle01]. In this chapter, we shall monitor SMTP servers using NERL recognizers and attempt to find errors in their behavior.

The chief characteristics of this case study are as follows

• Application layer protocol: runs on top of TCP
• Two participants - client and server
• Highly used, established protocol
• Popular open-source implementations: Sendmail, Postfix, Exim
• Monitoring environment: bottleneck, reliable message streams

Previous work on SMTP implementation testing consists mainly of bug-hunting by users in the field. Published reports on implementation errors primarily involve security-related bugs that may allow remote attackers to take control of the mail server. We know of no earlier study that systematically tests SMTP implementations for conformance with the standard specification.

In the next section, we describe the SMTP protocol. Then, in Section 9.2 we describe the monitoring environment provided by SMTP’s position in the Internet protocol stack. We show that since SMTP runs on top of TCP, we can correctly reconstruct SMTP commands and responses by writing recognizers for the IP Fragmentation and TCP Segmentation layers. In Section 9.4,
we show how to write the SMTP recognizer in NERL and discuss correctness issues. Finally, we analyze live executions of SMTP servers and present results in Section 9.5.

9.1 Simple Mail Transport Protocol

SMTP uses a TCP session between two 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 9.1 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.

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.

| 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 9.1: Sample SMTP Session

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.

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 [MR96], and the Internet Message Access Protocol [Cri96] 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.

9.1.1 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 [Cro82, Res01]. 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 9.1.

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. 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).

9.2 Monitoring Environment: Reliable Message Stream

To monitor emails as they are transferred between MTAs we must first reconstruct SMTP commands and responses. This process of layer-wise reconstruction is depicted in Figure 9.2

---

2[Res01] describes a number of other fields that may contain addressing information as well.
The key idea in reconstructing events at each layer is that the receiver of the packets, segments, commands will have to carry out this reconstruction anyway. So all we need to do is to mimic the actions of the receiver. Of course, we will have to assume that all layers below SMTP are operating correctly, so the sender and receiver behave in a predictable manner and can be mimicked.

Another important observation is that the command-response trace that we reconstruct is a true trace, even if the packet trace on the wire was not. This is because TCP smooths out any packet loss, re-ordering, and so the sequence of events as reconstructed by TCP is guaranteed to be correct. Again, note that we are assuming that the TCP sender and receiver are indeed working correctly and implement a reliable message stream.

Now we address each layer in order, showing how to write recognizers to reconstruct the SMTP command-response stream.

### 9.2.1 IP Fragment Reassembly

When IP packets must be forwarded over a link whose frame capacity is less than the size of the packet, the packet must be broken down into smaller fragments and sent across the link. These fragments travel separately all the way to the destination where they are reassembled. This process of IP fragmentation and reassembly is considered integral to the Internet Protocol [Ins81a, Bra89b].

The IP fragments that are part of the same packet contain a common ID field, and an Offset field that indicates their position in the packet. All fragments except the last one must have the More Fragments flag set, and their length must be a multiple of 64 bytes. So to reconstruct IP fragments, the receiver starts a reassembly process for every new ID, and simply copies every fragment into its correct position in a packet buffer. When the packet is complete, it sends to reconstructed packet to the higher layers.
To mimic this behavior, we write an **IP_Reassemb** recognizer in NERL that takes **IP_Frag** input events and outputs complete **IP_Pkt** events. One instance of **IP_Reassemb** is generated for every new \( \langle \text{ip}_{-} \text{src}, \text{ip}_{-} \text{dst}, \text{ip}_{-} \text{id}, \text{ip}_{-} \text{p} \rangle \) tuple seen on a fragmented packet. The reassembly operation itself is encoded according to a simple algorithm. More sophisticated algorithms have been developed for fast reassembly in IP implementations [Bra89a].

The key fragment of NERL code is shown in Appendix C.1. When a fragment is received, it is first inserted into the packet buffer \( \text{ipdatabuf} \). The range of bytes it occupies is marked in \( \text{ipdatarecd} \). If it is the last fragment, then we know what the length of the entire packet \( \text{ip}_{-} \text{plen} \) must be. Finally, if we have received the entire packet then we flag the **PktDone** event which contains the reassembled packet.

**Fragmentation Errors**

The reassembly procedure described above will work if the fragmentation is carried out correctly. However, it is possible that there are errors in the fragmentation code that lead to ambiguities:

- Two fragments with the same offset and length have different data.
- Two fragments have byte ranges that overlap but are not the same.

When such errors are seen, it is difficult to guess what the receiving IP will do. For instance, it could simply overwrite the older fragment, or it could reject the newer fragment. Clearly, the **IP_Reassemb** recognizer can no longer consistently reconstruct the IP packet that reaches the receiver. So when such errors are seen, the recognizer flags the observed error as **FragmentClashes** and gives up on the entire packet. Since each fragmented packet is reassembled by a different instance of the recognizer, the other instances can go on. However, if this packet is not retransmitted, then this reconstruction failure may lead to the recognizer stack giving up on the entire stream of which this packet was a part.

For the purposes of this chapter, we shall assume that such fragmentation errors do not occur in an SMTP trace. If they do, then we stop analyzing the trace.

**Recognizer Correctness**

Having written the NERL recognizer for reassembling IP fragments, we use the type-checker to find simple errors before translating the recognizer to an executable monitor.

In addition, we want to ensure that the recognizer is correct, which means that it satisfies the following property

**Property 9.1** If an IP sender process correctly fragments an IP packet and sends the fragments to an IP receiver over a channel that may lose or re-order the packets, and if the receiver reconstructs the packet, then the monitor reconstructs the same packet.

To show this property, we encode the IP sender, channel and receiver in Promela. We then translate the NERL **IP_Reassemb** recognizer into Promela as well, and it listens to all events on the channel. We then use the model-checker SPIN to show that for all possible executions of the system, the property never fails.

For the purpose of model-checking we use a simple sender and receiver. The sender simply breaks one IP packet into 10 different-sized fragments with blank data and sends it to the receiver which reconstructs it. The different executions of the system are provided by the non-deterministic channel which may lose or re-order fragments. We ignore the data and only vary the sizes of the fragments. Spin model-checks the system and finds no errors in our recognizer.
9.2.2 TCP Stream Reassembly

We shall describe the TCP protocol in more detail in Chapter 10. Here we only describe its aspects that are important for data reconstruction. When a sender hands some data to TCP to send to the receiver, the data is broken into segments: each segment has a sequence number indicating its offset from the beginning of the data and a portion of the data. Each segment is sent in one IP packet. When the receiver gets a sequence of segments, it puts them together in order using the sequence number information much like IP reassembly. A significant difference is that TCP is reliable, which means that the receiver indicates to the sender the portion of data it has received so far. So if a segment is lost on the channel, the sender can detect this and retransmit the segment until the receiver acknowledges receipt.

Reliability means that the sender of data knows how much of the data has definitely reached the receiver. We use this property in designing our monitor. This time our NERL recognizer mimics the sender of data, when we see segments we place them into a send buffer. When the receiver acknowledges some of them, then we remove the acknowledged data from the buffer and produce a TCP_Data event indicating that a piece of data was sent and received.

The key fragment of the TCP_Reassembl recognizer is shown in Appendix C.2. A new instance of the recognizer is started for every TCP session, and the recognizer reconstructs the data sent in both directions (to and from). When a data packet is seen in the to direction, it is placed in the to.segments buffer. We maintain the buffer sorted by sequence number. When an acknowledgment packet is seen in the reverse (from) direction, the ToData event is produced containing all the segments in the buffer that have been acknowledged. These segments are then removed from the buffer.

TCP Errors

Like in IP fragmentation, there are several errors that could make TCP reconstruction difficult:

- Two data segments with the same sequence number have different data
- Two data segments contain clashing sequence number ranges
- The receiver acknowledges data that was never sent

In addition, if the monitor fails to capture all packets sent that reach the receiver, then it no longer reliably knows what data was received.

In such cases, when the recognizer finds that it cannot correctly reconstruct the TCP_Data event, it flags an error event and gives up on the entire stream. Since each stream has a different recognizer, the reconstruction of other streams will go on. However, all higher layer recognizers depending on this stream reconstruction will also have to give up.

Recognizer Correctness

As long as the TCP sender and receiver operate correctly, we want our monitor to obey the following property:

**Property 9.2.** *If a TCP sender process correctly sends data to an IP receiver over a channel that may lose or re-order the packets, then the monitor reconstructs the data D if and only if the receiver reconstructs the same data D.*

To prove this property, we encode a TCP sender, channel and receiver in Promela. We translate the NERL TCP_Reassembl recognizer to Promela and use the SPIN model-checker to prove that the property is never violated. Our model-checking instance had a sender that sent 10 segments of varying length and blank data to the receiver, with the channel losing and re-ordering packets. SPIN found no violations of the property for our recognizer.
Having written recognizers for IP fragments and TCP streams, we can now set up the entire monitoring stack for SMTP executions (Figure 9.3). In the figure, the boxes depict instances of recognizers, while the arrows depict the input and output events. Error events are not shown. The various layers are as follows:

- The packet capture layer sniffs packets from the wire and produces IP_Pkt or IP_Frag events.
- The IP_Reassemb layer takes IP_Frag fragment events and reconstructs IP_Pkt packet events. For every new fragmented packet, a new instance of this recognizer is generated.
- The TCP_Parse layer looks in the IP payload of IP_Pkts and parses the TCP header if present. For those packets that contain TCP payloads, it produces equivalent TCP_Pkt events that contain more header fields. This recognizer is currently written in C since it must parse the low level packet header. Alternatively, the recognizer could be described in PacketTypes or ASN.1 and the parsing code automatically generated.
- TCP allows several streams to share the IP channel, so for each new stream a new TCP_Reassemb instance is generated that takes TCP_Pkt events and reconstructs TCP_Data events.
• Each SMTP session is embedded in one TCP stream. TCP_Data events corresponding to SMTP sessions (TCP port 25) are sent for SMTP parsing. SMTP commands and responses are written as clear text according to a grammar described in the standard [Pos82, Kle01]. We write the SMTP_Parse recognizer in Lex and generate C code for the parsing. The recognizer takes TCP_Data events and parses them into Command and Response events.

• The SMTP_Recog recognizer follows the Commands and Responses in an SMTP session and checks it for correctness with respect to the standard specification. If the session is correct and transfers an email, the recognizer produces a reconstructs EmailAccepted event.

• The IMH_Recog layer takes EmailAccepted events and parses the Internet message headers from it for presentation, and checks it for errors. This parser is also written in Lex, since the grammar specified in the standard [Cro82, Res01] is quite involved.

Once all the individual recognizer are written, it is straightforward to write a NERL_MOD program that models this stack.

9.4 The SMTP Recognizer in NERL

The complete SMTP server state machine for the minimal command set is shown in Figure 9.4. 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 A is received at the server from the client. B! indicates that B is sent from server to client. OK represents a positive server response (server accepts command), while ERR represents negative responses (server rejects command). Each OK and ERR response may span several lines and must be suitably parsed.

In addition, there are implicit transitions that allow the server to issue VRFY, EXPN, NOOP and HELP commands in any state other than DATAREAD and get a response without changing the state. These commands can be thought of as queries that the client makes of the server.

To write the SMTP recognizer, we simply translate this state machine into NERL states and state transitions. When an event occurs for which no valid transition exists, we flag an error event. When a mail envelope or a complete email is seen, the recognizer generates meta events for higher layer analysis.

We use a variable status to store the server state. In the state diagram, the server states are given names, while the intermediate states when the server has received an input and is going to produce an output are left unnamed. For the recognizer, we need to name these intermediate states as well; we use a variable respstatus that can have the value RESPONSENONE, RESPONSEOK or RESPONSEDONE, when the server has not produced a response, has produced a partial response, or has produced a complete response respectively. When a command is received in a particular server state, the status variable is changed to the next state and respstatus is set to RESPONSENONE. The recognizer then waits for the server response and then commits the state transition or rolls back to laststatus.

For instance, consider the MAIL command. Its transitions get converted to the NERL code in Table 9.4. Here the state transitions are as shown in Figure 9.4. When a MAIL is recd in the correct state - HELORECD - the state transition m is executed. The sender email is stored in a state variable. If MAIL is recd in an incorrect state a BadCommand error event is flagged. If the response to this MAIL command is positive (META_RESPONSE_OK), then the transition is committed and the sender email address is confirmed. On the other hand, if the server sent a negative reply (META_RESPONSE_ERR), then the transition is rolled back to laststatus. We also flag all negative server responses since they may indicate errors in the server if the original client command was correct.

The complete NERL recognizer for SMTP is included in Appendix C.3.
Figure 9.4: SMTP Server State Machine
Table 9.1: Events and State Transitions for the MAIL Command

transition m: MailFrom(m) OccurredWhen
    ((status == HELORECD) &&
     (respstatus == RESPDONE)) -> {
    status = MAILRECD;
    respstatus = RESPNONE;
    copy(senderseen.user,m.user);
    copy(senderseen.domain,m.domain);
    }

event MailFrom(h) OccurredWhen
    !((status == HELORECD) && (respstatus == RESPDONE)) ->
    Command_Error(b) WithAttributes
    {copy(b,"Unexpected MailFrom");};

transition mr2: MetaResponseOk(m) OccurredWhen
    (status == MAILRECD) -> {
    sender.user = senderseen.user;
    sender.domain = senderseen.domain;
    }

transition mr1: MetaResponseErr(m) OccurredWhen
    (status == MAILRECD) -> {
    status = laststatus;
    }

event MetaResponseErr(m) OccurredWhen
    (status == MAILRECD) ->
    NegResponse(n) WithAttributes
    {copy(b,"Mail Neg Resp");};
9.4.1 Recognizer Correctness

Given the relatively simple structure of the SMTP state machine, one might assume that writing the recognizer correctly is not that big an issue. However, there are several approaches to designing the recognizer that are inadequate. For instance, email surveillance programs, such as Altivore, implement an SMTP monitor but are only concerned with capturing the email and so ignore the responses. On the other hand, Network Intrusion Detection Systems, such as BlackICE, monitor SMTP sessions with a view to protecting the server from malicious clients. So they tend to ignore server errors because that is not their concern.

In order to check the correctness of server software, we want the SMTP monitor to obey the following property:

**Property 9.3** If an SMTP client process correctly sends an email $M$ to a correct SMTP server, then the monitor produces an `EmailAccepted` event containing $M$ if and only if the server successfully accepts the email $M$.

This property ensures that the monitor captures emails correctly. However, since we are monitoring the SMTP software for errors, we also want a *no false positives* property

**Property 9.4** If an SMTP client process and an SMTP server process carry out a correct SMTP session, then the monitor does not produce any error events.

As before, we prove these properties by encoding the SMTP sender and receiver in Promela and translating the NERL recognizer to Promela. In this case, we can assume that the channel between sender and receiver is reliable and FIFO because the channel is implemented by TCP.

We tried to check three versions of the SMTP monitor for these properties. The email surveillance software, Altivore, contains an SMTP monitor written in C and claims to imitate the FBI’s Carnivore. We encoded this monitor in NERL as recognizer $A$. We then tried to conjecture the kind of SMTP monitor used in a NIDS like BlackICE, based on survey reports [Gro01] and online documentation. We coded one such monitor in NERL as Recognizer $N$, which contains a more sophisticated stateful analysis like a NIDS might use. Finally we encoded our own SMTP recognizer that follows the complete SMTP state machine as $SMTP_{Recog}$.

We translated these three NERL recognizers to Promela and analyzed them using the SPIN model-checker. Recognizer $A$ fails and SPIN produces a counter-example: $A$ does not correctly handle the case when two `MailFrom` commands are issued by the client and the first is rejected. It captures the second email as part of the first email. To find this error, SPIN analyzed 871 states and 3135 transitions to produce a counter-example with 12 message exchanges. We then attempted the same proof for recognizer $N$. Again SPIN provides a counter-example: when a `Data` command results in an error response from the server, $N$ fails to notice this event and captures the next message sent as part of the same message. The SPIN counter-example has 16 messages and was found after analyzing 1610 states and 9897 transitions. Finally, we checked the property for the recognizer $SMTP_{Recog}$. SPIN model-checked this recognizer and found no errors; it analyzed 2330 states and 18,689 transitions to reach this conclusion.

In fact, as we shall see in the following sections, interpreting the SMTP standard correctly is a major issue and there are several ways to get it wrong.

9.5 Analyzing Live SMTP Executions

In our study, we concentrate on checking the correctness of the server, because SMTP servers are valuable resources and errors in server software are likely to have a bigger impact than errors in client software. Moreover, in the case of SMTP almost every email client has its own implementation of SMTP, while half the mail servers on the Internet run the same implementation - Sendmail, so errors in Sendmail are likely to be more important.
Our experimental setup consists of a Linux mail server running Sendmail 8.11.6, the default mail server packaged with RedHat Linux 7.1. We also test some other popular mail servers, Postfix 1.1.11 and Exim 4.10. The email client runs on a separate Linux machine on the same LAN. We primarily use a client of our own design. Sendmail comes with its own client that is fully compatible with its own server. As a result, it hardly ever tries to exercise the full range of commands allowed by SMTP. Our own SMTP client is written using the Expect terminal interaction language and uses Telnet to connect 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.

In addition, we use a second client Multimail to test the scalability of our SMTP monitor. Multimail is a mail server stress testing tool that attempts to create a large number of concurrent SMTP sessions with the server and delivers dummy emails.

We run our NERL monitor stack on a third machine on the same network. It observes all the traffic and flags errors and logs them. As mentioned earlier, the aim of the SMTP recognizer stack is to check if either the server accepts bad commands or rejects good commands in violation of the standard specification. In addition, if the server operates correctly, the monitor reconstructs the email sent between server and receiver.

We ran the SMTP monitor over 10 executions of this setup. The first few involved regular mail transactions as generated by a Sendmail client invoked through Pine. Then we ran some sessions between our Expect client and different SMTP servers: Sendmail, Exim, and Postfix. For instance, the first execution contains around 250 SMTP packets corresponding to 3 SMTP sessions. The log file contained no error events, only reconstructed email events. The log of the first email transaction, contained in packets 25-35, is shown in Table 9.5. In this trace, every line is prefixed by the packet number. The SMTP session began at packet number 14, but the first logged event is at packet 25 when the first envelope or sender-receiver pair is sent from the client to the server. This event is logged as EnvelopeSeen. The server then rejects the mail recipient (bkats@seas.upenn.edu) in packet 26, following which the client proposes another recipient in packet 27. In packet 28, the server accepts the second recipient, thus triggering an EnvelopeAccepted event. The client then sends an email in packet 34, which is then acknowledged in packet 35 - triggering an EmailSent event followed by an EmailAccepted event. The email is parsed according to the Internet Message Format and the parsed header fields are printed.

The Depends on clause that follows each event indicates the control dependencies of the output event - the complete sequence of packet that has resulted in the current output event. For instance, if we wish to know what caused the EnvelopeAccepted event to be triggered at packet 28, we should look at the packets between packet 14 and 28. This reduces the relevant trace by half. To narrow the search even further, the log contains Data Depends on clauses that indicate the data dependencies of the current output event - packets whose contents contributed to the contents of the current event.

In the following subsections, we describe error events that our SMTP monitor generated on some executions and the bugs they represent in the mail server software.

### 9.5.1 Server rejects valid sequences

When we analyze SMTP sessions between our Expect SMTP client and the Sendmail server, the monitor notices a large number of errors immediately, but almost all are client-side errors. Since our client is trying random sequences of commands, most of these sequences are incorrect. However, the server is still supposed to accept these sequences and produce the appropriate responses.

So we filter out all error events except for those concerning server responses. The largest number of server response errors are to do with negative responses to the EXPN and HELP commands. The server seems to reject these commands even if they are well formed. For instance, in one of these executions consisting of one session between our Expect client and the Sendmail server (903
Table 9.2: Output Trace of the Mail Monitoring Stack

<table>
<thead>
<tr>
<th>Line</th>
<th>Trace Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>25:</td>
<td>Found SMTP Envelope</td>
</tr>
<tr>
<td>26:</td>
<td>SMTP Response Error: Rcpt Neg Resp</td>
</tr>
<tr>
<td>27:</td>
<td>Depends on: &lt;14,15,16,18,19,20,22,23,24,25,26&gt;</td>
</tr>
<tr>
<td>28:</td>
<td>Found Acked SMTP Envelope</td>
</tr>
<tr>
<td>29:</td>
<td>Sender: <a href="mailto:bkarthik@verinet.cis.upenn.edu">bkarthik@verinet.cis.upenn.edu</a></td>
</tr>
<tr>
<td>30:</td>
<td>Receiver: <a href="mailto:bkarthik@seas.upenn.edu">bkarthik@seas.upenn.edu</a></td>
</tr>
<tr>
<td>31:</td>
<td>Envelope Depends on: &lt;14,15,16,18,19,20,22,23,24,25,26,27,28&gt;</td>
</tr>
<tr>
<td>32:</td>
<td>Envelope Data Depends on: &lt;23,25,27&gt;</td>
</tr>
<tr>
<td>33:</td>
<td>-------- Acked Envelope End</td>
</tr>
<tr>
<td>34:</td>
<td>Found SMTP Email</td>
</tr>
<tr>
<td>35:</td>
<td>Sender: <a href="mailto:bkarthik@verinet.cis.upenn.edu">bkarthik@verinet.cis.upenn.edu</a></td>
</tr>
<tr>
<td>36:</td>
<td>Receiver: <a href="mailto:bkarthik@seas.upenn.edu">bkarthik@seas.upenn.edu</a></td>
</tr>
<tr>
<td>37:</td>
<td>Text:</td>
</tr>
<tr>
<td>38:</td>
<td>Received: from verinet.cis.upenn.edu (verinet.cis.upenn.edu [158.130.13.33])</td>
</tr>
<tr>
<td>39:</td>
<td>by verinet.cis.upenn.edu (8.11.6/8.11.6) with ESMTNP id g2ELgTA05682;</td>
</tr>
<tr>
<td>41:</td>
<td>Date: Thu, 14 Mar 2002 16:42:29 -0500 (EST)</td>
</tr>
<tr>
<td>42:</td>
<td>From: Karthik <a href="mailto:bkarthik@verinet.cis.upenn.edu">bkarthik@verinet.cis.upenn.edu</a></td>
</tr>
<tr>
<td>43:</td>
<td>To: <a href="mailto:bkarthik@seas.upenn.edu">bkarthik@seas.upenn.edu</a></td>
</tr>
<tr>
<td>44:</td>
<td>cc: <a href="mailto:bkats@seas.upenn.edu">bkats@seas.upenn.edu</a></td>
</tr>
<tr>
<td>45:</td>
<td>Message-ID: <a href="mailto:Pine.LNX.4.44.0203141642070.5680-100000@verinet.cis.upenn.edu">Pine.LNX.4.44.0203141642070.5680-100000@verinet.cis.upenn.edu</a></td>
</tr>
<tr>
<td>46:</td>
<td>MIME-Version: 1.0</td>
</tr>
<tr>
<td>47:</td>
<td>Content-Type: TEXT/PLAIN; charset=US-ASCII</td>
</tr>
<tr>
<td>48:</td>
<td>.</td>
</tr>
<tr>
<td>49:</td>
<td>-------- Acked Email End</td>
</tr>
<tr>
<td>50:</td>
<td>-------- Parsed Internet Message Format ------------------------------------------</td>
</tr>
<tr>
<td>51:</td>
<td>Date: Thu, 14 Mar 2002 16:42:29 -0500 (EST)</td>
</tr>
<tr>
<td>52:</td>
<td>From: <a href="mailto:bkarthik@verinet.cis.upenn.edu">bkarthik@verinet.cis.upenn.edu</a></td>
</tr>
<tr>
<td>53:</td>
<td>To: <a href="mailto:bkarthik@seas.upenn.edu">bkarthik@seas.upenn.edu</a></td>
</tr>
<tr>
<td>54:</td>
<td>Cc: <a href="mailto:bkats@seas.upenn.edu">bkats@seas.upenn.edu</a></td>
</tr>
<tr>
<td>55:</td>
<td>Msg-ID: <a href="mailto:Pine.LNX.4.44.0203141642070.5680-100000@verinet.cis.upenn.edu">Pine.LNX.4.44.0203141642070.5680-100000@verinet.cis.upenn.edu</a></td>
</tr>
<tr>
<td>56:</td>
<td>-------- Parsed Internet Message Format Ends -------------------------------------</td>
</tr>
</tbody>
</table>
packets), the first response error is on packet no 14

| 14: SMTP Response Error: Expn Neg Resp |
| 14: Depends on: <1,2,3,6,8,9,10,11,12,13,14> |

On checking the TCP trace log for packet 14, we find the server response:

| 502 5.7.0 Sorry, we do not allow this operation |

So the server was simply following some security policy that disallowed EXPN commands. This was a false alarm and the server is really not at error. This is a common occurrence in SMTP monitoring - different mail servers vary in the set of commands they are willing to accept. We find that various configurations of Sendmail, Exim and Postfix disallow EXPN, HELP and VRFY commands because they are considered security risks.

**Sendmail Rejects Second HELO**

Later in the same execution described above, we find a server negative response to the HELO command. This is certainly unusual because the server is supposed to always accept HELO, NOOP, RSET and QUIT. If it rejects a HELO command for policy reasons, then it must logically reject all other commands as well. The HELO response error is at packet number 68

| 68: SMTP Response Error: Hello Neg Resp |
| 68: Depends on: <1,2,3,6,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68> |

The depends clause contains all the packets in the current SMTP session. So we start looking backwards along these packets to understand the current response. The TCP trace log has packet 67 containing the command

```
HELO verinet.cis.upenn.edu
```

and packet 68 containing the response

| 503 5.0.0 buddha.cis.upenn.edu Duplicate HELO/EHLO |

On tracing back we find that there was a HELO command earlier in the session. So the Sendmail server does not accept multiple HELO commands. This is a violation of the standard [Kle01]. On researching the cause of this error, we discovered that in an intermediate standard for SMTP, RFC1651, duplicate HELO’s were considered harmful and it was recommended that servers reject them. However, the new version of SMTP (since 2001) explicitly allows multiple HELO’s.

This error is considered a bug in Sendmail and has been corrected in the latest version of the Sendmail (8.12) software.

**Exim Rejects MAIL After Bad HELO**

Next we analyze a long execution of our Expect client against an Exim mail server. In this execution, the SMTP recognizer produces a large number of “Junk Response” errors which means that the server’s response was unparsable. On closer inspection, all the badly formatted responses were of the same kind. For instance, the first such error event says

| 23: SMTP Response Error: Junk Response: 550 HELO argument does not match calling hoststate |
| 23: Depends on: <1,2,3,5,6,7,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23> |

102
Packet no 23 contained the data

```
23: 550 HELO argument does not match calling host
```

This response was unparsable, because all SMTP commands and responses are supposed to end with the characters \r\n (carriage return, line feed), while this command only ended in \n. All other responses produced by Exim did have the preceding \r character. So this is a simple formatting bug where the developers forgot to add a \r in this particular response. We have since communicated it to the Exim developers.

To continue analyzing this trace, we modify our own SMTP parser to accept these responses from the Exim server. This is an instance of the tuning technique, where we modify our own monitoring code rather than modify the source code of the protocol implementation. We then run the same client and server against each other.

In this new execution, 611 SMTP sessions were started and 29 emails were exchanged; a total of 23981 packets were exchanged. The SMTP recognizer produces 1837 response errors, all of them negative responses to commands. Of these, we again find that a large number are negative responses to EXPN commands (583 of them). In addition 531 are negative responses to HELO commands. We looked at these error event and found that all of them were valid rejections because the Expect client was using bogus parameters in the HELO command. Similarly the 506 negative responses to RCPT commands all seem related to bad email addresses in the MAIL and RCPT commands.

That leaves us with the 200 negative responses to MAIL commands and the 17 negative responses to DATA commands. Of these, some seemed quite suspicious. For instance, consider the MAIL response and corresponding error event at packet 47 (the MAIL command was packet 46):

```
47: 550 HELO or EHLO required
```

But this error event indicates that the current state is HELLO, which means that a successful HELO must have been sent before this MAIL.

Similarly each of the 17 negative responses to DATA commands looked suspicious:

```
1592: 503 MAIL command must precede DATA
```

Here the response claims that no MAIL command was sent while the error event indicates that the current state is RCPT which means that both successful MAIL and RCPT commands must have been sent.

By tracing back from these errors, we find that each such error is preceded by a HELO command that fails. When a HELO command fails, the SMTP standard says that the server should stay in the same state as before. It seems that Exim is resetting the state and clearing all buffers whether or not the command succeeds.

To test this hypothesis, we again tune our recognizer to always reset state on HELO commands, and indeed all these incompatibilities go away. This represents another bug in Exim, and we confirmed the existence of the bug in the Exim source code. We have since notified the developers who acknowledged and fixed this error.

### 9.5.2 Server accepts invalid sequences

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. Before the first HELO command, clients are allowed to send informational queries such as HELP, VRFY and EXPN. In addition, session state can be cleared or deleted by issuing RSET, NOOP, or QUIT commands at any time in the session. So our Expect client mixes up these commands and tries out random sequences on the server.

The duplicate HELO error was a case in which the server rejected a valid command that it should have accepted. When monitoring several other sessions for errors in server responses, we find the server accepting commands that it should have rejected.

Servers accept MAIL without HELO

First we analyze a long execution of our client with the Sendmail server. This execution has 441 SMTP sessions that deliver 133 emails over 25527 packets. The recognizer generates a large number of error events, but the ones that catch the eye are “Junk Response is OK” events. For instance, the first instance of such an error is at packet no 53

```
53: SMTP Response Error: Junk Resp is OK! Code: 250, state: JUNK, respstate: RESPDONE
53: Depends on: <43, 44, 45, 47, 48, 49, 51, 52, 53>
53: Data Depends on: <52, 53>
```

This indicates that the SMTP response at packet 53 is positive even though the preceding command was incorrect. On inspecting the command sequence in the indicated packet sequence (<43, 44, 45, 47, 48, 49, 51, 52, 53>), we find that the packets 43, 44 and 45 established the TCP connection. Packet 47 contains the server’s greeting message. Packet 48 acknowledges this greeting, and packet 49 contains the NOOP command which then gets a positive response in packet 51. Thus far no real command has been sent.

Then, Packet 52 contains the MAIL command, which is incorrect because no preceding HELO exists, and the server should reject it. But the server accepts this MAIL command and sends a positive response in packet 53

```
250 2.1.0 <bkats@verinet.cis.upenn.edu>... Sender ok
```

This error is not isolated because the same pattern recurs later in the execution - the server consistently accepts MAIL commands without preceding HELO’s. In fact at packet 860, the server accepts an entire MAIL transaction without the preceding HELO.

This error is a direct violation of the standard. All SMTP standards, old and new, explicitly state that a mail transaction must be preceded by a HELO command. This is important because often a server will base its mail acceptance policy on the parameters to the HELO command.

Moreover we find that this behavior is consistently present across all versions of Sendmail, Qmail, Postfix and Exim. However, all of them offer configuration variables that when set will enforce a HELO before a MAIL. For instance, if the sendmail “privacy” option needmailhelo is set, then the server will require the client to send a HELO before sending a MAIL. This flag has the same status as the needvrfyhelo flag that similarly enforces a HELO before VRFY for server security. These flags are considered optional security mechanisms for the server.

We believe it is incorrect to bunch such options together because while flags like needvrfyhelo force the server to act stricter than usual (the standard allows VRFY before HELO), the needmailhelo flag is necessary for conformance with the standard.

We note a related error in Sendmail. When a MAIL is sent before a HELO, it is accepted as noted above. Now the next HELO should reset the state of the server, but Sendmail treats the first HELO in a special way and does not reset state. As a result, the sequence of commands MAIL, RCPT, HELO, DATA is allowed by Sendmail while prohibited by the standard on two different counts. This second error has a potential for breaking interoperability with other servers even if all
of them allow MAIL before HELO, because they all treat the first HELO differently. For instance, while Postfix also treats the first HELO like Sendmail, Exim resets state on the first HELO.

**Postfix treats second HELO as NOOP**

While Sendmail 8.11.6 rejects the second HELO, Postfix and Exim accept it. We have shown that 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 find this error in an execution of our client with the Postfix client - 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>
```

On tracing back 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 (EXPN, HELP, etc.)

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.

It turns out that Postfix treats the HELO command differently from EHLO (the new HELO) and for EHLO it does reset the state. However, the processing for EHLO commands contains the same error as in Exim - it always reset the state even if the EHLO failed.

**RSET assumes HELO**

Finally, we point out an error in the SMTP module of the Bro intrusion detection system, in order to indicate another potential error in the design of the mail server. While the error we describe in this section does not exist in any mail server that we analyzed, we believe it is important to understand how such errors might creep in to an implementation.
The RSET command can be issued at any time and must clear all buffers and reinitialize state. The primary purpose of this command is that if the server rejects an email or if the client decides to abort a transaction, the session can be reset and a new mail transaction can begin. In this scenario, issuing an RSET is considered more efficient than issuing a new HELO command, because the HELO command must be processed according to the server’s policy. The SMTP standard says that the HELO and RSET commands behave in the same way when aborting a mail transaction and issuing one or the other should not matter. This indicates that an RSET should also set the server state to HELORECD. Indeed the Bro NIDS, when following the SMTP server state machine sets the state to HELORECD on seeing a RSET.

This is correct in all cases except one - in a session before the HELO command is issued, a RSET command should set the new state to CLEAR and not to HELORECD. So a correct implementation of HELO needs two variables - one indicating that at least one HELO has been recd, and the other containing the current state: HELORECD or CLEAR.

This error may seem contrived in the sense that no real mail server contains this error. However, the Bro NIDS actually follows a principled approach of following complete protocol state machines where possible. Our study indicates that even such a well designed tool can have errors because of misinterpretations of the standard.

### 9.6 Performance

While monitoring the sessions described in the previous sections, the 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 analyses of SMTP traces. In Table 9.6 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 very small compared to the actual execution time.

In Table 9.6, 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 calculates as the difference in time between the start and end of the trace, the analysis time taken by the NERL 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.

To measure monitor performance when there were a large number of concurrent sessions, we used an SMTP stress testing tool Multimail to generate executions of many threads of clients interacting with a single Sendmail server. We gathered three such traces, represented as traced 4, 5, and 6 in Table 9.6. Notice that for these traces the period of execution is quite small - the Multimail clients are bombarding the server with packets - but the analysis time can still keep up. The largest trace was of 10 threads generating 100 emails each. We found that the SMTP

---

**Table 9.3: Time and memory usage for Expect trace analysis**

<table>
<thead>
<tr>
<th>Trace</th>
<th>Packets</th>
<th>Sessions</th>
<th>Emails</th>
<th>Execution(s)</th>
<th>Analysis(s)</th>
<th>Memory(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25527</td>
<td>441</td>
<td>133</td>
<td>1487.26</td>
<td>5.75</td>
<td>5.69</td>
<td>19.4</td>
</tr>
<tr>
<td>2</td>
<td>23981</td>
<td>611</td>
<td>48</td>
<td>605.42</td>
<td>4.55</td>
<td>7.52</td>
<td>15.15</td>
</tr>
<tr>
<td>3</td>
<td>8825</td>
<td>120</td>
<td>45</td>
<td>4309.37</td>
<td>1.92</td>
<td>1.60</td>
<td>5.79</td>
</tr>
</tbody>
</table>

---

9.6 Performance

While monitoring the sessions described in the previous sections, the 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 analyses of SMTP traces. In Table 9.6 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 very small compared to the actual execution time.

In Table 9.6, 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 calculates as the difference in time between the start and end of the trace, the analysis time taken by the NERL 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.

To measure monitor performance when there were a large number of concurrent sessions, we used an SMTP stress testing tool Multimail to generate executions of many threads of clients interacting with a single Sendmail server. We gathered three such traces, represented as traced 4, 5, and 6 in Table 9.6. Notice that for these traces the period of execution is quite small - the Multimail clients are bombarding the server with packets - but the analysis time can still keep up. The largest trace was of 10 threads generating 100 emails each. We found that the SMTP
recognizer could keep up with this session as well, it captured all 1000 emails successfully. The most number of concurrent sessions we tested against was 150.

We find the the processing time used by the recognizer is roughly proportionate to the size of the output file, or the number of bytes transferred in a session. For SMTP, this was 0.3 seconds per MB of output trace. In terms of packets, the recognizer handles between 4400 and 8400 packets per second. The peak memory requirement varies with the number of sessions monitored. For SMTP, the requirement per session was 12.8 KB per session. We note that the memory requirement does drop drastically as tracing features are disabled, so some optimization work might be required in the tracing implementation.

## 9.7 Results

**Errors.** We summarize the errors found by our analyses of SMTP sessions. 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 the following errors:

**Sendmail 8.11.6** has the following errors

1. It accepts emails without a preceding HELO command. This behavior can be corrected by modifying a privacy option (**needmailhelo**).
2. It does not reset state on receiving the first HELO.
3. It rejects the second HELO in an SMTP session, claiming that it is a duplicate HELO forbidden by an intermediate SMTP standard.

**Postfix 1.1.11** has the following errors

1. It accepts emails without a preceding HELO command. This behavior can be corrected by modifying a configuration option (**verify_helo_hosts**).
2. It does not reset state on the first HELO.
3. It does not reset the server state on receiving a second HELO. HELO’s in Postfix are always treated as NOOPs with regards to the state machine.
4. It resets the state on receiving an EHLO even if the EHLO was unsuccessful. machine.

**Exim 4.10** has the following errors

1. It accepts emails without a preceding HELO command. This behavior can be corrected by modifying a configuration option (**smtpd_helo_required**).
2. It’s negative response to a HELO command does not end in the mandatory \r\n.
3. It resets the state on receiving a HELO even if the HELO was unsuccessful.

All of the errors we found represent violations of the SMTP standard. As is evident, all the errors we found are based on different and incorrect interpretations of the HELO command. A correct specification of the HELO command would be as follows.
The first HELO command in a session initializes a mail transaction. It is necessary before any mail exchange - MAIL, RCPT, DATA - can take place.

A subsequent HELO command in a session has two possible behaviors

- if it succeeds (generates a positive server response), then it behaves exactly like a RSET, and
- if it fails (generates a negative server response), then it behaves exactly like a NOOP.

We believe that the standard should be modified to include such 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.

**Performance.** We find that the SMTP monitor 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.

**Expressiveness.** We demonstrate the expressiveness of the NERL and NERL\textsubscript{MOD} languages by programming an entire stack of recognizers. We show that it is important to be able to interpret the SMTP standard and translate it faithfully into NERL. The NERL\textsubscript{MOD} program successfully combines recognizers written in NERL, Lex and C to execute the SMTP monitor.

**Correctness.** While we find that monitor efficiency is not a major issue for SMTP, checking recognizer correctness is critical. We demonstrate several approaches to programming an SMTP recognizer, based on surveillance software and intrusion detection systems. We model-check all versions and find subtle errors. We show that our SMTP recognizer is correct.
Chapter 10
TCP: Reliable, In-order Delivery

TCP is the primary transport protocol on the Internet, to the extent that along with IP, ICMP, and UDP, it is considered an essential requirement for “host system implementations of the Internet protocol suite” [Bra89b]. A key to the popularity of TCP is that 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, even newer transport requirements such as secrecy 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 several critical Internet services. In the previous chapters we have shown how to passively monitor protocols when the monitor is co-located, and when the protocol operates on a reliable channel such as that provided by TCP. Monitoring protocols that do not operate on reliable channels brings into play monitoring infidelities caused by packet delivery non-determinism in the IP layer. In particular, the monitor cannot be sure that the packets it sees are exactly the packets seen or produced at the implementation under test. In testing such low-level protocols, one must distinguish between packet errors at the IP layer and errors in the protocol implementation, and this is difficult.

In this chapter, we shall attempt to monitor TCP implementations in a co-networked environment. First, we describe the TCP protocol to the extent that we shall be testing it - the TCP state machine and reliability protocol. We shall not be considering congestion control because it has many standards, and it can be considered as a thin layer above TCP reliability and analyzed like SMTP in the previous chapter. Then, we shall describe the NERL recognizer for TCP and attempt to monitor TCP implementations in a co-networked monitoring environment.

The chief characteristics of this case study are as follows

- Transport layer protocol
- Two participants - sender and receiver
- Established, infrastructural protocol
- Operating system implementations: Linux, Solaris, Windows XP
- Monitoring environment: co-networked

There is a wealth of previous work on TCP testing (see 4.1.1). While most of the efforts concentrate on the congestion control aspects of TCP, the errors they find also impact reliability and conformance to the standard state machine. Paxson carried out the most comprehensive passive testing study of TCP implementations using the tcpanaly tool and found errors in all major operating system implementations [Pax97]. However, this work fails in its original aim to create a
general, online monitor for TCP implementations. Paxson cites two reasons for this failure. First, online or one-pass analysis turns out to be infeasible for tcpanaly because of *vantage point issues*: the monitor does not see the trace at the device. These issues are closely related to the trace infidelities we have described earlier for non co-located monitors. Second, generic TCP monitoring is infeasible because of the wide variation in behavior between different TCP implementations. These variations stem from ambiguities in the TCP protocol that are meant to allow an implementation freedom in choosing its strategies - primarily with regard to congestion control. As a result, there are several *flavors* of TCP, each choosing one implementation strategy. So, Paxson writes a different monitor for each flavor that uses deep knowledge of the implemented strategy, and each monitor makes multiple passes over an offline captured trace to check an implementation for errors.

In our study, we write a general, online monitor for TCP state machine conformance and reliability properties. We address vantage-point issues by using the co-networked monitoring algorithm from Chapter 7. For our set of properties, we do not find that implementation ambiguity is a problem. However in general, we note that in the NER architecture, as long as there are a fixed number of flavors, we can always write different TCP monitors for each flavor and put them in parallel at the same layer sending copies of TCP events to each monitor. An alternative approach that we do not investigate in this chapter is to use a version of the co-networked monitoring algorithm that treats implementation ambiguity as an additional non-deterministic module.

The primary aim of this case study is to develop strategies for monitoring protocols in a co-networked environments. While it is unlikely that the well-established operating system implementations of TCP have significant errors, we shall attempt to understand what properties of TCP-like transport protocols can be feasibly checked by a passive monitoring system.

### 10.1 Transmission Control Protocol

TCP has been described in great detail and with increasing clarity in several RFC's [Ins81b, Bra89b] and books [Ste94]. Here we shall only highlight the portions that we intend to model. The TCP state machine is shown in Figure 10.1.

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 a 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 - it implements 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.

The TCP connection state machine has been described in detail in the standard specification [Ins81b]. Figure 10.1 contains the *observable* part of the state machine: TCP states and the
Figure 10.1: TCP State Machine
packets that denote state transitions. This state diagram does not describe the packet attributes in detail, it only represents the control states of a TCP sender or receiver.

### 10.1.1 TCP Reliability

The actual transfer of data takes place in the ESTABLISHED state, but some data transfer can also take place after one or both FINs have been sent. Figure 10.2 shows the control states in which data is sent, received, and acknowledged. In this section, we discuss the details of these DATA and ACK segments.

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 all been 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 $S_{\text{max}}$, the receiver has received contiguous data up to sequence number $S_{\text{Cont}}$, 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 $S_{\text{Cont}}$ monotonically increases, this also means that

$$A_N \geq A_L$$

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

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

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.

10.2 TCP Properties in NERL

Given the TCP state machine in Figure 10.1, we write a NERL recognizer to check deviations from it. Encoding the state machine is straightforward - the recognizer maintains a variable $\text{status}$ - and updates it as it sees input events.

The more involved part of TCP recognition is in checking the DATA and ACK packets for errors. We program the recognizer to check the following TCP properties:

1. ACK sequence numbers are non-decreasing.
2. The implementation is generating an ACK for at least every other message it receives.
3. ACKs always acknowledge exactly the contiguously received set of segments.
4. The sender implementation only sends data segments in the advertised receiver window. The first three properties check that the receiver is producing the right ACKs which is important for ensuring reliability. The last property checks that the sender is indeed respecting the receiver window and retransmitting lost segments instead of forging ahead past the window. The complete NERL recognizer for TCP is given in Appendix D.1.

10.3 Initial Analysis

To start with, let us ignore any infidelities in monitoring and simply run the TCP recognizer on a trace assuming that we have captured all the packets between the sender and receiver, and in the right order.

When we run the recognizer on some of the SMTP traces collected in the last chapter, we immediately notice a large number of errors. First, on a trace with 270 packets and one SMTP session, we see 2 errors of the same kind:

```
62: TCP HS Error:10:ACK sent with bad ack: ACK 188033049, seq 188034011, fin 4294967295
62: Depends on:<45,46,47,49,51,52,53,54,55,56,57,58,59,60,61,62>
```

Seemingly the receiver sent an incorrect sequence number 188033049 as its ACK. On tracing back through the packets indicated in the depends clause, we find that the previous packet sent by the sender was:

```
61: Flags ACK
61: Depends on:<61>
61: Found TCP Data len=963
```

So the sender has already sent up to sequence number $S_{Cont} = (188033049 + 963 - 1) = 188034011$ and the receiver should have actually sent an ACK 188034012 and not 188033049 ($A_N = S_{Cont} + 1$). This seems to be an error in the TCP receiver, but then we notice that the very next packet 63 is an ACK sent by the receiver:

```
63: Flags ACK
```

This ACK has the right sequence number. This means that the data segment in packet 61 was buffered at the receiver while it produces the ACK in packet 62. Then it processed the data segment in packet 61 and produced the correct ACK in packet 63. The receiver implementation was not incorrect, there simply was a buffer between the link layer and the IP layer and the receiver could not keep up with the speed at which packets arrived.

This kind of buffering is triggered as an error and occurs even in this small, controlled TCP trace. In larger traces, it can result in a much larger number of errors. For instance, in a TCP trace with 25527 packets there were as many as 788 such "bad ackno" error events. Clearly, these events can only serve to distract the monitor from finding real implementation errors.

In our analysis of the SMTP traces, our NERL recognizer generates error events for each of the following cases in almost all of the traces we analyze.

- Too many unacked packets: the receiver does not generate an ACK for every two packets.
• Incorrect ACKs: the ACKs that the receiver does produce do not acknowledge the packets it has received.

In fact we find that analyzing traces of Windows and Solaris TCP implementations also produces these error events.

Without modeling the buffers, it is impossible to tell if these errors really exist in the TCP implementations or are just a result of lower layer buffering. In the following sections, we shall attempt to model the IP buffer and distinguish between these cases.

10.4 Monitoring Environment

To understand the TCP monitoring environment, first note that while TCP provides a reliable duplex channel to the higher-layer application protocol, TCP itself operates on top of multiple unreliable, buffered, simplex IP channels. Both the TCP sender and receiver interact with two channels: an incoming channel and an outgoing channel. At the level of IP, there is no reason to think that the two channels are correlated - packets in each direction may be buffered or lost independent of the other direction.

If there was no buffering or loss on these channels, we could simply observe the inputs and outputs and assume that that is exactly the order and content of the protocol exchange. Unfortunately, even in a controlled LAN we observe that inputs are often buffered at a host before being consumed and in rare cases, when these buffers overflow, inputs are dropped as well.

Moreover, a protocol like TCP is predicated on the notion of a receiver window - the receiver tells the sender how many packets it is willing to accept. The sender is then free to send that many packets without waiting for any more information from the receiver. Usually, the sender then fires off these packets faster than any receiver can process them, assuming that they will get buffered and read by the receiver. So for TCP-like protocols that are trying to send data as fast as possible, buffering is also imposed by the protocol behavior. Therefore a monitor sitting between two TCP implementations A and B (Figure 10.3) actually monitors 4 channels:

1. The IP packets sent from A that are seen at M: the output channel at A
2. The IP packets sent from A that pass M and reach B: the input channel at B
3. The IP packets sent from B that are seen at M: the output channel at B
4. The IP packets sent from B that pass M and reach A: the input channel at A

We assume that there is no buffering or loss on the output channels, and that there could be buffering and loss on the input channels. So an output on channel 1 does not immediately or always translate to an input on channel 2. This is why channels 1 and 3 are independent from channels 2 and 4 respectively. On the other hand, inputs at A go through a different buffer from the inputs at B, so channel 2 is independent from channel 4. The only two channels that are correlated are channels 1 and 3 - since we assume that outputs are not buffered or lost, any output events from A or B can be considered to occur in exactly the observed order. This also means
that monitoring properties of just output events is much simpler than monitoring properties that correlate input and output events.

Earlier in Chapter 7, we have described a co-networked monitoring algorithm that takes a recognizer and annotations indicating its input and output channels and automatically generates code for dealing with the non-deterministic channel behavior.

An alternate approach would be to rewrite a specialized TCP recognizer that is aware of input buffering and loss. Using the particular features of TCP we may be able to write an efficient co-networked TCP monitor without using the general trace search algorithm. While we do present some specialized TCP recognizers toward the end of this chapter, we believe it is important to develop and use a general algorithm because these infidelity issues will present themselves time and again in various protocols in the Internet stack. All protocols that sit on unreliable channels, such as UDP that sits on IP, or NFS that sits on UDP, will need similar techniques to what we describe here. Moreover, even protocols such as FTP that sit on two TCP channels will need similar techniques to deal with the interleaving of segments on the two channels. So while our co-networked techniques may not be very efficient in particular cases, they must be considered as a first automated cut at co-networked analyses of these kinds of protocols.

To apply the trace search algorithm to the TCP recognizer, first consider its signature

```plaintext
input event ty_tcpflow Init;
input event ty_tcppkt TCPTo;
input event ty_tcppkt TCPFrom;

output event bool Done;
output event ty_tcpdata TCPDataTo;
output event ty_tcpdata TCPDataFrom;
output event string HSError;
output event string DataError;
```

Now, to run the co-networked monitoring algorithm, we first define 4 channels - 2 in each direction. For each direction, when a TCP packet is seen, it indicates two events TCPSent and TCPRecd. Earlier, we could assume that these are the same event because there were no buffers. Now, TCPRecd could be delayed because of a buffer at the receiver.

The new signature for TCP follows. We have decided to assume input buffer sizes of 5, with no loss. Outputs are assumed to be unbuffered since both sender and receiver are co-networked.

```plaintext
ichannel ITo[5,0]
ichannel IFrom[5,0]
ochannel OTo
nochannel OFrom

input event ty_tcpflow Init;
input event OTo ty_tcppkt TCPSentTo;
input event ITo ty_tcppkt TCPRecdTo;
input event OFrom ty_tcppkt TCPSentFrom;
input event IFrom ty_tcppkt TCPRecdFrom;

output event bool Done;
output event ty_tcpdata TCPDataTo;
output event ty_tcpdata TCPDataFrom;
output event string HSError;
output event string DataError;
```

116
This signature can be used by the co-networked NERL\textsubscript{MOD} compiler to model 2 input buffers for each TCP connection and generate every plausible sequence of TCPSent and TCPRecd events and trigger the TCP recognizer on every such sequence.

For instance, when a data segment is seen in the To direction, the TCPSentTo event is triggered immediately. Now this segment could be buffered at the receiver. Or if the receiver buffer is empty, it could have been consumed. This gives rise to 2 possibilities and the algorithm makes two copies of the state of the recognizer and tries these two sequences on them. In this manner, the algorithm keeps track of several instance-copies of TCP recognizer along with possible states of the IP packet buffer. When a recognizer instance produces an error event and flags the Done event, that copy is deleted and a “PossibleError” event is triggered. If all instance-copies of a recognizer are deleted, a “DefiniteError” event is triggered indicating that there are no more valid sequences that could explain the behavior of the TCP implementations under test, so they must be wrong.

### 10.5 Analyzing SMTP traces

When we run the modified TCP recognizer on the same SMTP traces as before, now 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:

<table>
<thead>
<tr>
<th>62: Possible TCP HS Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP HS Error:5b:ACK sent with bad ack, sender state:3, rcvr state: 3</td>
</tr>
<tr>
<td>Depends on:&lt;45,46,47,49,50,51,52,53,54,55,56,57,58,59&gt;</td>
</tr>
<tr>
<td>62: Possible TCP HS Error</td>
</tr>
</tbody>
</table>

This error is possible which means that one of the plausible sequences generated by the co-monitoring algorithm is rejected because of this error. But this error never translates to a “Definite TCP HS Error” because the monitor takes buffering into account and so there are other plausible sequences which do not have this error. There are no definite TCP errors in this trace.

But in a larger trace, we do find instances of definite errors. For instance, in the 25527 packet trace where we earlier found 788 “bad ackno” error events (935 total error events), we now find 76 “Definite TCP Error” events. The first is at packet 449:

<table>
<thead>
<tr>
<th>449: Possible TCP HS Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP HS Error:2a:Syn sent in bad state, sender state:0, rcvr state: 1</td>
</tr>
<tr>
<td>Depends on:&lt;&gt;</td>
</tr>
<tr>
<td>449: Possible TCP HS Error</td>
</tr>
<tr>
<td>TCP HS Error:2a:Syn sent in bad state, sender state:9, rcvr state: 1</td>
</tr>
<tr>
<td>Depends on:&lt;448&gt;</td>
</tr>
<tr>
<td>449: Possible TCP HS Error</td>
</tr>
<tr>
<td>TCP HS Error:2a:Syn sent in bad state, sender state:9, rcvr state: 1</td>
</tr>
<tr>
<td>Depends on:&lt;448&gt;</td>
</tr>
</tbody>
</table>

Here there were two plausible sequences both of which were found to be incorrect. On tracing back we find that the sender sent two SYN packets to the receiver in succession. The receiver never responded to the first, so that SYN might be considered lost. We modify the input channels to allow loss of 1 packet at a time:
Now when we re-run the monitor, the error disappears. We find no “Definite” errors in the resulting trace. We analyzed all the SMTP traces, that consisted of TCP sessions between Linux 2.4 machines and our recognizer could find no errors. All the errors we found earlier only appeared as possible errors in each trace. In the next section, we generate and analyze traces of Linux, Windows and Solaris traces and attempt to find errors in each of them.

It is important to note that while finding an error would have been significant, just knowing that the few thousand error events generated earlier were false alarms is a big time saver. The co-networked algorithm effectively sifts through these “PossibleError” events and throws away everything that can be explained by buffering and loss. Normally, this would be done manually, leading to a huge waste of time.

10.6 Analyzing ttcp Traces

Test TCP (ttcp) is a tool meant to send a large amount of data between two hosts over TCP and compute the bandwidth and computing resources consumed by the TCP implementation at each end. We generated several traces of ttcp sessions between Linux 2.4, Solaris 5.8, and Windows XP machines and analyzed them using our co-networked 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.

The co-networked monitor was able to follow the ttcp sessions accurately even though a large number (up to 44) packets were being buffered in each direction. All three TCP implementations were found to be mainly error-free with respect to our reliability properties except for one particular case.

The TCP standard says that a TCP receiver must send an acknowledgment for every two data segments that it sees. When data segments start arriving out of order the receiver should send an acknowledgment for every data segment. While Linux TCP traces did not trigger any error events, both Solaris and Windows TCP traces failed this property. For instance, the following error is generated in a Windows ttcp trace

| 36: ------------------------------------------------- |
| 36: Definite TCP Error                              |
| 36: ------------------------------------------------- |

On tracing back we find that the cause of the error is at packet 35, where there are a large number of possible errors of the form

| 35: ------------------------------------------------- |
| 35: Possible TCP Error                             |
| 35: TCP Error:11c:Too many unacked packets: 3      |
| 35: ------------------------------------------------- |

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. This is a form of tuning, where we modify our recognizer to mimic the implementation’s erroneous behavior. 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.

Our observation of Solaris’ ACK delays is not particularly novel. Paxson found this in his analysis of TCP implementations [Pax97] and the notion of delaying ACKs is a known implementation 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 did find would not count as a significant errors in the TCP implementations, this study does demonstrate the usefulness of packet buffer modeling. For instance, let us look at a fragment of the TCP trace of a Linux implementation

```
   -- seqno:3257542433 ackno:433220911
57:Flags ACK DATA
-----------------------------------------------
   -- seqno:3257543881 ackno:433220911
58:Flags ACK DATA
-----------------------------------------------
   -- seqno:3257545329 ackno:433220911
59:Flags ACK DATA
-----------------------------------------------
   -- seqno:3257546777 ackno:433220911
60:Flags ACK DATA
-----------------------------------------------
   -- seqno:433220911 ackno:3257522161
61:Flags ACK
```

Here the Linux TCP receiver (158.120.012.217) replies with an ACK only after four consecutive DATA packets. This seems like incorrect behavior, as does the following fragment of a Windows XP TCP trace
However, the Linux implementation is in fact correct, and the fragment we show here can be explained as 2 packets getting buffered and 2 being consumed before the receiver sends an ACK. On the other hand, the Windows implementation is incorrect because as we go further in the trace we find that there is no buffering that can account for the few number of ACKs produced. Indeed, Windows often sends a single ACK for as many as 10 data packets, rather than generating 5 separate ACKs. Clearly, only by taking buffering into account correctly can we hope to distinguish between these implementations.

10.7 Scalability

The time taken for the analysis of traces described in the previous section are shown in Table 10.7. Each trace consists of one TCP session between a ttcp sender and receiver. The first two traces were generating with Linux implementations and no errors were found. For the Windows and Solaris traces, we had to relax the properties to allow 5 packets to be acked at a time (instead of 2). In one case, we had to allow 10 packets to be acked. Before we relaxed these requirements, the NERL recognizer produced an error event and exited in seconds.

In Table 10.7, each row lists from left to right, the trace number, the implementation that produced the trace, 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.

Note how both the analysis time and memory usage is significantly greater than the two other case studies in this thesis. The co-networked monitor can use up to 23MB of memory to analyze a 5000 packet trace, and consumes packets at around 500 packets per second. 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 increases the number of plausible instances. But for lossy traces, it cannot handle more than 4 TCP sessions at a time.

The co-networked algorithm that 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 protocol-specific properties that often reduce the number of plausible instances that we analyze.

**Finite State** Several protocol properties are finite state. For instance, the SMTP recognizer was effectively finite state since it was only checking for conformance with a finite state machine. Similarly, parts of the TCP specification, such as the connection state machine are also finite state.

For finite state properties, the number of plausible instances is limited to number of states multiplied by the number of states of the buffers \( \Gamma \times 2^B \). The states in an FSM essentially defines an equivalence property on the set of plausible instances.

**Responsiveness** We say that a protocol is responsive if a participant must produce an output for every \( C \) inputs received. For instance, in TCP this number is 2, while in SMTP \( C \) is 1 - it must respond to every input. Responsive protocols limit the number of plausible buffering of packets, because no more than \( C \) inputs may be fed to a participant without it producing an input. All the rest must be buffered.

In addition, there are several property based optimizations that we can carry out for specific kinds of protocols. In the next section, we describe some of these for TCP.

### 10.8 Monitoring Optimizations

In an earlier paper on network monitoring [BCMG01], we proposed several optimizations of the co-networked monitoring algorithm based on the property that we were interested in monitoring. It turns out that for several classes of properties, buffering and/or loss on I-channels can be ignored. For some, we can significantly reduce the number of plausible sequences generated by the co-networked algorithm. In this section, we use some of these optimizations to try and speed up the monitoring of the TCP properties we have described earlier.

#### 10.8.1 Separating Properties

An effective technique to improve the performance of a monitor is to separate its properties and run them in parallel. The reason is that for each property, the amount of exploration needed in the co-networked algorithm differs. For instance, in the TCP analysis above, we were checking that implementations produce one ACK for every two DATA segments. If we separate this property from the rest of the TCP recognizer, we find that the performance of the remaining properties increases significantly (see Table 10.8.1). Both the memory usage and analysis time are halved when we eliminate this one error check. The NERL event slicing algorithm provides an automated technique for separating properties. If we want to separately check for one TCP error event \( E \), we can

<table>
<thead>
<tr>
<th>Trace</th>
<th>OS</th>
<th>Packets</th>
<th>Buffer</th>
<th>Unacked</th>
<th>Execution(s</th>
<th>Analysis(s)</th>
<th>Memory(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linux</td>
<td>4930</td>
<td>5</td>
<td>2</td>
<td>47.46</td>
<td>11.6</td>
<td>22.23</td>
</tr>
<tr>
<td>2</td>
<td>Linux</td>
<td>9804</td>
<td>5</td>
<td>2</td>
<td>95.36</td>
<td>23.21</td>
<td>21.89</td>
</tr>
<tr>
<td>3</td>
<td>Win XP</td>
<td>4752</td>
<td>5</td>
<td>5</td>
<td>55.2</td>
<td>12.38</td>
<td>14.62</td>
</tr>
<tr>
<td>4</td>
<td>Win XP</td>
<td>9474</td>
<td>5</td>
<td>10</td>
<td>113.96</td>
<td>25.23</td>
<td>13.16</td>
</tr>
<tr>
<td>5</td>
<td>Solaris</td>
<td>4930</td>
<td>5</td>
<td>5</td>
<td>47.46</td>
<td>13.92</td>
<td>18.11</td>
</tr>
<tr>
<td>6</td>
<td>Solaris</td>
<td>9814</td>
<td>5</td>
<td>5</td>
<td>95.36</td>
<td>27.75</td>
<td>21.64</td>
</tr>
</tbody>
</table>
slice the TCP recognizer for the output event $E$ and obtain a specialized executable recognizer. For all the optimizations described in this section, we first used the slicing tool to filter out unrelated properties and then modified the resulting recognizer.

### 10.8.2 Ignoring the Buffer

For some properties, we can effectively ignore the buffering and loss of input packets. For instance, consider the monotonic ACK property: *ACK sequence numbers are non-decreasing*. This property falls in a category we call P2: *Independent Outputs*. For a recognizer that only checks this property, there is only one monitoring channel, since the property does not care about data packets. Moreover, since this is a property of outputs - packets sent by a node - we can ignore buffering and loss.

Indeed, when we write a recognizer just for this property, we find that all the traces we have collected pass this test. So we did not have to take buffering into account as far as this property is concerned. The performance statistics for this property is shown in Table 10.8.2. Notice how both the analysis time and memory usage dip considerably.

When we first analyze a protocol, we always assume the absence of the input buffer. This is safe, because Lemmas 3.5 and 3.11 guarantee that if the implementation passes the zero-buffer zero-loss test then it will pass any test that assumes buffering. Only if the implementation fails this test do we need to account for buffering and loss.

### 10.8.3 Counting properties

We want to check that the TCP receiver acknowledges every two DATA packets. We noted earlier that some TCP implementations failed this property, and that it was taking up most of the time and memory of the TCP recognizer.

In this section, we shall analyze this property separately and write an optimal TCP recognizer for it. The property we are attempting to check is:

The receiver generates an ACK for at least every other data packet it receives.

---

Table 10.2: Time and memory usage for Remaining TCP Properties

<table>
<thead>
<tr>
<th>Trace</th>
<th>OS</th>
<th>Packets</th>
<th>Buffer</th>
<th>Execution(s)</th>
<th>Analysis(s)</th>
<th>Memory(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linux</td>
<td>4930</td>
<td>5</td>
<td>47.46</td>
<td>5.51</td>
<td>12.79</td>
</tr>
<tr>
<td>2</td>
<td>Linux</td>
<td>9804</td>
<td>5</td>
<td>95.36</td>
<td>11.23</td>
<td>12.74</td>
</tr>
<tr>
<td>3</td>
<td>Win XP</td>
<td>4752</td>
<td>5</td>
<td>55.2</td>
<td>5.84</td>
<td>4.29</td>
</tr>
<tr>
<td>4</td>
<td>Win XP</td>
<td>9474</td>
<td>5</td>
<td>113.96</td>
<td>11.79</td>
<td>4.29</td>
</tr>
<tr>
<td>5</td>
<td>Solaris</td>
<td>4930</td>
<td>5</td>
<td>47.46</td>
<td>6.15</td>
<td>5.85</td>
</tr>
<tr>
<td>6</td>
<td>Solaris</td>
<td>9814</td>
<td>5</td>
<td>95.36</td>
<td>12.19</td>
<td>5.85</td>
</tr>
</tbody>
</table>

Table 10.3: Time and memory usage for TCP Monotonic ACK property

<table>
<thead>
<tr>
<th>Trace</th>
<th>OS</th>
<th>Packets</th>
<th>Buffer</th>
<th>Execution(s)</th>
<th>Analysis(s)</th>
<th>Memory(MB)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linux</td>
<td>4930</td>
<td>5</td>
<td>47.46</td>
<td>0.23</td>
<td>0.071</td>
<td>Pass</td>
</tr>
<tr>
<td>2</td>
<td>Linux</td>
<td>9804</td>
<td>5</td>
<td>95.36</td>
<td>0.46</td>
<td>0.071</td>
<td>Pass</td>
</tr>
<tr>
<td>3</td>
<td>Win XP</td>
<td>4752</td>
<td>5</td>
<td>55.2</td>
<td>0.22</td>
<td>0.071</td>
<td>Pass</td>
</tr>
<tr>
<td>4</td>
<td>Win XP</td>
<td>9474</td>
<td>5</td>
<td>113.96</td>
<td>0.42</td>
<td>0.071</td>
<td>Pass</td>
</tr>
<tr>
<td>5</td>
<td>Solaris</td>
<td>4930</td>
<td>5</td>
<td>47.46</td>
<td>0.23</td>
<td>0.071</td>
<td>Pass</td>
</tr>
<tr>
<td>6</td>
<td>Solaris</td>
<td>9814</td>
<td>5</td>
<td>95.36</td>
<td>0.45</td>
<td>0.071</td>
<td>Pass</td>
</tr>
</tbody>
</table>
This property falls in a class we call P1: Counting properties. Such properties do not care about the content of packets, they only count the number of inputs and outputs. A general counting property checks that every output consumes between $c_{\text{min}}$ and $c_{\text{max}}$ inputs. For TCP, $c_{\text{min}} = 0$ and $c_{\text{max}} = 2$. 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 10.4. Here inputs (DATA) are indicated by $i_q$ and outputs (ACKs) are indicated by $o_d$. This algorithm maintains two integers, $buf_{\text{min}}$ and $buf_{\text{max}}$, representing the minimum and maximum number of inputs that are currently buffered on the channel between the monitor and the device under test. If $buf_{\text{min}}$ ever grows too large, it indicates that the particular $i_q, o_d$ 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 $buf_{\text{max}}$ ever becomes too small, it indicates that the particular $i_q, o_d$ 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 (raises an error event whenever $e$ is true), already takes buffering into account and so does not need the trace search algorithm. In effect, the algorithm in Table 10.4 is a specialization of trace search for a specific kind of recognizer. To prove the correctness of this algorithm, we carry out a reduction from the trace search algorithm in Chapter 7 (Table 7.2) to the counting algorithm. The reduction proceeds by defining a mapping between the two state spaces and showing that it is maintained when each algorithm takes a step in response to the same event. For illustration, we include the proof in Appendix D.2.

When we use this algorithm to check the TCP traces generated before, we find that the performance has improved manifold. Table 10.8.3 contains the time and memory usage for checking the TCP counting property.

### Table 10.4: Algorithm for checking P1

<table>
<thead>
<tr>
<th>Constants.</th>
<th>$c_{\text{max}}, c_{\text{min}}$ are integers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Type.</td>
<td>$buf_{\text{min}}$ and $buf_{\text{max}}$ are integers. $e$ is a boolean. $e$ is a boolean. Initially, $buf_{\text{min}} = buf_{\text{max}} = 0$ and $e = \text{false}$.</td>
</tr>
<tr>
<td>Event Handlers.</td>
<td>On receiving $i_q$: $buf_{\text{max}} = buf_{\text{max}} + 1, buf_{\text{min}} = buf_{\text{min}} + 1$ if ($buf_{\text{min}} &gt; B + c_{\text{max}}$) then $e = \text{true}$ else $buf_{\text{max}} = \min(buf_{\text{max}}, B + c_{\text{max}})$ $o_d$: if ($buf_{\text{max}} &lt; c_{\text{min}}$) then $e = \text{true}$ else $buf_{\text{max}} = \min(B, buf_{\text{max}} - c_{\text{min}}), buf_{\text{min}} = \max(0, buf_{\text{min}} - c_{\text{max}})$</td>
</tr>
</tbody>
</table>

If $e$ is true after executing either event handler, flag an error.
There is one quirk in the results - trace number 5 passed the counting test even though it generated this error in the general TCP recognizer. 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 acking 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 co-networked recognizers for special properties may be programmed. We have investigated such programming strategies for a variety of other property classes [BCM01] and aim to automate these strategies in a future version of NER.

10.9 Results

Errors. We analyzed several offline TCP traces generated by sessions between the Linux, Solaris and Windows XP TCP implementations. We checked these traces for violations of the TCP state machine, and violations of TCP’s reliability properties. We found that most of the traces produced by these implementations are correct.

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 correct TCP traces, such as those collected for the SMTP case study, we found that the co-networked monitor could filter out all the error events generated by the TCP recognizer. It could consistently distinguish between errors in the implementation and abnormalities caused by channel buffering and loss.

Performance. We found that the co-networked monitor for a complete TCP recognizer could just about keep pace with the speed of the session, and could handle at most 4 sessions at a time. This monitor processed 500 packets every second. On the other hand, when the TCP properties are separated out into different recognizers, the performance of the recognizer doubles. For some properties, we were able to write recognizers that handled more than 10000 packets every second and consumed an insignificant amount of memory.

Expressiveness. We demonstrate that the NER architecture and NERLMod language are flexible enough to incorporate co-networked monitoring. We apply the trace-search transformation on a TCP monitor represented by a NERLMod program, and automatically generate a co-networked...
monitor from co-located one. This emphasizes the value of using domain-specific languages for monitoring - it enables a structured modification of the execution semantics.
Chapter 11

Conclusions and Future Work

In this thesis, we have presented and demonstrated a programmable passive protocol monitoring system that we call Network Event Recognition. We presented languages, tools and techniques that form part of the NER suite, and used them to analyze several protocol implementations. In this chapter, we shall evaluate our results and describe some extensions and future work.

Expressiveness. We showed that NERL can be used to program recognizers for AODV monitoring, IP fragment reassembly, TCP stream reassembly, SMTP monitoring, and TCP state machine and reliability monitoring. While most of these recognizers involve 2 participants, we showed that NERL can be used to analyze multi-party protocols such as AODV. We showed that NERL\textsubscript{MOD} can put together and execute a stack of recognizers written in C, Lex and NERL. In this way, we showed that NER is expressive enough to analyze protocols at three different layers in the protocol stack.

Flexibility. We demonstrated that network event recognition is applicable to both network simulations (of AODV) and live network monitoring (for SMTP and TCP). For live networks, we demonstrated its effectiveness in bottleneck monitoring on top of reliable message streams, as well as co-networked monitoring. We showed that the NERL\textsubscript{MOD} language and associated tools could account for trace infidelities in the co-networked monitoring environment.

Correctness. We showed that incorrectly designed recognizers are quite prevalent for SMTP. We demonstrate that the model-checking tool can find suble errors in protocol recognizers. We believe that the combination of a formal semantics and model-checker can provide correctness guarantees for NERL programs.

Diagnostics. We demonstrate that the diagnostic tools and analysis techniques described for NER are adequate to interpret errors in all three protocols. For AODV, we show that these tools and techniques can be used to guide us directly to errors in the implementation even without inspecting the source code.

Efficiency. We demonstrated that NER can keep up with large AODV routing protocol simulations as well as several concurrent SMTP sessions. However, in the presence of significant trace infidelities, the performance of automatically generated TCP protocol monitors is not adequate and protocol specific optimizations become necessary.

In this manner, we have demonstrated a passive monitoring system that satisfies most of the requirements we outlines in the introduction and problem statement. On the other hand, more
development work is indicate for the implementations of the languages and tools, since these were shown to be inefficient in some cases. In addition, we would need to carry out several more case studies to fully evaluate the effectiveness of NER. In the following sections, we list other extensions and future work.

### 11.1 Network Monitoring Applications

In this thesis we developed techniques to address the protocol testing problem, but several of our techniques are also applicable in network monitoring. The NERL and NERL\textsubscript{MOD} languages can be used as programming or specification languages for a variety of monitoring tools.

**Surveillance** We have used NERL and NERL\textsubscript{MOD} to build an open-source email surveillance tool called OpenWarrants [BG02a]. This tool takes a warrant - a specification of whose email to capture and what part of the email to capture - and automatically generates a monitor that is guaranteed to capture only warranted emails. We provide this guarantee by using the model-checker to prove monitor correctness and by using the alarm tracing tool to filter out the parts of the email that are not warranted. On the other hand, prevalent surveillance tools such as Carnivore cannot provide these guarantees.

**Network Intrusion Detection** NERL and NERL\textsubscript{MOD} can also be used to write prototype intrusion detection systems. While the prototype written in NERL may not be fast enough to catch intrusions in a heavily loaded network it would be useful for testing the attack signature or intrusion specification. Moreover, the diagnostics provided by NERL alarm tracing feature would be invaluable in gathering attack signatures in a real network.

**Detecting Infidelities** Most network surveillance, intrusion detection and network management systems fail to account for trace infidelities. NER provides a generic monitoring algorithm that may be applicable in all these instances. Moreover, the co-networked monitoring algorithm could be used as a stand-alone infidelity detector that forms part of a network management system and raises alarms every time significant infidelities are detected.

### 11.2 Language Extensions

**Real-time Properties.** While we designed the NERL and NERL\textsubscript{MOD} languages to be aware of time, none of the case studies we presented analyze real-time protocol properties. Future case studies should incorporate real-time properties and evaluate the effectiveness of NER. In particular, it would be interesting to check whether such properties can be effectively monitored in the presence of infidelities.

**Regular Expressions and Temporal Logic.** Currently NERL events must be specified using properties of input events and the current state of the recognizer. For some properties, it is far more convenient to express events using regular expressions of temporal logic formulae. Such expressions and formulae can then be translated into NERL. For instance, the following event sequence is a convenient way to express usual TCP handshakes:

```
  event (Syn;Synack;Ack) -> TCP_HandShake;
```

This event can be translated to the following NERL fragment:
transition Init \rightarrow \{\text{tcp_handshake_state} = 0;\}
transition Syn \rightarrow \{\text{tcp_handshake_state} = 1;\}
transition Synack \rightarrow
\quad \text{OccurredWhen (tcp_handshake_state == 1) \rightarrow (tcp_handshake_state = 2;)}
\quad \text{event Ack}
\quad \text{OccurredWhen (tcp_handshake_state == 2) \rightarrow \text{TCP_Handshake}}
\quad \text{event Ack}
\quad \text{OccurredWhen (tcp_handshake_state == 2) \rightarrow (tcp_handshake_state = 0;)}

Such automatic translations would be even more convenient for complex expressions \((\text{Syn};\text{Synack};(\text{Data} |\text{EchoReply})\rightarrow\text{EchoRequest}!\rightarrow!\text{BadEchoReply})\). In addition, we might want to allow events expressed in temporal logic formulae that contain past safety operators. The past temporal operators are So-far \(p ([\text{-}p])\), Once \(p(\text{<}->p)\), p Since \(q, p\) \(\text{BackTo} q,\) and Previous \(p ([\text{-}p])\). For instance, the following formula checks that a reply matches a request sometime in the past

\begin{verbatim}
transition Init \rightarrow \{\text{once_echorequest} = false;\}
transition EchoRequest \rightarrow \{\text{once_echorequest} = true;\}
transition EchoReply
\quad \text{OccurredWhen !(\text{once_echo_request == true}) \rightarrow \text{BadEchoReply}}
\end{verbatim}

This can also be translated to a NERL fragment

11.3 Programming Strategies for Bottleneck Monitoring

In Chapter 10, we showed how the co-located monitor can be transformed automatically into a co-networked monitor. We also showed how to program specific properties to take input loss and buffering into account. We have earlier identified 11 classes of protocol properties for which such transformations are possible [BCMG01]. Figure 11.1 shows these property classes and inclusion relations between them. (An arrow from property class Q to property class R means that Q is in R.)

![Figure 11.1: Co-networked Monitorable Properties](image)

128
In a network monitoring context, where there are a large number of infidelities, it would be helpful to automate the transformations for all these classes. We would extend the NERL\textsubscript{MOD} compiler with an automatic transformation for each class, and an algorithm for checking whether a property lies in the class.

In addition, for bottleneck monitoring, we need to take other infidelities such as output loss and buffering into account as well. We would like to study monitoring algorithms that also take these infidelities into account.
Appendix A

Ping Event Recognition

A.1 The Ping Recognizer: ping.nerl

```
Recognizer Ping =

typedef {
    int ip_src;
    int ip_dst;
    int icmp_type;
    int icmp_code;
    int icmp_id;
    int icmp_seq;
    string icmp_data;
} ty_echo;

typedef {
    int ip_src;
    int ip_dst;
    int icmp_id;
} ty_ping;

input event ty_ping Init;
input event ty_echo EchoRequest;
input event ty_echo EchoReply;

output event string IsAlive;
output event string BadEchoRequest;
output event string BadEchoReply;
output event bool Done;

event bool MyEchoRequest;
event bool MyEchoReply;

int ip_src;
int ip_dst;
int icmp_id;
```
#define CLEAR 0
#define WAIT 1
#define DONE 2

int status;

int icmp_seq;
string icmp_data;

transition Init(i) -> {
    ip_src = i.ip_src;
    ip_dst = i.ip_dst;
    icmp_id = i.icmp_id;
    status = CLEAR;
}

event EchoRequest(p)
    OccurredWhen
        ((p.ip_src == ip_src) &&
        (p.ip_dst == ip_dst) &&
        (p.icmp_id == icmp_id) &&
        (p.icmp_code == 0) &&
        (p.icmp_type == 8)) -> MyEchoRequest;

event EchoReply(p)
    OccurredWhen
        ((p.ip_src == ip_dst) &&
        (p.ip_dst == ip_src) &&
        (p.icmp_id == icmp_id) &&
        (p.icmp_code == 0) &&
        (p.icmp_type == 0)) -> MyEchoReply;

transition (EchoRequest(e) & MyEchoRequest)
    OccurredWhen
        (status == CLEAR) -> {
            icmp_seq = e.icmp_seq;
            icmp_data = e.icmp_data;
            status = WAIT;
        }

event (EchoRequest(e) & MyEchoRequest)
    OccurredWhen
        (status != CLEAR) -> BadEchoRequest(b)
            WithAttributes
            {b = "Multiple Echo Requests"};

event (EchoReply(e) & MyEchoReply)
    OccurredWhen
        ((status == WAIT) &&
        (e.icmp_seq == icmp_seq) &&
        (e.icmp_data == icmp_data)) -> IsAlive(a)
event (EchoReply(e) & MyEchoReply)
OccurredWhen
((status != WAIT) ||
(e.icmp_seq != icmp_seq) ||
(e.icmp_data != icmp_data)) → BadEchoReply(b)
WithAttributes
{ b="Incorrect_Echo_Reply" }

event (IsAlive | BadEchoReply | BadEchoRequest) → Done(d)
WithAttributes
{ d = true }

transition IsAlive(a) → { status = DONE };
EndRecognizer;

A.2 Ping Recognizer translated to C: ping.c

#include "includes.h"

/* Type Definitions */

#define CELL(t) struct {int changed; t value; t oldvalue;}
typedef CELL(bool) boolcell;
typedef CELL(int) intcell;
typedef CELL(double) doublecell;
typedef CELL(string) stringcell;

#define EVENT_TY(a) struct {bool flag; double time; a}
typedef struct {
    intcell* ip_src;
    intcell* ip_dst;
    intcell* icmp_type;
    intcell* icmp_code;
    intcell* icmp_id;
    intcell* icmp_seq;
    stringcell* icmp_data;
} ty_echo;

typedef struct {
    intcell* ip_src;
    intcell* ip_dst;
    intcell* icmp_id;
} ty_ping;
/ * State Declarations */

typedef struct {
    double now;
    int thisround;

    intcell* ip_src;
    intcell* ip_dst;
    intcell* icmp_id;
    intcell* status;
    intcell* icmp_seq;
    stringcell* icmp_data;
} Ping.state;

/* Control Block */

typedef struct {

    /* Input Event Declarations */

    struct {
        EVENT_TY(ty_ping* attrib;) Init;
        EVENT_TY(ty_echo* attrib;) EchoRequest;
        EVENT_TY(ty_echo* attrib;) EchoReply;
    }* inputs;

    Ping.state* state;

    /* Output Event Declarations */

    struct {
        EVENT_TY(stringcell* attrib;) IsAlive;
        EVENT_TY(stringcell* attrib;) BadEchoRequest;
        EVENT_TY(stringcell* attrib;) BadEchoReply;
        EVENT_TY(boolcell* attrib;) Done;
    }* outputs;

    /* Local Event Declarations */

    struct {
        EVENT_TY(boolcell* attrib;) MyEchoRequest;
        EVENT_TY(boolcell* attrib;) MyEchoReply;
    }* locals;

    } Ping.cb;

    /* Recognizer Function */

    void Ping.recognize(Ping_cb *cb) {

        /* State Allocation and Initialization */

        if (cb->inputs->Init.flag == true) {
            cb->state = (Ping.state*) malloc(sizeof(Ping.state));
            cb->state->thisround = 0;
            cb->state->now = 0.0;
        }
cb->state->ip_src = (intcell*) malloc(sizeof(intcell));
cb->state->ip_src->changed = cb->state->thisround;
cb->state->ip_src->value = 0;
cb->state->ip_src->oldvalue = 0;

cb->state->ip_dst = (intcell*) malloc(sizeof(intcell));
cb->state->ip_dst->changed = cb->state->thisround;
cb->state->ip_dst->value = 0;
cb->state->ip_dst->oldvalue = 0;

cb->state->icmp_id = (intcell*) malloc(sizeof(intcell));
cb->state->icmp_id->changed = cb->state->thisround;
cb->state->icmp_id->value = 0;
cb->state->icmp_id->oldvalue = 0;

cb->state->status = (intcell*) malloc(sizeof(intcell));
cb->state->status->changed = cb->state->thisround;
cb->state->status->value = 0;
cb->state->status->oldvalue = 0;

cb->state->icmp_seq = (intcell*) malloc(sizeof(intcell));
cb->state->icmp_seq->changed = cb->state->thisround;
cb->state->icmp_seq->value = 0;
cb->state->icmp_seq->oldvalue = 0;

cb->state->icmp_data = (stringcell*) malloc(sizeof(stringcell));
cb->state->icmp_data->changed = cb->state->thisround;
cb->state->icmp_data->value = "";
cb->state->icmp_data->oldvalue = "";
}

/* Round Begins */

cb->state->thisround++;

/* Local and Output Event Initialization */

cb->locals->MyEchoReply.flag = false;
{boolcell* MyEchoReply_init;
MyEchoReply_init = cb->locals->MyEchoReply.attrib;
MyEchoReply_init->value = false;
}

cb->locals->MyEchoRequest.flag = false;
{boolcell* MyEchoRequest_init;
MyEchoRequest_init = cb->locals->MyEchoRequest.attrib;
MyEchoRequest_init->value = false;
}

cb->outputs->Done.flag = false;
{boolcell* Done_init;
Done_init = cb->outputs->Done.attrib;
Done_init->value = false;
}
cb->outputs->BadEchoReply.flag = false;
{ stringcell* BadEchoReply_init;
BadEchoReply_init = cb->outputs->BadEchoReply.attrib;
BadEchoReply_init->value = "";
}

cb->outputs->BadEchoRequest.flag = false;
{ stringcell* BadEchoRequest_init;
BadEchoRequest_init = cb->outputs->BadEchoRequest.attrib;
BadEchoRequest_init->value = "";
}

cb->outputs->IsAlive.flag = false;
{ stringcell* IsAlive_init;
IsAlive_init = cb->outputs->IsAlive.attrib;
IsAlive_init->value = "";
}

/* Transitions and Event Definitions */

if (cb->inputs->Init.flag == true) {
  ty_ping* i;
  i = (cb->inputs->Init.attrib);
  {cb->state->ip_src->changed = cb->state->thisround;
   cb->state->ip_src->oldvalue = cb->state->ip_src->value;
   cb->state->ip_src->value = (i)->ip_src->value;
   cb->state->ip_dst->changed = cb->state->thisround;
   cb->state->ip_dst->oldvalue = cb->state->ip_dst->value;
   cb->state->ip_dst->value = (i)->ip_dst->value;
   cb->state->icmp_id->changed = cb->state->thisround;
   cb->state->icmp_id->oldvalue = cb->state->icmp_id->value;
   cb->state->icmp_id->value = (i)->icmp_id->value;
   cb->state->status->changed = cb->state->thisround;
   cb->state->status->oldvalue = cb->state->status->value;
   cb->state->status->value = 0;
  }
}

if (cb->inputs->EchoRequest.flag == true) {
  ty_echo* p;
  p = (cb->inputs->EchoRequest.attrib);
  if ((((p)->ip_src->value) == 
       ((cb->state->ip_src->changed == cb->state->thisround ?
         cb->state->ip_src->oldvalue:cb->state->ip_src->value)))
       &
       (((p)->ip_dst->value) ==
         ((cb->state->ip_dst->changed == cb->state->thisround ?
           cb->state->ip_dst->oldvalue:cb->state->ip_dst->value)))
       &
       (((p)->icmp_id->value) ==
         ((cb->state->icmp_id->changed == cb->state->thisround ?
           cb->state->icmp_id->oldvalue:cb->state->icmp_id->value)))
       &
       (((p)->icmp_code->value) == (0))
       &
       (((p)->icmp_type->value) == (8)))
  { cb->locals->MyEchoRequest.flag = true;
  }
}
if (cb->inputs->EchoReply.flag == true) {
    ty_echo* p;
    p = (cb->inputs->EchoReply.attrib);
    if ((((((p)->ip_src->value) ==
        (cb->state->ip_dst->changed == cb->state->thisround ?
        cb->state->ip_dst->oldvalue : cb->state->ip_dst->value))) &&
        (((p)->ip_src->value) ==
        (cb->state->ip_src->changed == cb->state->thisround ?
        cb->state->ip_src->oldvalue : cb->state->ip_src->value)))) &&
        (((p)->icmp_id->value) ==
        (cb->state->icmp_id->changed == cb->state->thisround ?
        cb->state->icmp_id->oldvalue : cb->state->icmp_id->value))) &&
        (((p)->icmp_code->value) == (0)) &&
        (((p)->icmp_type->value) == (0)))
    { cb->locals->MyEchoReply.flag = true; }
}

if (cb->inputs->EchoRequest.flag == true) {
    ty_echo* e;
    e = (cb->inputs->EchoRequest.attrib);
    if (cb->locals->MyEchoRequest.flag == true) {
        if (((cb->state->status->changed == cb->state->thisround ?
            cb->state->status->oldvalue : cb->state->status->value)) ==
            (0))
        { cb->state->icmp_seq->changed = cb->state->thisround;
            cb->state->icmp_seq->oldvalue = cb->state->icmp_seq->value;
            cb->state->icmp_seq->value = (e)->icmp_seq->value;
            cb->state->icmp_data->changed = cb->state->thisround;
            cb->state->icmp_data->oldvalue = cb->state->icmp_data->value;
            string_copy(&cb->state->icmp_data->value,(e)->icmp_data->value);
            cb->state->status->changed = cb->state->thisround;
            cb->state->status->oldvalue = cb->state->status->value;
            cb->state->status->value = 1;
        }
    }
}

if (cb->inputs->EchoRequest.flag == true) {
    ty_echo* e;
    e = (cb->inputs->EchoRequest.attrib);
    if (cb->locals->MyEchoRequest.flag == true) {
        if (((cb->state->status->changed == cb->state->thisround ?
            cb->state->status->oldvalue : cb->state->status->value)) !=
            (0))
        { cb->outputs->BadEchoRequest.flag = true;
            { stringcell* b;
            b = cb->outputs->BadEchoRequest.attrib;
            { b->value = "Multiple Echo Requests"; }
        }
    }
}
if (cb->inputs->EchoReply.flag == true) {
    ty_echo* e;
    e = (cb->inputs->EchoReply.attrib);
    if (cb->locals->MyEchoReply.flag == true) {
        if (((cb->state->status->changed == cb->state->thisround ?
                cb->state->status->oldvalue : cb->state->status->value)) ==
            (1) &&
            ((e)->icmp_seq->value) ==
            ((cb->state->icmp_seq->changed == cb->state->thisround ?
                cb->state->icmp_seq->oldvalue : cb->state->icmp_seq->value)) &&
            ((e)->icmp_data->value) ==
            ((cb->state->icmp_data->changed == cb->state->thisround ?
                cb->state->icmp_data->oldvalue : cb->state->icmp_data->value))))
            { cb->outputs->IsAlive.flag = true;
              stringcell* a;
              a = cb->outputs->IsAlive.attrib;
              a->value = (e)->icmp_data->value;}
    }
}

if (cb->inputs->EchoReply.flag == true) {
    ty_echo* e;
    e = (cb->inputs->EchoReply.attrib);
    if (cb->locals->MyEchoReply.flag == true) {
        if (((cb->state->status->changed == cb->state->thisround ?
                cb->state->status->oldvalue : cb->state->status->value)) !=
            (1) ||
            ((e)->icmp_seq->value) !=
            ((cb->state->icmp_seq->changed == cb->state->thisround ?
                cb->state->icmp_seq->oldvalue : cb->state->icmp_seq->value)) ||
            ((e)->icmp_data->value) !=
            ((cb->state->icmp_data->changed == cb->state->thisround ?
                cb->state->icmp_data->oldvalue : cb->state->icmp_data->value)))
            { cb->outputs->BadEchoReply.flag = true;
              stringcell* b;
              b = cb->outputs->BadEchoReply.attrib;
              b->value = "Incorrect_EchoReply";}
    }
}

if (cb->outputs->IsAlive.flag == true) {
    cb->outputs->Done.flag = true;
    boolcell* d;
    d = cb->outputs->Done.attrib;
    d->value = true;}
else
    if (cb->outputs->BadEchoReply.flag == true) {
        cb->outputs->Done.flag = true;
        boolcell* d;
        d = cb->outputs->Done.attrib;
        d->value = true;}
}
} else  
  if (cb->outputs->BadEchoRequest.flag == true) {  
    cb->outputs->Done.flag = true;  
    boolcell* d;  
    d= cb->outputs->Done.attrib;  
    {d->value = true;  
  }
}

if (cb->outputs->IsAlive.flag == true) {
  stringcell* a;
  a= (cb->outputs->IsAlive.attrib);
  {cb->state->status->changed = cb->state->thisround;
   cb->state->status->oldvalue = cb->state->status->value;
   cb->state->status->value = 2;
  }
}

A.3 Ping Monitor Stack in NERL_MOD: pingmod.nerl

begin

typedef {
  string filename;
} file;

signature PCap_Sig = {
  input event file Init;
  output event Ty_iPPkt IP;
  output event bool Done;
}

signature ICMP_Parse_Sig = {
  input event bool Init;
  input event Ty_iPPkt IP;
  output event Ty_echo ICMP_Echo;
}

signature Ping_Sig = {
  input event Ty_ping Init;
  input event Ty_echo EchoRequest;
  input event Ty_echo EchoReply;
  output event Ty_echo IsAlive;
  output event string BadEchoRequest;
  output event string BadEchoReply;
  output event bool Done;
}
Recognize PCap_Sig PCap;
Recognize ICMP_Parse_Sig ICMP_Parse;
Recognize Ping_Sig Ping;
Recognize Ping_Sig Ping_Multiple;

instance PCap P WithAttributes
    {init = "icmp.pcap"};
instance ICMP_Parse I WithAttributes
    {init = true};
instance Ping E[Typ_ping, c] WithAttributes
    {init = c};

event P.IP(p) OccurredWhen (p.ip_p == 1) ->
    I.IP(q) WithAttributes {q = p};

event I.ICMP.Echo(e) OccurredWhen (e.type == 8) ->
    E[x].EchoRequest(p) WithIndex {
        x.ip_src = e.ip_src;
        x.ip_dst = e.ip_dst;
        x.icmp_id = e.icmp_id
    } WithAttributes {p = e}

event I.ICMP.Echo(e) OccurredWhen (e.type == 0) ->
    E[x].EchoReply(p) WithIndex {
        x.ip_dst = e.ip_src;
        x.ip_src = e.ip_dst;
        x.icmp_id = e.icmp_id
    } WithAttributes {p = e}

event E[x].BadEchoRequest(b) -> PRINT;

event E[x].BadEchoReply(b) -> PRINT;

event E[x].IsAlive(a) -> PRINT;

end

A.4 Ping Monitor Stack Translated to C: pingmod.c

/* Index Type Definitions */
typedef struct {
    stringcell* filename;
} file;

/* Signatures */
typedef struct {
    EVENT_TYP(file) Init;
}
void* state;
EVENT_TY(ty_ippkt) IP;
EVENT_TY(bool) Done;
} PCap_Sig;

typedef struct {
  EVENT_TY(bool) Init;
  EVENT_TY(ty_ippkt) IP;
  void* state;
  EVENT_TY(ty_echo) ICMP_Echo;
} ICMP_Parse_Sig;

/* Executable monitor stack – Main Module */

void main() {

  /* Instance Allocation */

  PCap_Sig* P_cb = (PCap_Sig*) malloc(sizeof(PCap_Sig));
  ICMP_Parse_Sig* I_cb = (ICMP_Parse_Sig*) malloc(sizeof(ICMP_Parse_Sig));
  HashTable* E_table;
  Ping_Sig* E_cb = (Ping_Sig*) malloc(sizeof(Ping_Sig));
  E_table = hash_table_new(ty_ping_hash, ty_ping_cmp);

  /* Single Instance Initialization */

  P_cb->inputs->Init.flag = false;
  P_cb->inputs->Init.flag = true;
  {file* init;
   copy_string(init->filename->value,"icmp.pcap");
   P_cb->inputs->Init.attrib = init;
   PCap_recognize(P_cb);
  }

  I_cb->inputs->Init.flag = false;
  I_cb->inputs->IP.flag = false;
  I_cb->inputs->Init.flag = true;
  {boolcell* init;
   init->value = true;
   I_cb->inputs->Init.attrib = init;
   ICMP_Parse_recognize(I_cb);
  }

  /* Packet Capture Loop */

  while (1) {

    /* Capture Module Execution */

    P_cb->inputs->Init.flag = false;
    PCap_recognize(P_cb);

    if (P_cb->outputs->Done.flag == true) break;

  }

}
if (P.cb->outputs->IP.flag == true) {
    ty_ip_pkt* p;
    p = (P.cb->outputs->IP.attrib);
    if (((p)->ip_p->value) == 1) {
        /* ICMP_Parse Module Execution */

        I_cb->inputs->Init.flag = false;
        I_cb->inputs->IP.flag = false;
        I_cb->inputs->IP.flag = true;
        (ty_ip_pkt*) q;
        q = p;
        I_cb->inputs->IP.attrib = q;
        ICMP_Parse_recognize(I_cb);
    }

    if (I_cb->outputs->ICMP.Echo.flag == true) {
        /* Ping Module Execution */

        ty_echo* e;
        e = (I_cb->outputs->ICMP.Echo.attrib);

        if (((e)->icmp_type->value) == 8) {
            ty_ping* x;
            x = (ty_ping*) malloc(sizeof(ty_ping));
            (x)->ip_src->value = (e)->ip_src->value;
            (x)->ip_dst->value = (e)->ip_dst->value;
            (x)->icmp_id->value = (e)->icmp_id->value;

            if ((E_cb = hash_table_lookup(E_table, x)) == NULL) {
                E_cb->inputs->Init.flag = false;
                E_cb->inputs->EchoRequest.flag = false;
                E_cb->inputs->EchoReply.flag = false;
                E_cb->inputs->Init.flag = true;

                (ty_ping*) c;
                c = x;

                (ty_ping* init;
                init = c;
                E_cb->inputs->Init.attrib = init;
                Ping_recognize(E_cb);
            }

            hash_table_insert(E_table, c, E_cb);
        }
    }
}

E_cb->inputs->Init.flag = false;
E_cb->inputs->EchoRequest.flag = false;
E_cb->inputs->EchoReply.flag = false;
E_cb->inputs->EchoRequest.flag = true;
{ty_echo* p;
 p = e;
 E_cb->inputs->EchoRequest.attrib = p;
Ping_recognize(E_cb);
}

if (E_cb->outputs->BadEchoReply.flag == true) {
  printf("%f: Event BadEchoReply has occurred, ",
      E_cb->outputs->BadEchoReply.time);
  printf("BadEchoReply:");
  print_string(E_cb->outputs->BadEchoReply.attrib->value);
  printf("\n");
}

if (E_cb->outputs->BadEchoRequest.flag == true) {
  printf("%f: Event BadEchoRequest has occurred, ",
      E_cb->outputs->BadEchoRequest.time);
  printf("BadEchoRequest:");
  print_string(E_cb->outputs->BadEchoRequest.attrib->value);
  printf("\n");
}

if (E_cb->outputs->IsAlive.flag == true) {
  printf("%f: Event IsAlive has occurred, ",
      E_cb->outputs->IsAlive.time);
  printf("IsAlive:");
  print_string(E_cb->outputs->IsAlive.attrib->value);
  printf("\n");
}

if (E_cb->outputs->Done.flag == true) {
  hash_table_remove(E_table,x);
  E_free(E_cb);
  free(x);
}

if (((e)->icmp_type->value) == 0) {
  ty_ping* x;
  x = (ty_ping*) malloc(sizeof(ty_ping));
  (x)->ip_dst->value = (e)->ip_src->value;
  (x)->ip_src->value = (e)->ip_dst->value;
  (x)->icmp_id->value = (e)->icmp_id->value;

  if ((E_cb = hash_table_lookup(E_table,x)) == NULL) {
    E_cb->inputs->Init.flag = false;
    E_cb->inputs->EchoRequest.flag = false;
    E_cb->inputs->EchoReply.flag = false;
    E_cb->inputs->Done.flag = true;
    {ty_ping* c;
     c = x;
    }
    init = c;
    E_cb->inputs->Init.attrib = init;
    Ping_recognize(E_cb);
  }
  hash_table_insert(E_table,c,E_cb);
}
A.5 Ping Promela Model: ping.spin

We include two files: pingmain.spin containing models for the sender and receiver, and ping.spin containing the Promela model generated from ping.nerl.
#include "ping.spin"

#define MAXSESSIONS 1
#define MAXPING 10

chan SOut = [1] of {ty_echo};
chan SIn = [1] of {ty_echo};
chan ROut = [1] of {ty_echo};
chan RIn = [1] of {ty_echo};

bool isalive;
bool recisalive;
bool recbadechorequest;
bool recbadechoreply;
bool recdone;

proctype Ping_sender(int me; int dst; int id) {
  ty_echo e;
  ty_echo r;
  int seq;
  int data;

  data = 10;
  seq = 0;

  e.ip_src.value = me;
  e.ip_dst.value = dst;
  e.icmp_id.value = id;
  e.icmp_code.value = 0;
  e.icmp_type.value = 8;
  e.icmp_data.value = data;
  do :: seq <= MAXPING ->
    seq++;
  :: break;
  od;

  e.icmp_seq.value = seq;
  SOut!e;
  SIn?r ->
    if :: (r.icmp_seq.value == seq) &&
      (r.icmp_data.value == data) -> isalive = true;
    :: else -> skip
    fi
}

proctype Ping_receiver(int me) {
  int seq;
  ty_echo e;
RIn?e ->
  if
    :: e.ip.dst.value == me ->
      e.ip.dst.value = e.ip.src.value;
      e.ip.src.value = me;
      e.icmp.type.value = 0;
      ROut!e;
    :: else -> skip
  fi
}

init {
  ty.ping p;
  ty.echo r;
  bool cell d;
  int cell a;

  isalive = false;
  recisalive = false;
  recbadechorequest = false;
  recbadechoreply = false;

  cb_table[0].inputs.Init = Init_table[0];
  cb_table[0].inputs.EchoRequest = EchoRequest_table[0];
  cb_table[0].inputs.EchoReply = EchoReply_table[0];

  p.ip.src.value = 1;
  p.ip.dst.value = 2;
  p.icmp_id.value = 3000;
  cb_table[0].inputs.Init!p;
  run Ping.recognize(0);
  timeout;

  run Ping.sender(1,2,3000);
  run Ping.receiver(2);

  SOut?r -> {
    cb_table[0].inputs.EchoRequest!r;
    RIn!r;
    timeout;
    if
      :: cb_table[0].outputs.BadEchoRequest?[a] -> recbadechorequest = true;
      :: else -> skip
    fi;
    ROut?r -> {
      cb_table[0].inputs.EchoReply!r;
      SIn!r;
      timeout;
      if
        :: cb_table[0].outputs.IsAlive?[a] -> recisalive = true;
        :: else -> skip;
        :: else -> skip;
      }
fi;
if :: cb_table[0].outputs.BadEchoReply?[a] \rightarrow recbadechoreply = true;
:: else \rightarrow skip;
fi;
if :: cb_table[0].outputs.Done?[d] \rightarrow recdone = true;
:: else \rightarrow skip
fi
}

/* Properties checked:
Correct meta event:
<>(recisalive == true)
[[]((recisalive == true) \rightarrow <>(isalive == true))]

No false positives:
[][(rechadechorequest == false)
[][(rechadechorequest == false)
*/

/***************************************************************************************
/* ping.spin */
/***************************************************************************************

/* Type Definitions */

typedef intcell { 
    int changed;
    int value;
    int oldvalue;
};

typedef boolcell { 
    int changed;
    bool value;
    bool oldvalue;
};

typedef ty_echo { 
    intcell ip_src;
    intcell ip_dst;
    intcell icmp_id;
    intcell icmp_code;
    intcell icmp_type;
    intcell icmp_seq;
    intcell icmp_data;
};

typedef ty_ping { 
    intcell ip_src;
    intcell ip_dst;
    intcell icmp_id;

146
typedef Ping_state {
    int now;
    int thisround;
    intcell ip_src;
    intcell ip_dst;
    intcell icmp_id;
    #define CLEAR 0
    #define WAIT 1
    #define DONE 2
    intcell status;
    intcell icmp_seq;
    intcell icmp_data;
};

typedef Ping_inputs {
    chan Init;
    chan EchoRequest;
    chan EchoReply;
};

typedef Ping_outputs {
    chan IsAlive;
    chan BadEchoRequest;
    chan BadEchoReply;
    chan Done;
};

typedef Ping_locals {
    chan MyEchoRequest;
    chan MyEchoReply;
};

/* Control Block */

typedef Ping_sig {
    Ping_inputs inputs;
    Ping_outputs outputs;
    Ping_state state;
    Ping_locals locals;
};

/* Instance and Event Tables */

Ping_sig cb_table[MAXSESSIONS];

chan Init_table[MAXSESSIONS] = [1] of { ty_ping };
chan EchoRequest_table[MAXSESSIONS] = [1] of {ty_echo};
chan EchoReply_table[MAXSESSIONS] = [1] of {ty_echo};
chan IsAlive_table[MAXSESSIONS] = [1] of {intcell};
chan BadEchoRequest_table[MAXSESSIONS] = [1] of {intcell};
chan BadEchoReply_table[MAXSESSIONS] = [1] of {intcell};
chan Done_table[MAXSESSIONS] = [1] of {boolcell};
chan MyEchoRequest_table[MAXSESSIONS] = [1] of {boolcell};
chan MyEchoReply_table[MAXSESSIONS] = [1] of {boolcell};

/∗ Ping Monitor Process ∗/

proc type Ping_recognize ( int me ) {
    ty_ping i;
    ty_echo p;
    ty_echo e;
    intcell d;
    boolcell a;
    intcell b;

    do
        : : ( cb_table[me].inputs.Init?[i] ||
            cb_table[me].inputs.EchoRequest?[p] ||
            cb_table[me].inputs.EchoReply?[p] ) ->

    /∗ Output and Local Event Channel Allocation ∗/

    if
        : : cb_table[me].inputs.Init?[i] ->
            cb_table[me].outputs.IsAlive = IsAlive_table[me];
            cb_table[me].outputs.BadEchoRequest = BadEchoRequest_table[me];
            cb_table[me].outputs.BadEchoReply = BadEchoReply_table[me];
            cb_table[me].outputs.Done = Done_table[me];
            cb_table[me].locals.MyEchoRequest = MyEchoRequest_table[me];
            cb_table[me].locals.MyEchoReply = MyEchoReply_table[me];
        : : else -> skip;
    fi;

    /∗ Round Begins ∗/

    cb_table[me].state.thisround++;

    /∗ Local Event Initialization ∗/

    do
        : : cb_table[me].locals.MyEchoRequest?[d] ->
            cb_table[me].locals.MyEchoRequest?d
        : : cb_table[me].locals.MyEchoReply?[d] ->
            cb_table[me].locals.MyEchoReply?d;
        : : else -> break;
    od;

    /∗ Transitions and Event Definitions ∗/
if cb_table[me].inputs.Init?[i] -> cb_table[me].inputs.Init?<i>;
    cb_table[me].state.ip_src.changed = cb_table[me].state.thisround;
    cb_table[me].state.ip_src.oldvalue = cb_table[me].state.ip_src.value;
    cb_table[me].state.ip_src.value = i.ip_src.value;
    cb_table[me].state.ip_dst.changed = cb_table[me].state.thisround;
    cb_table[me].state.ip_dst.oldvalue = cb_table[me].state.ip_dst.value;
    cb_table[me].state.ip_dst.value = i.ip_dst.value;
    cb_table[me].state.icmp_id.changed = cb_table[me].state.thisround;
    cb_table[me].state.icmp_id.oldvalue = cb_table[me].state.icmp_id.value;
    cb_table[me].state.icmp_id.value = i.icmp_id.value;
    cb_table[me].state.status.changed = cb_table[me].state.thisround;
    cb_table[me].state.status.oldvalue = cb_table[me].state.status.value;
    cb_table[me].state.status.value = 0;
    else -> skip;
fi;
if cb_table[me].inputs.EchoRequest?[p] ->
    cb_table[me].inputs.EchoRequest?<p>;
    if (((((p.ip_src.value) ==
        ((cb_table[me].state.ip_src.changed ==
            cb_table[me].state.thisround ->
            cb_table[me].state.ip_src.oldvalue :
            cb_table[me].state.ip_src.value))) &&
        ((p.ip_dst.value) ==
            ((cb_table[me].state.ip_dst.changed ==
                cb_table[me].state.thisround ->
                cb_table[me].state.ip_dst.oldvalue :
                cb_table[me].state.ip_dst.value))) &&
        ((p.icmp_id.value) ==
            ((cb_table[me].state.icmp_id.changed ==
                cb_table[me].state.thisround ->
                cb_table[me].state.icmp_id.oldvalue :
                cb_table[me].state.icmp_id.value))) &&
        ((p.icmp_code.value) == (0)) &&
        (p.icmp_type.value) == (8) ) ) ) )
        d.value = true;
    cb_table[me].locals.MyEchoRequest!d
    else -> skip;
fi;
else -> skip;
fi;
if cb_table[me].inputs.EchoReply?[p] ->
    cb_table[me].inputs.EchoReply?<p>;
    if (((((p.ip_src.value) ==
        ((cb_table[me].state.ip_dst.changed ==
            cb_table[me].state.thisround ->
            cb_table[me].state.ip_dst.oldvalue :
            cb_table[me].state.ip_dst.value))) &&
        ((p.ip_dst.value) ==
            ((cb_table[me].state.ip_return ==
                cb_table[me].state.return_value :
                cb_table[me].state.ip_return.value))) &&
        ((p.icmp_code.value) == (0)) &&
        (p.icmp_type.value) == (8) ) ) )
        d.value = true;
    cb_table[me].locals.MyEchoReply!d
    else -> skip;
fi;
else -> skip;
fi;

149
((cb_table[me].state.ip_src.changed ==
    cb_table[me].state.thisround ->
    cb_table[me].state.ip_src.oldvalue :
    cb_table[me].state.ip_src.value))))) &&
((p.icmp_id.value) ==
((cb_table[me].state.icmp_id.changed ==
    cb_table[me].state.thisround ->
    cb_table[me].state.icmp_id.oldvalue :
    cb_table[me].state.icmp_id.value)))) &&
((p.icmp_code.value) == (0)) &&
((p.icmp_type.value) == (0)) ->
d.value = true;
    cb_table[me].locals.MyEchoReply!d
:: else -> skip;
fi
:: else -> skip;
fi;

if :: cb_table[me].inputs.EchoRequest?[e] ->
    cb_table[me].inputs.EchoRequest?[e];
if :: cb_table[me].locals.MyEchoRequest?[d] ->
    if :: (((cb_table[me].state.status.changed ==
        cb_table[me].state.thisround ->
        cb_table[me].state.status.oldvalue :
        cb_table[me].state.status.value)) == (0)) ->
    cb_table[me].state.icmp_seq.changed = cb_table[me].state.thisround;
    cb_table[me].state.icmp_seq.oldvalue =
        cb_table[me].state.icmp_seq.value;
    cb_table[me].state.icmp_seq.value = e.icmp_seq.value;
    cb_table[me].state.icmp_data.changed = cb_table[me].state.thisround;
    cb_table[me].state.icmp_data.oldvalue =
        cb_table[me].state.icmp_data.value;
    cb_table[me].state.icmp_data.value = e.icmp_data.value;
    cb_table[me].state.status.changed = cb_table[me].state.thisround;
    cb_table[me].state.status.oldvalue =
        cb_table[me].state.status.value;
    cb_table[me].state.status.value = 1;
:: else -> skip;
fi;
:: else -> skip;
fi;
:: else -> skip;
fi;

if :: cb_table[me].inputs.EchoRequest?[e] ->
    cb_table[me].inputs.EchoRequest?[e];
if :: cb_table[me].locals.MyEchoRequest?[d] ->
    if :: (((cb_table[me].state.status.changed ==

if cb_table[me].state.thisround ->
  cb_table[me].state.status.oldvalue :
  cb_table[me].state.status.value)) != (0)) ->
  b.value = 22;
  cb_table[me].outputs.BadEchoRequest!b;
:: else -> skip;
fi
:: else -> skip;
fi
:: else -> skip;
fi

if cb_table[me].inputs.EchoReply?[e] ->
  cb_table[me].inputs.EchoReply?[e>;
if :: cb_table[me].locals.MyEchoReply?[d] ->
  :: (((cb_table[me].state.status.changed ==
    cb_table[me].state.thisround ->
    cb_table[me].state.status.oldvalue :
    cb_table[me].state.status.value)) == (1)) &&
    ((e.icmp_seq.value) ==
      ((cb_table[me].state.icmp_seq.changed ==
        cb_table[me].state.thisround ->
        cb_table[me].state.icmp_seq.oldvalue :
        cb_table[me].state.icmp_seq.value))) &&
    ((e.icmp_data.value) ==
      ((cb_table[me].state.icmp_data.changed ==
        cb_table[me].state.thisround ->
        cb_table[me].state.icmp_data.oldvalue :
        cb_table[me].state.icmp_data.value)))
  ->
    a.value = e.icmp_data.value;
  cb_table[me].outputs.IsAlive!a
:: else -> skip
fi
:: else -> skip
fi
:: else -> skip
fi

if cb_table[me].inputs.EchoReply?[e] ->
  cb_table[me].inputs.EchoReply?[e>;
if :: cb_table[me].locals.MyEchoReply?[d] ->
  :: (((cb_table[me].state.status.changed ==
    cb_table[me].state.thisround ->
    cb_table[me].state.status.oldvalue :
    cb_table[me].state.status.value))
  != (1)) ||
    ((e.icmp_seq.value) !=
      ((cb_table[me].state.icmp_seq.changed ==
        cb_table[me].state.icmp_seq.value))
  ->
    cb_table[me].state.thisround ->
cb_table[me].state.icmp_seq.oldvalue =
    cb_table[me].state.icmp_seq.value))))) ||
((e.icmp_data.value) !=
    ((cb_table[me].state.icmp_data.changed ==
        cb_table[me].state.thisround ->
        cb_table[me].state.icmp_data.oldvalue :
        cb_table[me].state.icmp_data.value)))) ->
    b.value = 20;
    cb_table[me].outputs.BadEchoReply!b;
:: else -> skip;
fi
:: else -> skip;
fi
:: else -> skip;
fi

if :: (cb_table[me].outputs.IsAlive?[a] ||
    cb_table[me].outputs.BadEchoReply?[d] ||
    cb_table[me].outputs.BadEchoRequest?[d]) ->
    d.value = true;
    cb_table[me].outputs.Done!d;
:: else -> skip;
fi;

if :: cb_table[me].outputs.IsAlive?[a] ->
    cb_table[me].state.status.changed = cb_table[me].state.thisround;
    cb_table[me].state.status.oldvalue = cb_table[me].state.status.value;
    cb_table[me].state.status.value = 2;
:: else -> skip;
fi;

/* Removing Input Events From Channel */

if :: cb_table[me].inputs.Init?[i] -> cb_table[me].inputs.Init?i;
:: else -> skip;
fi;

if :: cb_table[me].inputs.EchoRequest?[e] -> cb_table[me].inputs.EchoRequest?e;
:: else -> skip;
fi;

if :: cb_table[me].inputs.EchoReply?[e] -> cb_table[me].inputs.EchoReply?e;
:: else -> skip;
fi;
od;
Appendix B

AODV Event Recognition

B.1 The AODV State Machine

The AODV Specification [Per97] is an evolving document published by the MANET working group at the IETF (http://www.ietf.org). The document describes the various packets and network events that an AODV process needs to respond to. Here we present the reactive state machine that an implementation of AODV version 0 is supposed to implement. There are two control states corresponding to the presence or absence of a route to the destination. In addition, for each destination AODV keeps track of the best known route: seq_no, hop_cnt, next_hop, and lifetime. An AODV node runs a state machine for each destination; the state machine for the destination dst is shown in Table B.1. We have left out some details of timeouts and link error events, which the protocol needs to handle as well. The state machine presented here captures the major packet events and their relation to the state at an AODV process.
### STATE: No Route

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeOut</td>
<td>seq_no ← 0</td>
<td>No Route</td>
</tr>
<tr>
<td>Recv from p: RREQ ((d, \text{hops_to_src, dest_seq_no}, s, \text{src_seq_no})) (\land s = \text{dst})</td>
<td>dest_seq_no ← max(seq_no, dest_seq_no); Broadcast RREQ((d, \text{hops_to_src} + 1, \text{dest_seq_no}, s, \text{src_seq_no}))</td>
<td>No Route</td>
</tr>
<tr>
<td>Recv from p: RREQ ((d, \text{hops_to_src, dest_seq_no}, s, \text{src_seq_no})) (\land s = \text{dst} \land \text{src_seq_no} \geq \text{seq_no})</td>
<td>next_hop ← p; hop_cnt ← hops_to_src + 1; seq_no ← src_seq_no; lifetime ← REV_ROUTE_LIFE</td>
<td>Has Route</td>
</tr>
<tr>
<td>Recv from p: RREP ((\text{hops_to_dest}, d, \text{dest_seq_no}, \text{route_lifetime})) (\land d = \text{dst} \land \text{dest_seq_no} \geq \text{seq_no})</td>
<td>next_hop ← p; hop_cnt ← hops_to_dest + 1; seq_no ← dest_seq_no; lifetime ← route_lifetime</td>
<td>Has Route</td>
</tr>
</tbody>
</table>

### STATE: Has Route

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeOut</td>
<td>seq_no ← seq_no + 1; next_hop ← 0; hop_cnt ← 255 Send to active neighbors: RREP((255, \text{dst, seq_no, BAD_LINK_LIFETIME}))</td>
<td>No Route</td>
</tr>
<tr>
<td>Recv from p: RREQ ((d, \text{hops_to_src, dest_seq_no}, s, \text{src_seq_no})) (\land s = \text{dst}) (\land [\text{src_seq_no, hops_to_src}]) is better than ([\text{seq_no, hop_cnt}])</td>
<td>next_hop ← p; hop_cnt ← hops_to_src + 1; seq_no ← src_seq_no; lifetime ← REV_ROUTE_LIFE</td>
<td>Has Route</td>
</tr>
<tr>
<td>Recv from p: RREP ((\text{hops_to_dest}, d, \text{dest_seq_no}, \text{route_lifetime})) (\land d = \text{dst}) (\land [\text{dest_seq_no, hops_to_dest}]) is better than ([\text{seq_no, hop_cnt}])</td>
<td>next_hop ← p; hop_cnt ← hops_to_dest + 1; seq_no ← dest_seq_no; lifetime ← route_lifetime</td>
<td>Has Route</td>
</tr>
<tr>
<td>Recv from p: RREP ((255, d, \text{dest_seq_no}, \text{route_lifetime})) (\land d = \text{dst}) (\land \text{dest_seq_no} &gt; \text{seq_no})</td>
<td>next_hop ← 0; hop_cnt ← 255; seq_no ← dest_seq_no; lifetime ← BAD_LINK_LIFETIME</td>
<td>Has Route</td>
</tr>
<tr>
<td>Recv from p: RREQ ((d, \text{hops_to_src, dest_seq_no}, s, \text{src_seq_no})) (\land d = \text{dst} \land \text{dest_seq_no} \leq \text{seq_no})</td>
<td>Unicast from me for s: RREP((\text{hop_cnt, d, seq_no, MY_ROUTE_TIMEOUT}))</td>
<td>Has Route</td>
</tr>
<tr>
<td>Recv unicast from p for dst: DATA</td>
<td>Send to next_hop : DATA</td>
<td>Has Route</td>
</tr>
<tr>
<td>Recv unicast from p for dst: RREP ((\text{hops_to_dest} + 1, d, \text{dest_seq_no}, \text{route_lifetime}))</td>
<td>Send to next_hop : RREP((\text{hops_to_dest}, d, \text{dest_seq_no}, \text{route_lifetime}))</td>
<td>Has Route</td>
</tr>
</tbody>
</table>
B.2 AODV Event Recognizer

This is the NERL recognizer, aodv.nerl, that was used to carry out the simulation analysis for AODV in Chapter 8.

```c
/* Type/Struct Definitions, typically for event attributes */

#define NODES 51
#define INFINITY 255

typedef {
    int atnode;
    int fordest;
    int src;
    int src_seq;
    int src_hc;
    int dest;
    int dest_seq;
    int bcastid;
    int prev;
    int next_hop;
    int eventty;
    int pktty;
    int ttl;
    int life
}
pkt;

typedef {
}
basic;

typedef {
    int at;
    int dst;
    int best_seq;
    int best_hc;
    int best_next
}
cell;

typedef {
    int node;
    int next;
    int dest;
    int seq_at_node;
    int hc_at_node;
    int seq_at_next;
    int hc_at_next
} loopinfo;

/* Input Events */

input event pkt Pkt ;
input event basic Init ;
input event basic Finish ;

/* Output Alarms */
```

155
out event cell NotSeqMono;
out event cell DestForwards;
out event cell BadFwd;
out event cell BadDestRep;
out event cell BadNodeRep;
out event cell BadRReq;
out event cell BadRerr;
out event cell BadRerrFwd;
out event loopinfo LoopInvFails;

/* Private Local Events */

event cell Routeinfo;
event cell Recvroute;
event cell Recvbetter;
event cell Sendroute;
event cell Send;
event cell Sendnext;
event cell Sendreq;
event cell Incseq;
event cell Senderr;

/* State Variables */

int best_seq[NODES][NODES];
int best_HC[NODES][NODES];
int best_next[NODES][NODES];
int at;
int dst;

/* Protocol (local) Event Definitions */

event Routeinfo = Pkt OccurredWhen
    (Pkt.pktty > 0)
    WithAttributes {
        Routeinfo.at := Pkt.atnode;
        Routeinfo.dst := Pkt.src;
        Routeinfo.best_seq := best_seq[Pkt.atnode][Pkt.src];
        Routeinfo.best_HC := best_HC[Pkt.atnode][Pkt.src];
        Routeinfo.best_next := best_next[Pkt.atnode][Pkt.src];
    }

event Recvroute = (Pkt & Routeinfo) OccurredWhen
    (Pkt.eventty == 0)
    WithAttributes {Recvroute := Routeinfo};

event Recvbetter = (Pkt & Recvroute) OccurredWhen
    (Pkt.src_seq > best_seq[Recvroute.at][Recvroute.dst]) ||
    (((Pkt.src_seq == best_seq[Recvroute.at][Recvroute.dst]) &&
      (Pkt.src_HC < best_HC[Recvroute.at][Recvroute.dst])))
    WithAttributes {Recvbetter := Recvroute};

event Sendroute = (Pkt & Routeinfo) OccurredWhen
    (Pkt.eventty == 1) &&
    (Pkt.src_HC < INFINITY)
WithAttributes {Sendroute := Routeinfo};

event Senderr = (Pkt & Routeinfo) OccurredWhen
  (Pkt.eventty == 1) &&
  (Pkt.pktty == 2) &&
  (Pkt.src_HC >= INFINITY)
WithAttributes {Senderr := Routeinfo};

event Incseq = (Pkt & Sendroute) OccurredWhen
  (Sendroute.dst == Sendroute.at) &&
  (Pkt.src_seq > best_seq[Sendroute.at][Sendroute.at])
WithAttributes {Incseq := Sendroute};

event Send = Pkt OccurredWhen
  (Pkt.eventty == 1)
WithAttributes {
  Send.at := Pkt.atnode;
  Send.dst := Pkt.dest;
  Send.best_seq := best_seq[Pkt.atnode][Pkt.dest];
  Send.best_HC := best_HC[Pkt.atnode][Pkt.dest];
  Send.best_next := best_next[Pkt.atnode][Pkt.dest];
}

event Sendnext = (Pkt & Send) OccurredWhen
  ((Pkt.pktty == 0) ||
  (Pkt.pktty == 2))
WithAttributes {Sendnext := Send};

event Sendreq = (Pkt & Send)
  OccurredWhen (Pkt.pktty == 1)
WithAttributes {Sendreq := Send};

/* Output (error) Event Definitions: Conformance Check */

event NotSeqMono = (Pkt & Sendroute) OccurredWhen
  (Sendroute.dst == Sendroute.at) &&
  (Pkt.src_seq < best_seq[Sendroute.at][Sendroute.at])
WithAttributes {NotSeqMono := Sendroute};

event DestForwards = Sendnext OccurredWhen
  (Sendnext.at == Sendnext.dst)
WithAttributes {DestForwards := Sendnext};

event BadFwd = (Pkt & Sendnext) OccurredWhen
  (Pkt.next_hops != best_next[Sendnext.at][Sendnext.dst])
WithAttributes {BadFwd := Sendnext};

event BadDestRep = (Pkt & Sendroute) OccurredWhen
  ((Sendroute.dst == Sendroute.at) &&
  (Pkt.src_HC != 0))
WithAttributes {BadDestRep := Sendroute};

event BadNodeRep = (Pkt & Sendroute) OccurredWhen
  ((Sendroute.dst != Sendroute.at) &&
  (Pkt.src_seq != best_seq[Sendroute.at][Sendroute.dst]) ||
  (Pkt.src_HC != best_HC[Sendroute.at][Sendroute.dst])))
WithAttributes {BadNodeRep := Sendroute};

event BadRReq = (Pkt & Sendreq) OccurredWhen
((Sendreq.dst == Sendreq.at) ||
 (Pkt.dest_seq != 0) &&
 (Pkt.dest_seq < best_seq[Sendreq.at][Sendreq.dst])))
WithAttributes {BadRReq := Sendreq};

event BadRerr = (Pkt & Senderr) OccurredWhen
(best_hc[Senderr.at][Senderr.dst] < INFINITY) &&
(Pkt.src_seq < best_seq[Senderr.at][Senderr.dst] + 1)
WithAttributes {BadRerr := Senderr};

event BadRerrFwd = (Pkt & Senderr) OccurredWhen
(best_hc[Senderr.at][Senderr.dst] == INFINITY) &&
(Pkt.src_seq != best_seq[Senderr.at][Senderr.dst])
WithAttributes {BadRerrFwd := Senderr};

/* State Transitions (AODV State Machine) */

transition i:
Init -> {
  int i;
  i := 0;
  while (i < 50) do
  {
    int j;
    j := 0;
    while (j < 50) do
    {
      best_seq[i][j] := 0;
      best_hc[i][j] := 0;
      best_next[i][j] := 0;
      j := j + 1
    }
    i := i + 1
  }
};

transition d:
Finish -> { };

transition better:
Pkt & Recvbetter -> {
  best_seq[Recvbetter.at][Recvbetter.dst] := Pkt.src_seq;
  if (Pkt.src_hc == INFINITY) then {
    best_hc[Recvbetter.at][Recvbetter.dst] := INFINITY
  } else {
    best_hc[Recvbetter.at][Recvbetter.dst] := Pkt.src_hc + 1
  }
  best_next[Recvbetter.at][Recvbetter.dst] := Pkt.prev
};

transition seq:
Pkt & Incseq -> {

158
best_seq[Incseq.at][Incseq.at] := Pkt.src_seq

transition err:
Pkt & Senderr -> {
    best_seq[Senderr.at][Senderr.dst] := Pkt.src_seq;
    best_hc[Senderr.at][Senderr.dst] := INFINITY;
    best_next[Senderr.at][Senderr.dst] := 0
};

/* Requirements Checking: Loop Invariant */

event LoopInvFails = (Pkt & Recvbetter)
    OccurredWhen
        ((best_seq[Recvbetter.at][Recvbetter.dst] >
            best_seq[best_next[Recvbetter.at][Recvbetter.dst]][Recvbetter.dst]))
        || ((best_seq[Recvbetter.at][Recvbetter.dst] ==
            best_seq[best_next[Recvbetter.at][Recvbetter.dst]][Recvbetter.dst])
            && (best_hc[Recvbetter.at][Recvbetter.dst] <=
            best_hc[best_next[Recvbetter.at][Recvbetter.dst]][Recvbetter.dst]))
    WithAttributes {
        LoopInvFails.node := Recvbetter.at;
        LoopInvFails.next := best_next[Recvbetter.at][Recvbetter.dst];
        LoopInvFails.dest := Recvbetter.dst;
        LoopInvFails.seq_at_node := best_seq[Recvbetter.at][Recvbetter.dst];
        LoopInvFails.hc_at_node := best_hc[Recvbetter.at][Recvbetter.dst];
        LoopInvFails.seq_at_next :=
            best_seq[best_next[Recvbetter.at][Recvbetter.dst]][Recvbetter.dst];
        LoopInvFails.hc_at_next :=
            best_hc[best_next[Recvbetter.at][Recvbetter.dst]][Recvbetter.dst]
    };

159
Appendix C

SMTP Event Recognition

C.1 IP Fragment Reassembly

The following is the key portion of the IP_Reassemb recognizer. The Frag input event is triggered whenever a new fragment \( p \) is seen. The LastFrag event indicates that this fragment occurs at the end of the packet and so can be used to compute the IP packet length. Finally, the PktDone output event is generated when packet reassembly is complete.

```plaintext
transition Frag(p) -> {
    push(ipdbuf,1);
    ipdbuf[ipdbuf#length - 1].first = p.ip_off;
    ipdbuf[ipdbuf#length - 1].len = p.ip_plen;
    ipdbuf[ipdbuf#length - 1].data = p.payload;

    int i = p.ip_off in {
        while (i*8 < p.ip_plen) {
            ipdatarecv[i] = true;
            ipdataptr[i] = ipdbuf#length - 1;
        }
    }
}

transition LastFrag(p) -> {
    ip_plen = p.ip_off*8 + p.ip_plen;
}

event Frag(p) OccurredWhen (ip_plen != -1) ->
    PktDone(d) WithAttributes {
        int pdone = true in {
            int i = 0 in {
                while ((i*8 < ip_plen) && (pdone == true)) {
                    if (ipdatarecv[i] == false) then pdone = false;
                    i = i + 1;
                }
            }
            d.isdone = pdone;
            if (pdone == true) then {
```
C.2 TCP Stream Reassembly

The following is the key portion of the stream reassembly code for a single TCP session. The input event TCP is triggered when a TCP packet has been seen, Data indicates that the packet contains data and To indicates the direction of transfer (the other direction is From). The out-of-order data in the To direction is stored in the variable length array to.segments. A data segment is considered delivered when an Ack is seen in the reverse direction acknowledging the sequence number.

```c
string s = "" in {
    int i = 0 in {
        while (i*8 < ip.plen) {
            concat(s, ipdatabuf[ipdataptr[i]].data);
            i = i + ipdatabuf[ipdataptr[i]].len/8;
        }
        d.data = s;
    }
    else d.data = "";
}
```

transition TCP(p) && Data(d) && To(t) -> {
    int copy = false in {
        int i = 0 in {
            while ((i < to.segments#length) &&
                    (to.segments[i].beg <= p.tcp_seqno)) {
                if (to.segments[i].beg == p.tcp_seqno)
                    then copy = true
                else copy = false
            }
        }
        if (copy == false) {
            push(to.segments);
            int i = to.segments#length - 2 in {
                while ((i >= 0) &&
                        (to.segments[i].beg > p.tcp_seqno)) {
                    to.segments[i+1].beg = to.segments[i].beg;
                    to.segments[i+1].dlen = to.segments[i].dlen;
                    to.segments[i+1].data = to.segments[i].data;
                }
                to.segments[i+1].beg = p.tcp_seqno;
                to.segments[i+1].dlen = p.tcp_dlen;
                to.segments[i+1].data = p.tcp_data;
            }
        }
    }
}
event TCP(p) & & Ack(a) & & From(t) ->
  ToData(d) WithAttributes {
    d.wbase = p.tcp_ackno;
    int i = 0 in {
      while ((i < to.segments#length) & &
             (to.segments[i].beg < p.tcp_ackno)) {
        push(d.segments);
        d.segments[d.segments#length-1] = to.segments[i].data;
      }
    }
  }

C.3 SMTP Event Recognizer

Recognizer SMTP =

# define RESPNONE 0
# define RESPOK 1
# define RESPDONE 2

# define CLEAR 0
# define HELLO 1
# define MAIL 2
# define RCPT 3
# define DATA 4
# define DATAREAD 5
# define DATASEND 6
# define RESET 7
# define QUIT 8
# define NOOP 9
# define HELP 10
# define VERIFY 11
# define EXPAND 12
# define JUNK 13

typedef {
    int client.ip;
    int server.ip;
    int client.port;
} ty.smtpsession;

typedef {
    string user;
    string domain;
} ty.mbox;

typedef {
    ty.mbox sender;
    ty.mbox receiver;
} ty.env;

typedef {

typedef {  
  int client_ip;  
  int server_ip;  
  int client_port;  
  
  bool dataline;  
  bool dataend;  
  bool hello;  
  bool mailfrom;  
  bool rcptto;  
  bool data;  
  bool quit;  
  bool reset;  
  bool noop;  
  bool help;  
  bool verify;  
  bool expand;  
  bool junk;  
  
  string text;  
  ty_mbox mbox;  
} ty_cmd;  

typedef {  
  int client_ip;  
  int server_ip;  
  int client_port;  
  
  bool junk;  
  
  int code;  
  bool ok;  
  string data;  
  bool contd;  
} ty_resp;  

input event ty_smtpsession Init;  
input event ty_cmd Command;  
input event ty_resp Response;  

output event ty_env Envelope_Sent;  
output event ty_env Envelope_Accepted;  
output event ty_msg Mail_Sent;  
output event ty_msg Mail_Accepted;  

output event string Command_Error;  
output event string Response_Error;  
output event bool Done;  

state int client_ip;
state int server_ip;
state int client_port;

state int status;
state int respstatus;
state int laststatus;

state int respcode;
state ty_mbox sender;
state ty_mbox senderseen;
state ty_mbox receiverseen;
state ty_mbox receivers[];
state ty_mbox receiversseen[];

state string databuf;

event string Hello;
event ty_mbox MailFrom;
event ty_mbox RcptTo;
event bool Data;
event string Dataline;
event string Dataend;
event bool Reset;
event bool Quit;
event bool Help;
event bool Noop;
event bool Verify;
event bool Expand;
event bool Junk;

event ty_resp MetaResponse;

event Command(c)
  OccurredWhen (c.hello == true) -> Hello(h)
  WithAttributes {h = c.text;};

event Command(c)
  OccurredWhen (c.mailfrom == true) -> MailFrom(m)
  WithAttributes {m = c.mbox;};

event Command(c)
  OccurredWhen (c.rcptto == true) -> RcptTo(r)
  WithAttributes {r = c.mbox;};

event Command(c)
  OccurredWhen (c.data == true) -> Data(d)
  WithAttributes {d = true;};

event Command(c)
  OccurredWhen (c.dataline == true) -> Dataline(d)
  WithAttributes {d = c.text;};

event Command(c)
OccurredWhen (c.dataend == true) -> Dataend(d)  
   WithAttributes {d = c.text;};

event Command(c)  
   OccurredWhen (c.quit == true) -> Quit(b)  
      WithAttributes {b = true;};

event Command(c)  
   OccurredWhen (c.reset == true) -> Reset(b)  
      WithAttributes {b = true;};

event Command(c)  
   OccurredWhen (c.noop == true) -> Noop(b)  
      WithAttributes {b = true;};

event Command(c)  
   OccurredWhen (c.verify == true) -> Verify(b)  
      WithAttributes {b = true;};

event Command(c)  
   OccurredWhen (c.help == true) -> Help(b)  
      WithAttributes {b = true;};

event Command(c)  
   OccurredWhen (c.expand == true) -> Expand(b)  
      WithAttributes {b = true;};

event Command(c)  
   OccurredWhen (c.junk == true) -> Junk(b)  
      WithAttributes {b = true;};

event Junk(b)  
   OccurredWhen (status != DATA) -> Command_Error(b)  
      WithAttributes {b = "Junk_Command";};

transition i: Init(i) -> {client_ip = i.client_ip;  
   server_ip = i.server_ip;  
   client_port = i.client_port;  
   status = CLEAR;  
   respstatus = RESPNONE;};

transition h: Hello(h)  
   OccurredWhen  
      (respstatus == RESPDONE) -> {laststatus = status;  
         status = HELLO;  
         respstatus = RESPNONE;};

event Hello(h)  
   OccurredWhen (respstatus != RESPDONE) -> Command_Error(b)  
      WithAttributes {b = "Unexpected_Hello";};
transition m: MailFrom(m)
  OccurredWhen
  ((status == HELLO) &&
   (respstatus == RESPDONE)) \rightarrow \{laststatus = status;
          status = MAIL;
          respstatus = RESPNONE;
          senderseen.user = m.user;
          senderseen.domain = m.domain;
  \};

event MailFrom(h)
  OccurredWhen !((status == HELLO) &&
   (respstatus == RESPDONE)) \rightarrow Command_Error(b)
    WithAttributes
    \{b = "Unexpected„MailFrom";\};

transition r: RcptTo(r)
  OccurredWhen
  (((status == MAIL) ||
     (status == RCPT)) &&
   (respstatus == RESPDONE)) \rightarrow \{
    laststatus = status;
    status = RCPT;
    respstatus = RESPNONE;
    receiverseen.user = r.user;
    receiverseen.domain = r.domain;
    push(receiverseen, 1);
    receiverseen[receiverseen#length−1].user =
                      r.user;
    receiverseen[receiverseen#length−1].domain =
                      r.domain;
  }

event RcptTo(h)
  OccurredWhen !(((status == MAIL) ||
     (status == RCPT)) &&
   (respstatus == RESPDONE)) \rightarrow Command_Error(b)
    WithAttributes
    \{b = "Unexpected„Rcpt";\};

event RcptTo(r)
  OccurredWhen (((status == MAIL) ||
     (status == RCPT)) &&
   (respstatus == RESPDONE)) \rightarrow Envelope_Sent(e)
    WithAttributes \{
    e.sender.user = senderseen.user;
    e.sender.domain = senderseen.domain;
    e.receiver.user = receiverseen.user';
    e.receiver.domain = receiverseen.domain';\};

transition d: Data(d)
  OccurredWhen (((status == RCPT) &&
     (respstatus == RESPDONE)) \rightarrow \{
    laststatus = status;
status = DATA;
respstatus = RESPNONE;

\textbf{event} Data(h)
\hspace{1em}OccurredWhen \((\text{status} == \text{DATA}) \land \text{respstatus} == \text{RESPDONE})\)
\hspace{1em}\rightarrow Command_{\text{Error}}(b)
\hspace{1em}\text{WithAttributes}
\hspace{1em}\{b = \text{"Unexpected \text{\textquotedbl}Data\text{\textquotedbl}"};\}

\textbf{transition} dl: Dataline(d)
\hspace{1em}OccurredWhen \((\text{status} == \text{DATA}) \land \text{respstatus} == \text{RESPDONE})\)
\hspace{1em}\rightarrow \{\text{concat(databuf, d)};\}

\textbf{transition} de: Dataend(d)
\hspace{1em}OccurredWhen \((\text{status} == \text{DATA}) \land \text{respstatus} == \text{RESPDONE})\)
\hspace{1em}\rightarrow \{\text{\text{laststatus} = status; status = DATAEND; respstatus = RESPNONE; concat(databuf, d)};\}

\textbf{event} Dataend(d)
\hspace{1em}OccurredWhen \((\text{status} == \text{DATAREAD}) \land \text{respstatus} == \text{RESPDONE})\)
\hspace{1em}\rightarrow \text{Mail\_Sent(m)}
\hspace{1em}\text{WithAttributes}\{\text{int i; m.sender.user} = \text{senderseen.user}; m.sender.domain} = \text{senderseen.domain}; \text{push(m.receivers, receiversseen\#length)}; i = 0; \text{while (i < receiversseen\#length) do }\{m.receivers[i].user} = receiversseen[i].user; m.receivers[i].domain} = receiversseen[i].domain; \}\text{; m.data} = \text{databuf '};\}

\textbf{transition} rs: Reset(r)
\hspace{1em}OccurredWhen \text{respstatus} == \text{RESPDONE}\)
\hspace{1em}\rightarrow \{\text{laststatus} = status; status = RESET; respstatus = RESPNONE; pop(receivers, receivers\#length); pop(receiversseen, receiversseen\#length); databuf} = \text{""};\}

\textbf{event} Reset(h)
\hspace{1em}OccurredWhen \text{respstatus} != \text{RESPDONE}\)
\hspace{1em}\rightarrow \text{Command\_Error}(b)
\hspace{1em}\text{WithAttributes}

167
transition q: Quit(q)
   OccurredWhen (respstatus == RESPDONE) -> {
      laststatus = status;
      status = QUIT;
      respstatus = RESPNONE;
      pop(receivers, receivers#length);
      pop(receiversseen, receiversseen#length);
      databuf = "";
   };

event Quit(h)
   OccurredWhen (respstatus != RESPDONE) -> Command_Error(b)
      WithAttributes
      {b = "Unexpected_Quiet";};

transition n: Noop(n)
   OccurredWhen (respstatus == RESPDONE) -> {
      laststatus = status;
      status = NOOP;
      respstatus = RESPNONE;
   };

event Noop(h)
   OccurredWhen (respstatus != RESPDONE) -> Command_Error(b)
      WithAttributes
      {b = "Unexpected_Noop";};

transition hl: Help(n)
   OccurredWhen (respstatus == RESPDONE) -> {
      laststatus = status;
      status = HELP;
      respstatus = RESPNONE;
   };

event Help(h)
   OccurredWhen (respstatus != RESPDONE) -> Command_Error(b)
      WithAttributes
      {b = "Unexpected_Help";};

transition v: Verify(n)
   OccurredWhen (respstatus == RESPDONE) -> {
      laststatus = status;
      status = VERIFY;
      respstatus = RESPNONE;
   };

event Verify(h)
   OccurredWhen (respstatus != RESPDONE) -> Command_Error(b)
      WithAttributes
      {b = "Unexpected_Verify";};

transition e: Expand(n)
OccurredWhen (respstatus == RESPDONE) -> {laststatus = status; status = EXPAND; respstatus = RESPNONE; }

\textbf{event} Expand(h)
OccurredWhen (respstatus != RESPDONE) -> Command_Error(b)  
\hspace{1em} \textbf{WithAttributes}
\hspace{2em} \{b = "Unexpected Expand";\}

\textbf{transition} junk: Command_Error(b) -> {laststatus = status; status = JUNK; respstatus = RESPNONE; }

\textbf{event} Response(r)
OccurredWhen (respstatus == RESPDONE) || (respstatus == RESPOK) && (r.code != respcode)) -> Response_Error(b)  
\hspace{1em} \textbf{WithAttributes}
\hspace{2em} \{b = "Unexpected Response";\}

\textbf{transition} rsp: Response(r)
OccurredWhen ((respstatus == RESPNONE) || ((respstatus == RESPOK) && (r.code == respcode))) -> {
\hspace{1em} if (r.contd == true) then {
\hspace{2em} respstatus = RESPOK;
\hspace{2em} respcode = r.code;
\hspace{1em} } else {respstatus = RESPDONE;}
\hspace{1em} }

\textbf{event} Response(r)
OccurredWhen (r.contd == false) -> MetaResponse(m)  
\hspace{1em} \textbf{WithAttributes}
\hspace{2em} \{m.client_ip = r.client_ip;
\hspace{2em} m.server_ip = r.server_ip;
\hspace{2em} m.client_port = r.client_port;
\hspace{2em} m.code = r.code;
\hspace{2em} m.ok = r.ok;
\hspace{2em} m.data = r.data;
\hspace{2em} m.contd = r.contd;
\hspace{2em} \}

\textbf{event} MetaResponse(m)
OccurredWhen (|(status == CLEAR) && ((m.ok == true) => (m.code != 220)) && ((m.ok == false) => (m.code != 554))) || (status == HELLO) && ((m.ok == true) => (m.code != 250)) && ((m.ok == false) => ((m.code != 504) && (m.code != 550)))) || (status == MAIL) &&
((m.ok == true) => (m.code != 250) &&
(m.ok == false) => (m.code != 552) &&
(m.code != 451) &&
(m.code != 452) &&
(m.code != 550) &&
(m.code != 553) &&
(m.code != 503))) ||

((status == RCPT) &&
((m.ok == true) => ((m.code != 250) &&
(m.code != 251))) &&
((m.ok == false) => ((m.code != 550) &&
(m.code != 551) &&
(m.code != 552) &&
(m.code != 553) &&
(m.code != 450) &&
(m.code != 451) &&
(m.code != 452) &&
(m.code != 503))) ||

((status == DATA) &&
((m.ok == true) => (m.code != 354)) &&
((m.ok == false) => ((m.code != 451) &&
(m.code != 554) &&
(m.code != 503))) ||

((status == DATAEND) &&
((m.ok == true) => (m.code != 250)) &&
((m.ok == false) => ((m.code != 551) &&
(m.code != 554) &&
(m.code != 552) &&
(m.code != 553) &&
(m.code != 550) &&
(m.code != 502) &&
(m.code != 504))) ||

((status == RESET) && (m.ok => m.code != 250)) ||

((status == NOOP) && (m.ok => m.code != 250)) ||

((status == QUIT) && (m.ok => m.code != 221)) ||

((status == VERIFY) &&
((m.ok == true) => ((m.code != 250) &&
(m.code != 251) &&
(m.code != 252))) &&
((m.ok == false) => (m.code != 550) &&
(m.code != 551) &&
(m.code != 553) &&
(m.code != 502) &&
(m.code != 504))) ||

((status == EXPAND) &&
((m.ok == true) => ((m.code != 250) &&
(m.code != 252))) &&
((m.ok == false) => ((m.code != 550) &&
(m.code != 500) &&
(m.code != 502) &&
(m.code != 504))) ||

((status == HELP) &&
((m.ok == true) => ((m.code != 211) &&
(m.code != 214))) &&
((m.ok == false) => ((m.code != 502) &&
(m.code != 504))) ||

((m.ok == false) => ((m.code != 500) &&
(m.code != 501) &&
(m.code != 504)))

170
(m.code != 421)) \rightarrow 
Response_Error(b) \ WithAttributes \{b = "Incorrect\_Response\_Code"\};

**transition** mr: MetaResponse(m)
OccurredWhen ((status == HELLO) \\
(\ m.ok == false)) \rightarrow \{status = laststatus;\};

**transition** mr: MetaResponse(m)
OccurredWhen ((status == HELLO) \\
(\ m.ok == true)) \rightarrow { \\
    pop(receivers, receivers#length); \\
    pop(receiversseen, receiversseen#length); \\
    databuf = ""; \\
};

**transition** mr1: MetaResponse(m)
OccurredWhen ((status == MAIL) \\
(\ m.ok == false)) \rightarrow \{status = laststatus;\};

**transition** mr2: MetaResponse(m)
OccurredWhen ((status == MAIL) \\
(\ m.ok == true)) \rightarrow { \\
    sender.user = senderseen.user; \\
    sender.domain = senderseen.domain; \\
};

**transition** mr3: MetaResponse(m)
OccurredWhen ((status == RCPT) \\
(\ m.ok == false)) \rightarrow \{status = laststatus;\};

**transition** mr4: MetaResponse(m)
OccurredWhen ((status == RCPT) \\
(\ m.ok == true)) \rightarrow { \\
    push(receivers, 1); \\
    receivers[receivers#length -1].user = \\
    receiversseen.user; \\
    receivers[receivers#length -1].domain = \\
    receiversseen.domain; \\
};

**event** MetaResponse(m)
OccurredWhen ((status == RCPT) \\
(\ m.ok == true)) \rightarrow \ Envelope_Accepted(e) \\
    WithAttributes \{ \\
        e.sender.user = sender.user; \\
        e.sender.domain = sender.domain; \\
        e.receiver.user = receiverseen.user'; \\
        e.receiver.domain = receiverseen.domain'; \\
    \};

**transition** mr5: MetaResponse(m)
OccurredWhen ((status == DATA) \\
(\ m.ok == false)) \rightarrow { \\
    status = HELLO; \\
    pop(receivers, receivers#length);
event MetaResponse (m)
  OccurredWhen ((status == DATAEND) &&
          (m.ok == true)) -> Mail_Accepted (m)
    WithAttributes {
      int i;
      m.sender.user = sender.user;
      m.sender.domain = sender.domain;
      push(m.receivers, receivers#length);
      i = 0;
      while (i < receivers#length) do {
        m.receivers[i].user =
        receivers[i].user;
        m.receivers[i].domain =
        receivers[i].domain;
      }
      m.data = databuf;
    }
}

transition mr6: MetaResponse (m)
  OccurredWhen (status == DATAEND) -> {
    status = HELLO;
    pop(receivers, receivers#length);
    databuf = "";
  }

transition mr7: MetaResponse (m)
  OccurredWhen ((status == RESET) &&
              (laststatus != CLEAR)) -> {status = HELLO;};

transition mr7a: MetaResponse (m)
  OccurredWhen ((status == RESET) &&
               (laststatus == CLEAR)) -> {status = CLEAR;};

transition mr8: MetaResponse (m)
  OccurredWhen (status == QUIT) -> {status = CLEAR;};

event MetaResponse (m)
  OccurredWhen (status == QUIT) -> Done(d) WithAttributes {d = true};

transition mr9: MetaResponse (m)
  OccurredWhen (status == NOOP) -> {status = laststatus;};

transition mr10: MetaResponse (m)
  OccurredWhen (status == VERIFY) -> {status = laststatus;};

transition mr11: MetaResponse (m)
  OccurredWhen (status == HELP) -> {status = laststatus;};

transition mr12: MetaResponse (m)
  OccurredWhen (status == EXPAND) -> {status = laststatus;};
transition mr13: MetaResponse(m)
    OccurredWhen ((m.ok == false) &&
        (status == JUNK)) -> {status = laststatus;};

event MetaResponse(m)
    OccurredWhen ((status == JUNK) &&
        (m.ok == true)) -> Response_Error(b)
        WithAttributes
        {b = "Junk Resp is OK."};

event MetaResponse(m)
    OccurredWhen (m.ok == false) -> Response_Error(b)
        WithAttributes
        {b = "Neg Resp."};

event MetaResponse(m)
    OccurredWhen (m.code == 421) -> Done(d) WithAttributes {d = true};

EndRecognizer;
Appendix D

TCP Event Recognition

D.1 TCP Recognizer in NERL

```c
typedef {
    int ip_src;
    int ip_dst;
    int tcp_sport;
    int tcp_dport;
} ty/tcpflow;

typedef {
    int ip_src;
    int ip_dst;
    int tcp_sport;
    int tcp_dport;
    int tcp_seqno;
    int tcp_ackno;
    bool tcp_fin;
    bool tcp_syn;
    bool tcp_rst;
    bool tcp_push;
    bool tcp_ack;
    bool tcp_urg;
    int tcp_win;
    int tcp_dlen;
    string tcp_data;
} ty/tcppkt;

typedef {
    int beg;
    int dlen;
    string data;
} ty/seg;

typedef {
    int ip_src;
    int ip_dst;
```

174
typedef {
  int status;
  int wbase;
  int wsize;
  int syn;
  int fin;
  int unacked;
  int lastack;
  ty_seg segments[];
} ty_win;

input event ty_tcpflow Init;
input event ty_tcppkt TCP;

int ip_src;
int ip_dst;
int tcp_srcport;
int tcp_dport;

ty_win sender;
ty_win receiver;

output event bool Done;
output event ty_tcpdata TCP_Data;
output event string TCP_Error;

event bool Syn;
event bool Synack;
event bool Reset;
event bool Fin;
event bool Data;
event bool Ack;
event bool To;
event bool From;

event ty_win sendData;
event ty_win FromData;

transition Init(i) -> {
ip_src = i.ip_src;
ip_dst = i.ip_dst;
tcp_sport = i.tcp_sport;
tcp_dport = i.tcp_dport;
sender.status = CLOSED;
receiver.status = CLOSED;
};

event TCP(t) OccurredWhen ((t.ip_src == ip_src) &&
(t.ip_dst == ip_dst) &&
(t.tcp_sport == tcp_sport) &&
(t.tcp_dport == tcp_dport)) ->
To(t) WithAttributes {t = true};

event TCP(t) OccurredWhen ((t.ip_src == ip_dst) &&
(t.ip_dst == ip_src) &&
(t.tcp_sport == tcp_dport) &&
(t.tcp_dport == tcp_sport)) ->
From(t) WithAttributes {t = true};

event TCP(t) OccurredWhen ((t.tcp_syn == true) &&
(t.tcp_ack == false)) ->
Syn(t) WithAttributes {t = true};

event TCP(t) OccurredWhen ((t.tcp_syn == true) &&
(t.tcp_ack == true)) ->
Synack(s) WithAttributes {s = true};

event TCP(t) OccurredWhen ((t.tcp_syn == false) &&
(t.tcp_ack == true)) ->
Ack(s) WithAttributes {s = true};

event TCP(t) OccurredWhen ((t.tcp_rst == true)) ->
Reset(s) WithAttributes {s = true};

event TCP(t) OccurredWhen ((t.tcp_fin == true)) ->
Fin(s) WithAttributes {s = true};

event TCP(t) OccurredWhen ((t.tcp_syn == false) &&
(t.tcp_ack == false) &&
(t.tcp_rst == false)) ->
Data(s) WithAttributes {s = true};
event (TCP(p) & Syn(s) & To(t))
  OccurredWhen (sender.status != CLOSED) ->
  TCP_Error(hs) WithAttributes
    {hs = "1:Syn_sent_in_non-closed_statex"};

transition (TCP(p) & Syn(s) & To(t))
  OccurredWhen (sender.status == CLOSED) -> {
    sender.status = SYNSENT;
    sender.wsize = p.tcp_win;
  };

transition (TCP(p) & Syn(s) & To(t))
  OccurredWhen (receiver.status == CLOSED) -> {
    receiver.status = C_SYNRECD;
    receiver.syn = p.tcp_seqno;
    receiver.wbase = p.tcp_seqno;
  };

transition (TCP(p) & Syn(s) & To(t))
  OccurredWhen (receiver.status == SYNSENT) -> {
    receiver.status = S_SYNRECD;
    receiver.syn = p.tcp_seqno;
    receiver.wbase = p.tcp_seqno;
  };

event (TCP(p) & Syn(s) & From(t))
  OccurredWhen (receiver.status != CLOSED) ->
  TCP_Error(hs) WithAttributes
    {hs = "2:Syn_sent_in_non-closed_statex"};

transition (TCP(p) & Syn(s) & From(t))
  OccurredWhen (receiver.status == CLOSED) -> {
    receiver.status = SYNSENT;
    receiver.wsize = p.tcp_win;
  };

transition (TCP(p) & Syn(s) & From(t))
  OccurredWhen (sender.status == CLOSED) -> {
    sender.status = C_SYNRECD;
    sender.syn = p.tcp_seqno;
    sender.wbase = p.tcp_seqno;
  };

transition (TCP(p) & Syn(s) & From(t))
  OccurredWhen (sender.status == SYNSENT) -> {
    sender.status = S_SYNRECD;
  };

177
sender.syn = p.tcp_seqno;
sender.wbase = p.tcp_seqno;
}

/** ****************************************** */
/** SYNACK To */
/** ****************************************** */

event (TCP(p) & Synack(s) & To(t))
  OccurredWhen (sender.status != C_SYNRECD) ->
  TCP_Error(hs) WithAttributes
  {hs = "3: Synack sent in non-c_synrecd state"};

event (TCP(p) & Synack(s) & To(t))
  OccurredWhen (((sender.status == C_SYNRECD) &&
                  (p.tcp_ackno != sender.syn+1)) ->
  TCP_Error(hs) WithAttributes
  {hs = "4: Synack sent with bad ack"};

transition (TCP(p) & Synack(s) & To(t))
  OccurredWhen (sender.status == C_SYNRECD) ->
  {sender.status = SYNRECD;
   sender.wsize = p.tcp_win;
  };

transition (TCP(p) & Synack(s) & To(t))
  OccurredWhen (receiver.status == SYNSENT) ->
  {receiver.status = S_ESTAB;
   receiver.syn = p.tcp_seqno;
   receiver.wbase = p.tcp_seqno;
  };

/** ****************************************** */
/** SYNACK From */
/** ****************************************** */

event (TCP(p) & Synack(s) & From(t))
  OccurredWhen (receiver.status != C_SYNRECD) ->
  TCP_Error(hs) WithAttributes
  {hs = "5: Synack sent in non-c_synrecd state"};

event (TCP(p) & Synack(s) & From(t))
  OccurredWhen ((receiver.status == C_SYNRECD) &&
                  (p.tcp_ackno != receiver.syn+1)) ->
  TCP_Error(hs) WithAttributes
  {hs = "6: Synack sent with bad ack"};

transition (TCP(p) & Synack(s) & From(t))
  OccurredWhen (receiver.status == C_SYNRECD) ->
  {receiver.status = SYNRECD;
receiver.wsize = p.tcp_win;
}

transition (TCP(p) & Synack(s) & From(t))
OccurredWhen (sender.status == SYNSENT) -> {
    sender.status = S_ESTAB;
    sender.syn = p.tcp_seqno;
    sender.wbase = p.tcp_seqno;
};

/∗ --------------------------------------------------------------- ∗/
/∗ ACK of SYN To ∗/
/∗ --------------------------------------------------------------- ∗/

event (TCP(p) & Ack(s) & To(t))
OccurredWhen ((sender.status == CLOSED) ||
    (sender.status == C_SYNRECD) ||
    (sender.status == SYNSENT) ||
    (sender.status == SYNRECD)) ->
    TCP_Error(hs) WithAttributes
    {hs = “7:Ack_sent_in_bad_state”};

event (TCP(p) & Ack(s) & To(t))
OccurredWhen (p.tcp_ackno != sender.wbase+1) & &
    (p.tcp_ackno != sender.fin + 1) ||
    (p.tcp_ackno < sender.lastack)) ->
    TCP_Error(hs) WithAttributes
    {hs = “8:Ack_sent_with_bad_ackno”};

transition (TCP(p) & Ack(s) & To(t))
OccurredWhen (sender.status == S_SYNRECD) -> {
    sender.status = SYNRECD;
};

transition (TCP(p) & Ack(s) & To(t))
OccurredWhen (sender.status == S_ESTAB) -> {
    sender.status = ESTABLISHED;
};

transition (TCP(p) & Ack(s) & To(t))
OccurredWhen (receiver.status == SYNRECD) -> {
    receiver.status = ESTABLISHED;
};

/∗ --------------------------------------------------------------- ∗/
/∗ ACK of SYN From ∗/
/∗ --------------------------------------------------------------- ∗/

event (TCP(p) & Ack(s) & From(t))
OccurredWhen ((receiver.status == CLOSED) ||
    (receiver.status == C_SYNRECD) ||
(receiver.status == SYNSENT) ||
(receiver.status == SYNRECD)) ->
TCP_Error(hs) WithAttributes
{hs = "9:Ack_sent_in_bad_state"};

event (TCP(p) & Ack(s) & From(t))
OccurredWhen (((p.tcp_ackno != receiver.wbase + 1) &&
(p.tcp_ackno != receiver.fin + 1)) ||
(p.tcp_ackno < sender.lastack)) ->
TCP_Error(hs) WithAttributes
{hs = "10:Ack_sent_with_bad_ackno"};

transition (TCP(p) & Ack(s) & From(t))
OccurredWhen (receiver.status == S_SYNRECD) -> {
receiver.status = SYNRECD;
};

transition (TCP(p) & Ack(s) & From(t))
OccurredWhen (receiver.status == S_ESTAB) -> {
receiver.status = ESTABLISHED;
};

transition (TCP(p) & Ack(s) & From(t))
OccurredWhen (sender.status == SYNRECD) -> {
sender.status = ESTABLISHED;
};

/* ************************************************************ *
/*                         FIN To                             */
/* ************************************************************** *

event (TCP(p) & Fin(s) & To(t))
OccurredWhen ((sender.status != SYNRECD) &&
(sender.status != CLOSEWAIT) &&
(sender.status != ESTABLISHED)) ->
TCP_Error(hs) WithAttributes
{hs = "11:Fin_sent_in_bad_state"};

transition (TCP(p) & Fin(s) & To(t))
OccurredWhen (sender.status == SYNRECD) -> {
sender.status = FINWAIT1;
};

transition (TCP(p) & Fin(s) & To(t))
OccurredWhen (sender.status == ESTABLISHED) -> {
sender.status = FINWAIT1;
};

transition (TCP(p) & Fin(s) & To(t))
OccurredWhen (sender.status == CLOSEWAIT) -> {
sender.status = LASTACK;
};

180
transition (TCP(p) & Fin(s) & To(t))
    OccurredWhen (receiver.status == ESTABLISHED) -> {
        receiver.status = E_CLOSEWAIT;
        receiver.fin = p.tcp_seqno + p.tcp_dlen;
    }

transition (TCP(p) & Fin(s) & To(t))
    OccurredWhen (receiver.status == FINWAIT1) -> {
        receiver.status = F_CLOSING;
        receiver.fin = p.tcp_seqno + p.tcp_dlen;
    }

transition (TCP(p) & Fin(s) & To(t))
    OccurredWhen (receiver.status == FINWAIT2) -> {
        receiver.status = F_CLOSED;
        receiver.fin = p.tcp_seqno + p.tcp_dlen;
    }

/* *******************************************************/
/* FIN From */
/* *******************************************************/

event (TCP(p) & Fin(s) & From(t))
    OccurredWhen ((receiver.status != SYNRECD) ||
        (receiver.status != CLOSEWAIT) ||
        (receiver.status != ESTABLISHED)) ->
    TCP_Error(hs) WithAttributes
    {hs = "12:Fin.sent.in_bad_state"};

transition (TCP(p) & Fin(s) & From(t))
    OccurredWhen (receiver.status == SYNRECD) -> {
        receiver.status = FINWAIT1;
    }

transition (TCP(p) & Fin(s) & From(t))
    OccurredWhen (receiver.status == ESTABLISHED) -> {
        receiver.status = FINWAIT1;
    }

transition (TCP(p) & Fin(s) & From(t))
    OccurredWhen (receiver.status == CLOSEWAIT) -> {
        receiver.status = LASTACK;
    }

transition (TCP(p) & Fin(s) & From(t))
    OccurredWhen (sender.status == ESTABLISHED) -> {
        sender.status = E_CLOSEWAIT;
        sender.fin = p.tcp_seqno + p.tcp_dlen;
    }

transition (TCP(p) & Fin(s) & From(t))
OccurredWhen (sender.status == FINWAIT1) -> {
    sender.status = FCLOSING;
    sender.fin = p.tcp_seqno + p.tcp_dlen;
};

transition (TCP(p) & Fin(s) & From(t))
    OccurredWhen (sender.status == FINWAIT2) -> {
        sender.status = FCLOSED;
        sender.fin = p.tcp_seqno + p.tcp_dlen;
    };

/* ****************************************** */
/* ACK of FIN To */
/* ****************************************** */

transition (TCP(p) & Ack(s) & To(t))
    OccurredWhen (sender.status == ECLOSEWAIT) -> {
        sender.status = CLOSEWAIT;
    };

transition (TCP(p) & Ack(s) & To(t))
    OccurredWhen (sender.status == FCLOSING) -> {
        sender.status = CLOSING;
    };

transition (TCP(p) & Ack(s) & To(t))
    OccurredWhen (sender.status == FCLOSED) -> {
        sender.status = CLOSED;
    };

transition (TCP(p) & Ack(s) & To(t))
    OccurredWhen ((receiver.status == FINWAIT1) &&
            (p.tcp_ackno == sender.fin + 1)) -> {
        receiver.status = FINWAIT2;
    };

transition (TCP(p) & Ack(s) & To(t))
    OccurredWhen ((receiver.status == CLOSING) &&
            (p.tcp_ackno == sender.fin + 1)) -> {
        receiver.status = CLOSED;
    };

transition (TCP(p) & Ack(s) & To(t))
    OccurredWhen ((receiver.status == LASTACK) &&
            (p.tcp_ackno == sender.fin + 1)) -> {
        receiver.status = CLOSED;
    };

/* ****************************************** */
/* ACK of FIN From */
/* ****************************************** */
transition (TCP(p) & Ack(s) & From(t))
    OccurredWhen (receiver.status == E_CLOSEWAIT) -> {
        receiver.status = CLOSEWAIT;
    };

transition (TCP(p) & Ack(s) & From(t))
    OccurredWhen (receiver.status == F_CLOSING) -> {
        receiver.status = CLOSING;
    };

transition (TCP(p) & Ack(s) & From(t))
    OccurredWhen (receiver.status == F_CLOSED) -> {
        receiver.status = CLOSED;
    };

transition (TCP(p) & Ack(s) & From(t))
    OccurredWhen (receiver.status == S_ESTAB) -> {
        receiver.status = ESTABLISHED;
    };

transition (TCP(p) & Ack(s) & From(t))
    OccurredWhen ((receiver.status == FINWAIT1) &&
        (p.tcp_ackno == receiver.fin + 1)) -> {
        sender.status = FINWAIT2;
    };

transition (TCP(p) & Ack(s) & From(t))
    OccurredWhen ((receiver.status == CLOSING) &&
        (p.tcp_ackno == receiver.fin + 1)) -> {
        sender.status = CLOSED;
    };

transition (TCP(p) & Ack(s) & From(t))
    OccurredWhen ((receiver.status == LASTACK) &&
        (p.tcp_ackno == receiver.fin + 1)) -> {
        sender.status = CLOSED;
    };

// *******************************************************/
// RESET To
// *******************************************************/

event (TCP(p) & Reset(s) & To(t))
    OccurredWhen ((receiver.status == SYNSENT) &&
        (p.tcp_ackno != sender.syn + 1)) ->
    TCP_Error(hs) WithAttributes
    {hs = "13:Bad reset reply to Syn"};

event (TCP(p) & Reset(s) & To(t))
    OccurredWhen ((receiver.status != SYNSENT) &&
        ((p.tcp_seqno < receiver.wbase + 1) ||
(p.tcp_seqno > receiver.wbase + receiver.wsize))) ->
TCP_Error(hs) WithAttributes
{hs = "14:Reset_outside_window"};

transition (TCP(p) & Reset(s) & To(t)) -> {
    sender.status = CLOSED;
};

transition (TCP(p) & Reset(s) & To(t)) -> {
    receiver.status = CLOSED;
};

/* ************************************************** */
/* RESET From */
/* ************************************************** */

event (TCP(p) & Reset(s) & From(t))
  OccurredWhen ((sender.status == SYNSENT) & &(p.tcp_ackno != receiver.syn + 1)) ->
    TCP_Error(hs) WithAttributes
    {hs = "15:Bad_reset_reply_to_Syn"};

event (TCP(p) & Reset(s) & From(t))
  OccurredWhen ((sender.status != SYNSENT) & &(p.tcp_seqno < sender.wbase + 1) & &(p.tcp_seqno > sender.wbase + sender.wsize)) ->
    TCP_Error(hs) WithAttributes
    {hs = "16:Reset_outside_window"};

transition (TCP(p) & Reset(s) & From(t)) -> {
    receiver.status = CLOSED;
};

transition (TCP(p) & Reset(s) & From(t)) -> {
    sender.status = CLOSED;
};

/* ************************************************** */
/* DATA To */
/* ************************************************** */

transition TCP(p) & Data(d) & To(t) -> {
    bool tmpcopy;
    int i;
    tmpcopy = false;
    i = 0 ;
    receiver.unacked = receiver.unacked + 1;
while ($(i < receiver.segments#length) &&
(receiver.segments[i].beg <= p.tcp_seqno)) do {
    if (receiver.segments[i].beg == p.tcp_seqno)
        then {tmpcopy = true;}
    else {tmpcopy = false;}
};
if (tmpcopy == false) then {
    int i;
    push(receiver.segments, 1);
    i = receiver.segments#length - 2;
    while ($(i >= 0) && (receiver.segments[i].beg > p.tcp_seqno)) do {
        receiver.segments[i+1].beg = receiver.segments[i].beg;
        receiver.segments[i+1].dlen = receiver.segments[i].dlen;
        receiver.segments[i+1].data = receiver.segments[i].data;
    };
    receiver.segments[i+1].beg = p.tcp_seqno;
    receiver.segments[i+1].dlen = p.tcp_dlen;
    receiver.segments[i+1].data = p.tcp_data;
} else {}
if (tmpcopy == false) then {
  int i;
  push(sender.segments,1);
  i = sender.segments#length - 2;
  while ((i >= 0) && (sender.segments[i].beg > p.tcp_seqno)) do {
    sender.segments[i+1].beg = sender.segments[i].beg;
    sender.segments[i+1].dlen = sender.segments[i].dlen;
    sender.segments[i+1].data = sender.segments[i].data;
  };
  sender.segments[i+1].beg = p.tcp_seqno;
  sender.segments[i+1].dlen = p.tcp_dlen;
  sender.segments[i+1].data = p.tcp_data;
} else {}
}

event (TCP(p) & Data(d) & From(t))
  OccurredWhen (sender.unacked > 1) ->
    TCP_Error(e)
    WithAttributes
    {e = "19:Too_many_unacked_packets";};

event (TCP(p) & Data(d) & To(t))
  OccurredWhen ((p.tcp_seqno < lastack) ||
    (p.tcp_seqno >= sender.lastack + sender.wsize) ||
    ((sender.fin != 0) &&
      (p.tcp_seqno > sender.fin))) ->
    TCP_Error(e)
    WithAttributes
    {e = "20:Data_sent_outside_window";};

/∗ ********************** Ack of DATA To ∗/
/∗ ******************** Ack of DATA To  ∗/
event TCP(p) & Ack(a) & From(t) ->
ToData(d) WithAttributes {
    int i;
    d.wbase = p.tcp_ackno;
    while ((i < sender.segments#length) &
            (receiver.segments[i].beg < p.tcp_ackno)) do {
        push(d.segments,1);
        d.segments[d.segments#length-1].data = receiver.segments[i].data;
    };
}

transition TCP(p) & Ack(a) & From(t) -> {
    receiver.lastack = p.tcp_ackno;
    receiver.wsize = p.tcp_win;
    receiver.unacked = 0;
};

event ToData(d) OccurredWhen (d.segments#length > 0) ->
TCP_Data(t) WithAttributes {
    t.ip_src = ip.src;
    t.ip_dst = ip.dst;
    t.tcp_sport = tcp_sport;
    t.tcp_dport = tcp_dport;
    t.to = true;
    t.from = false;
    t.segs = d.segments;
};

event FromData(d) OccurredWhen (d.segments#length > 0) ->
TCP_Data(t) WithAttributes {
    t.ip_src = ip.src;
    t.ip_dst = ip.dst;
    t.tcp_sport = tcp_sport;
    t.tcp_dport = tcp_dport;
    t.to = false;
    t.from = true;
    t.segs = d.segments;
};

transition ToData(d) OccurredWhen (d.segments#length > 0) -> {
    int i; i = 0;
    receiver.wbase = d.wbase;
    while (i < receiver.segments#length-d.segments#length) do {

D.2 Proof of Counting Property Optimization

In this section we prove the correctness of the optimized co-monitoring algorithm for counting properties (Table 10.4). We carry out a reduction from the brute-force algorithm in Chapter 7 (Table 7.2) to the counting algorithm. The reduction proceeds by defining a mapping between the two state spaces and showing that it is maintained when each algorithm takes a step in response to the same event.

**Definition D.1 (iqtail)** Let iqtail(\(\omega\)) equal the sequence of iq elements in \(\omega\) that are prior to the last oq element but which were not consumed before the oq, that is, which do not have a corresponding id element before the oq, plus all iq’s that appear after the last oq element, regardless of whether the corresponding id is in \(\omega\). If \(\omega\) does not contain any oq’s, then iqtail(\(\omega\)) is just the sequence of iq’s in \(\omega\).

**Definition D.2 (idtail)** Let idtail(\(\omega\)) equal the sequence of id elements in \(\omega\) that appear after the last od element. If \(\omega\) does not contain any od’s, then idtail(\(\omega\)) is just the sequence of id’s in \(\omega\).

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

\[ e \iff (\forall i : \text{buf}_\text{min} \leq i \leq \text{buf}_\text{max} \iff (\exists (\omega, b) \in \Omega :: |\text{iqtail}(\omega)| = i)) \]

\[ \land e \iff \Omega = \emptyset \]

**Base Case.** Initially, \(\Omega\) contains only one element, which has a zero length \(\omega\) and a zero length \(b\). The initial conditions for \(\text{buf}_\text{max}', \text{buf}_\text{min}', \) and \(e\) are all consistent with the state space mapping.
**Inductive Step.** Consider the input actions in each algorithm corresponding to \( iq \) events. In Table 7.2, this action deletes each member of \( \Omega \) and adds a new member with the \( iq \) action added to both \( \omega \) and \( b \), as long as \( \omega \) is admissible and its projection \( [\omega]_{id,od} \) is in \( g \). In Table 10.4, an input event increments both \( buf_{min} \) and \( buf_{max} \), checks for an error condition, and then adjusts \( buf_{max} \) if necessary. We need to show that the state space mapping still holds. First, note that if any \( (\omega, b) \in \Omega \) has an \( \omega \) such that \( |iqtai(\omega)| > B + c_{max} \), it will be deleted from \( \Omega \) because the result of adding the \( iq \) to \( \omega \) will be inadmissible according to \( M' \), which recall is equal to \( M(S, m+c+n, 0) \). In this case \( c = c_{max} \) and \( B = m+c+n \). \( \omega \) will be inadmissible because at most \( c_{max} \) id's can appear after the last \( od, oq \) by way of property P1. The remaining \( B \) iq's fill up the buffer and adding one more causes it to overflow, removing the resulting string from consideration.

If \( buf_{min} \geq B + c_{max} \) before the input action, this implies (by the inductive hypothesis) that every \( (\omega, b) \in \Omega \) has an \( iqtai(\omega) \) whose size is at least \( B + c_{max} \). When the \( iq \) event is added, all will be deleted and \( \Omega \) will be empty. This corresponds to setting \( c \) to true in Table 10.4. If \( buf_{min} < B + c_{max} \) before the input action, then there is at least one \( (\omega, b) \in \Omega \) such that \( |iqtai(\omega)| = i \) for every \( i \) between \( buf_{min} \) and \( buf_{max} \). Each will be removed and replaced with an \( (\omega, b) \) pair with exactly one \( iq \) event added to each of \( \omega \) and \( b \), except for those that become inadmissible with respect to \( M' \). This corresponds exactly to \( buf_{min} \) and \( buf_{max} \) being incremented by one, and \( buf_{max} \) being capped at \( B + c_{max} \) in Table 10.4. The iteration step 2 in Table 7.2 has no effect on the mapping because it only adds id events to \( \omega \) and does not modify \( iqtai(\omega) \). Note, however, that this iteration guarantees that every pattern of \( iqtai(\omega) \) such that \( 0 \leq |iqtai(\omega)| \leq \min(c_{max}, buf_{max}) \) will be generated. This is because each element of the \( iqtai(\omega) \) will be converted to an \( id \) and check-added to \( \Omega \). Those with more than \( c_{max} \) id's will be discarded, since they are not in \( g \), but all others will be accepted due to property P1. The longest \( iqtai \) has length \( buf_{max} \), leading to the upper bound.

Next, consider the output actions in each algorithm corresponding to the \( oq \) events. In Table 7.2, this action first deletes every element of \( \Omega \), replacing each with one that has the \( od, oq \) event pair added. Then the same iteration discussed previously is repeated. Note that if any \( (\omega, b) \) has an \( iqtai(\omega) < c_{min} \), it will be deleted because the result of adding the \( od \) will not be in \( g \), according to P1. In Table 10.4, if \( buf_{max} < c_{min} \), then all \( iqtai(\omega) \) must have length less than \( c_{min} \), because \( iqtai(\omega) < iqtai(\omega) < buf_{max} \). \( \Omega \) thus becomes empty after the first step. This corresponds to setting \( c \) to true in Table 10.4. Otherwise, Table 10.4 decrements \( buf_{max} \) by \( c_{min} \), but caps \( buf_{max} \) at \( B \) because even though \( iqtai \) can grow larger than \( B \), the new \( iqtai \) will be at most \( B \) because it consists of only the un-consumed \( iq \) events in \( \omega \), of which there can be at most \( B \). The algorithm also decrements \( buf_{min} \) by \( c_{max} \), setting it to zero if it goes negative. This corresponds to the fact that pairs \( (\omega, b) \) will be deleted from \( \Omega \) if they violate the \( c_{min} \) and \( c_{max} \) limits given by \( g \), which means that the new minimum and maximum values of \( iqtai(\omega) \) will be given when the maximum number of inputs are consumed from the minimum previous \( iqtai \) and the maximum number of inputs are consumed from the maximum previous \( iqtai \). These consumptions are justified because we have a guarantee that every pattern \( iqtai(\omega) \), such that \( 0 \leq |iqtai(\omega)| \leq \min(c_{max}, buf_{max}) \), will have been generated at the end of the previous \( iq \) or \( oq \) event.

189
Bibliography


191


Kevin Fall and Kannan Varadhan. ns Notes and Documentation. The VINT Project, February 2000.


