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

2003

Supervisor of Dissertation

Graduate Group Chairperson

To Amma and Appa

ACKNOWLEDGEMENTS

I am deeply grateful to Prof. Carl Gunter for his guidance during my doctoral research. Carl has been an excellent supervisor and co-author; he has always encouraged my ill-formed ideas and helped me develop them into research projects. This dissertation could not have been completed without his support.

I am grateful to Davor Obradovic, who partnered me in my early research on protocol verification. Davor has co-authored several of my publications. Our discussions on protocol testing served to motivate this dissertation.

I am grateful to the Monitoring and Checking group at Penn. Prof. Insup Lee, Prof. Oleg Sokolsky, Prof. Sampath Kannan, Prof. Mahesh Vishwanathan, and Dr. Moonjoo Kim, together with Davor, Carl, and me, developed the Verisim monitoring tool that resulted in the AODV case study. Their language, MEDL, directly inspired the design of NERL.

I am grateful to Dr. Satish Chandra and Dr. Peter McCann, who helped me understand the problem of trace infidelity in protocol monitoring. Our joint work and subsequent publication motivated the TCP case study and the channel transformation feature of NERL.
I am also grateful to Satish for his comments and suggestions on improving this dissertation.

I am deeply grateful to Prof. Benjamin Pierce for his detailed and insightful comments on earlier versions of this dissertation. Benjamin has bravely tried to help me understand the

importance of good research writing. I feel I am making slow progress although I still have much to learn.

I am grateful to the thesis committee for their constructive comments. Dr. Satish Chandra, Prof. Michael Greenwald, Prof. Sampath Kannan, and Prof. Benjamin Pierce helped me iron out several deficiencies in an earlier draft of this dissertation.

I am grateful to my friends and family for their support and encouragement. I am grateful to my sister, Mythreyi Bhargavan, and friends, Annapurna Valluri and Ninad Prabhu, who helped me proofread this document.

Karthikeyan Bhargavan
Philadelphia
December, 2003

ABSTRACT

Network Event Recognition

Karthikeyan Bhargavan

Supervisor: Carl A. Gunter

This dissertation demonstrates and evaluates the use of passive run-time monitoring to test black-box implementations of network protocols for conformance with their published specifications. Our goal is to design a flexible, programmable, passive protocol monitoring system and apply it to the analysis of a variety of network protocol implementations in diverse monitoring environments.

Passive monitoring has not been successful as a protocol testing technique because of several deficiencies in existing monitoring systems. Run-time software monitoring frameworks lack the domain-specific constructs to express protocol specifications. Passive testing systems based on non-deterministic protocol specifications are inefficient. Network intrusion detection systems are susceptible to false positives due to incorrect or incomplete protocol modeling. Error diagnostics provided by many of these systems are inadequate for debugging.

None of these systems can accurately monitor network links where messages can be lost and delayed.

We introduce a new passive protocol monitoring framework, Network Event Recognition, that provides analysis tools and techniques to combat these deficiencies. The framework consists of a domain-specific language, NERL, and three automated tools for monitor programs written in NERL. To guarantee correctness and completeness, NERL programs can be translated to a formal model and analyzed using a model checker. To provide diagnostics, NERL monitors can compute and print out the relevant event history for every error event. To account for non-deterministic packet loss and delay, NERL programs can be transformed to monitors that incorporate these network characteristics.

We evaluate the effectiveness of our methodology through three case studies and find new errors in existing protocol implementations. First, we analyze network simulations of a wireless routing protocol, AODV, and find significant flaws in both the prototype implementation and the AODV standard. Second, we analyze live sessions of SMTP mail servers and find several flaws in popular mail server software. Third, we analyze TCP packet traces produced by three popular operating systems and confirm a defect in two of them.

We establish that Network Event Recognition is an effective and widely applicable protocol testing technique. We conclude that passive monitoring of protocol implementations against a formal specification can significantly improve the reliability of networked applications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 misconfigured 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 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 (IETF,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 interoperate 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. 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 misconfigurations.

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. If the standard is incorrect, then the standards group must be notified. A misbehaving user may indicate a malicious attacker from whom private assets must be protected, and network misconfigurations 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*. In contrast to protocol standards and implementations, users and network configurations are evident only at run-time and can vary depending on user behavior patterns, network load, and network topology. This makes them unpredictable and 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 misbehavior or misconfiguration is detected by *network management* software. While this dissertation primarily concerns the protocol testing problem, some of the techniques we develop are applicable to intrusion detection and network management as well.

Traditionally, protocols have been tested using *active* techniques. A test network topology is constructed and artificial users are programmed. 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; active techniques are inapplicable for testing implementations in the field after deployment.

Passive Protocol Monitoring *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. As a result, when an implementation malfunctions, it is possible to find the cause of the error and propose a corrective measure as feedback. This technique has been used to good effect in the telecommunications industry for 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 does not affect the performance of the implementation. 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), they have rarely been used for protocol testing because of the difficulty of programming such monitors. Executing an active test is simple—feed the protocol implementation a sequence of inputs and compare the observed output sequence with the expected output sequence that has been computed beforehand. In contrast, a passive monitor has no control on the inputs and so cannot precompute the outputs; it must observe the inputs and compute the expected outputs on the fly. Moreover, the passive monitor has limited visibility since it can see only the low-level messages on the network; it cannot observe the high-level interactions between the implementation and the user or examine the internal state of the implementation.

To write a passive monitor, a programmer must first identify the parts of the protocol that need to be checked and then extract the state machine corresponding to the visible actions for these parts. The monitor program will take protocol messages as input and check them for conformance with this state machine. If the protocol is at a high layer, the monitor must also reconstruct events for all the lower layer protocols. If the resulting monitor is incorrect or incomplete, it may signal errors when there are none. Such *false positives* are

a significant drain on human resources and are considered a serious limitation for network monitors (PN98).

Another limitation that is unique to the network monitoring domain is *trace infidelity*. In some network conditions, because of packet loss and delay, the message sequence observed by a monitor is not the same as the actual sequence of events at the implementation under test. Previous monitoring studies (Pax97) have failed in the presence of trace infidelity, and overcoming this limitation requires a significant and careful reprogramming of the monitor.

A final limitation is that an effective monitor must provide good diagnostics. Most network monitoring systems flag an error and provide no further information about its cause. This means that the analyst must wade through the packet trace manually to try and map the error first to a violation of the specification and then to a flaw in the implementation. An effective passive monitor must be programmed to print as much diagnostic information as possible with an error event. This can be tedious and time-consuming.

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

Expressiveness. The ability to write a monitor to test a protocol at any layer in the protocol stack for any subset of the protocol specification. In particular, the ability to model message formats as well as protocol layering.

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

Correctness. The ability to correlate the monitor program with the protocol specification, and to guarantee that the errors produced by the monitor indicate violations of the specification.

Diagnostics. The ability to provide enough information with error events to guide the analyst toward flaws in the implementation.

Monitoring with Infidelity. The ability to monitor a protocol session even when the observed trace differs, up to a limit, from the actual trace at the implementation under test.

Existing passive monitoring systems do not fully satisfy these requirements. *Run-time verification* frameworks are used to monitor and check general software for safety properties. For instance, the MEDL language has been used to check Java programs for properties like mutual exclusion (LKK+99). Although such languages can be uses to program protocol monitors (BGK+02), they lack the domain-specific constructs to model messages and protocol layering. Formal description techniques, such as LOTOS (Lot87), have expressive languages that have are used to write protocol specifications. However, *passive testing systems* based on such protocol specifications are inefficient because they must account for the non-deterministic choices allowed by the specification. For instance, the TETRA passive testing system (BB89) performs a state-space search of the specification to check that an implementation trace is correct. As a result, its performance is inadequate for live network monitoring. *Network intrusion detection systems*, such as Bro, are highly efficient because they are engineered to look for bad protocol participants when the network is heavily loaded and under attack. However, intrusion detection systems are typically written in programming languages that do not support formal analyses for correctness. For instance, protocol modules in Bro are usually written in C++, which offers little support for formal analysis. Moreover, most existing monitoring systems assume that the monitored channel has no packet loss or delay. When monitoring realistic network links, they produce a lot of false positives. Even the more sophisticated intrusion detection systems only try to detect network errors; they do not try to monitor in spite of them.

Network Event Recognition We design a passive protocol monitoring framework called Network Event Recognition that consists of a domain-specific language with automated tools and analysis techniques for protocol monitors.

We introduce the Network Event Recognition Language (NERL), a domain specific language for programming run-time monitors based on protocol specifications. NERL programs can be used to passively monitor protocol implementations in real-time as they execute on a live network. NERL programs can also be used to analyze packet traces captured from a network simulation or live network. A NERL program consists of several *protocol event recognizers* and a *main module*. The event recognizer for a protocol analyzes an event stream and produces *alarm* events when deviations from the protocol specification are detected. To monitor a protocol at a high layer in the stack, a NERL program contains recognizers for all the lower layers in order to reconstruct the messages at the protocol of interest. The main module in a NERL program specifies and manages such stacks of recognizers, where each recognizer may be written in NERL or C.

We demonstrate that monitors written in NERL satisfy all the requirements for effective passive monitoring. NERL is expressive and generates monitors that are efficient. The NERL compiler includes a *model-checking* feature to check the correctness of a monitor program with respect to the protocol specification. NERL monitors can also be compiled with an *event tracing* feature that provides important diagnostic information by computing the history of relevant events that caused an error. NERL monitors can be compiled with *channel transformations* that enable the monitor to work even in the presence of trace infidelity.

To evaluate the design and implementation of NERL and its effectiveness for passive monitoring, we perform three case studies and find new errors in existing protocol implementations. First, we analyze network simulations of a wireless routing protocol, AODV, and find significant flaws in both the prototype implementation and the AODV standard. Second, we analyze live sessions of mail servers running SMTP and find several deviations from the protocol in popular mail server software. Third, we analyze packet traces produced by the TCP implementations in three popular operating systems and identify a minor defect in two of them.

We establish that NERL satisfies the requirements set out for passive monitoring and can be used to effectively monitor protocols at several different layers in diverse network environments.

1.1 Summary of Contributions

This dissertation makes several novel contributions to research in network protocol testing and monitoring. Our main contributions are as follows:

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

- **NERL:** We present a new domain-specific language for programming specification-based run-time monitors for network protocols. NERL represents an advance over previous approaches for protocol monitoring. It simplifies the programming task by automatically generating code for common protocol monitoring tasks. It enables the programmer to write monitors that are more efficient than monitors generated from protocol specifications.
- **Monitor Model Checking:** We present a tool that checks whether monitors written in NERL are consistent with respect to a protocol specification. Our tool translates NERL programs to formal models and checks them for correctness using the SPIN model-checker. This is a novel application of model-checking in run-time verification.
- **Event Tracing:** We present a new technique for providing automated diagnostics for a protocol monitor. Our technique computes and presents the causal event history for every error detected by the monitor.

Channel Transformations: We present a new technique for automatically generating monitors that account for trace infidelity. Our technique is based on a novel trace-search algorithm and is applicable to other passive monitoring systems.

Protocol Testing. We demonstrate new passive protocol testing techniques applicable to both simulation analysis and live network 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). Our framework, Verisim, is the first specification-based framework that tests simulator implementations of network protocols.

Tuning: We present a novel technique for extracting multiple errors from a single packet trace.

Fault Origin Adjudication: We present a novel technique that uses multiple tests on a single packet trace to differentiate between errors in implementation and errors in specification.

Case Studies. We analyze protocols at three different layers in the Internet stack and find errors of interest to protocol developers and implementors. The variety of case studies illustrates the flexibility and expressiveness of NERL. Each case study evaluates the efficiency of monitors in different conditions. All case studies demonstrate the value of event tracing as a diagnostic tool.

AODV: We use NERL to find five different flaws in the simulator implementation of AODV. In addition, we find one flaw in the AODV (version 0) standard. We demonstrate the effectiveness of tuning and fault origin adjudication.

SMTP: We use NERL to find six different flaws in the SMTP implementations of three popular mail servers. We demonstrate the benefits of model-checking SMTP monitors.

TCP: We use NERL to confirm the existence of one flaw in two TCP implementations and its absence in a third implementation. We demonstrate the effectiveness of channel transformations for the co-networked monitoring of TCP implementations.

1.2 Outline

The rest of this dissertation 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. We precisely define passive protocol monitoring and list its advantages and disadvantages with respect to other testing techniques. We discuss the monitoring environments in which passive monitors are expected to operate and identify the key requirements for effective protocol monitoring. Then, we survey related research that parallels or motivates our work. Readers familiar with the protocol design process may skip this chapter.

In Chapter 3, we formally define the problem addressed by this dissertation and summarize our goals, methodology, and evaluation strategy.

Chapter 4 introduces the Network Event Recognition Language. This chapter is structured as a programmer's manual. We present the syntax and semantics for NERL through a series of examples. A detailed reference manual for NERL is included as Appendix A.

In Chapter 5, we describe the NERL implementation. We outline the implementation of the compiler for NERL recognizers and main modules. In addition, we describe compiler features that implement various requirements such as correctness, diagnostics, and monitor-

ing with infidelity. 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 these features will become clear only in the case studies. So, we recommend that the reader skim Chapter 5 and go on to read the case studies first, referring back to Chapter 5 as new features are introduced.

The next three chapters contain case studies. In Chapter 6, we analyze simulation traces of the AODV wireless routing protocol. In Chapter 7, we analyze live networks running the SMTP mail delivery protocol. In Chapter 8, we analyze packet traces of TCP implementations. All chapters demonstrate the use of diagnostic features in NERL. In addition, Chapter 7 exercises the model-checking feature, while Chapter 8 demonstrates the effectiveness of channel transformations.

Finally, in Chapter 9, we evaluate our results, state our conclusions about the effectiveness of passive protocol monitoring, 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* (MTAs), 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 in 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 Simple Mail Transport Protocol (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. This ensures that standard-conformant implementations will interoperate correctly to provide the required service.

We are interested in checking whether a network protocol implementation actually provides the service that it claims to. For proprietary implementations, often the only information available about implementation behavior is the sequence of standardized messages exchanged between protocol participants. We 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, in some cases, cause it to fail to deliver email. We want to use the SMTP message sequence 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 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 Section 2.1, 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. We describe our monitoring model and identify specific requirements for passive monitoring to be an effective testing technique. In Section 2.4, we overview existing passive monitoring systems. We evaluate the designs of some representative systems and demonstrate that they

do not satisfy our requirements. Finally, Section 2.5 outlines our earlier work that motivated and contributed to our 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 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 the Internet Protocol (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 Open Systems Interconnection (OSI) model (Zim80). 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.

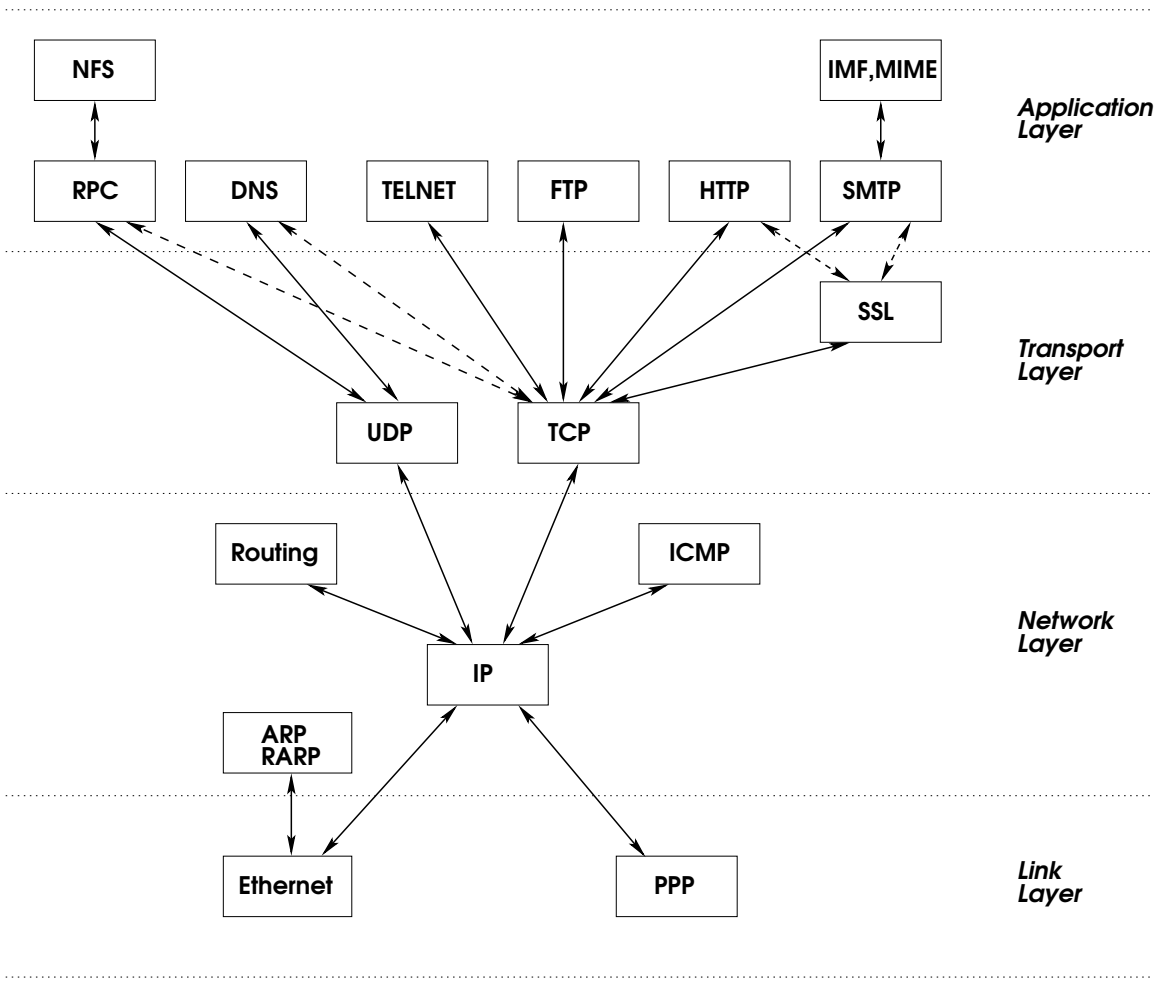

Figure 2.1 An Internet Protocol Stack

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 network device 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, the Point-to-Point Protocol (PPP) is commonly used across links such as phone lines.

Network Layer. An internetwork (such as the Internet) consists of several networks of devices, where some devices (such as 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 IP and a suite of associated protocols that are used to gather information about network topology and errors. An IP *routing protocol*, such as the Open Shortest Path First (OSPF) protocol, tries 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 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. The Address Resolution Protocol (ARP) carries out this translation for Ethernet. To communicate network layer connectivity and error

information, routers and endpoints use the Internet Control Message Protocol (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, overload 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. The Secure Sockets Layer (SSL) 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. The User Datagram Protocol (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, which 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, which 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 is a service that several competing protocols provide. 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 focus 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, and wireless. On the other hand, there is only one network layer protocol of note, IP. 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, and 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 on which it depends. 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 following section, we 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

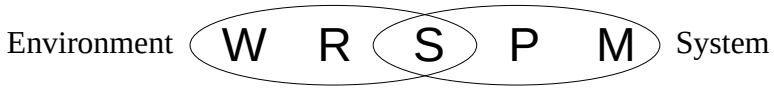

Figure 2.2 Five Software Artifacts

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 assumptions 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 that 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 *com-*

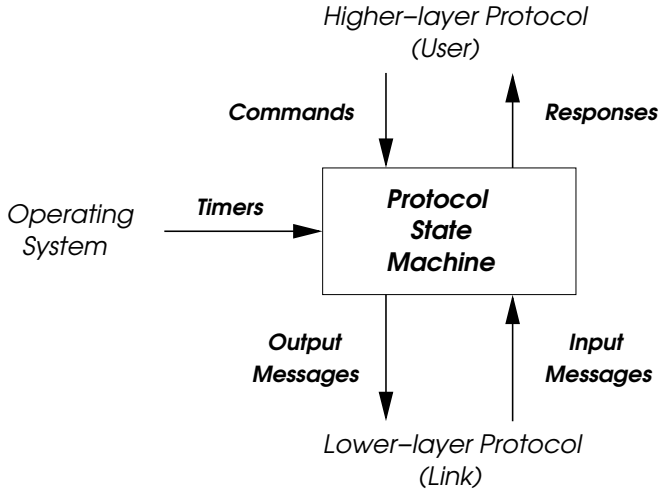

Figure 2.3 Protocol Inputs and Outputs

mands 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 .

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. 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. 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 (that is, W, P, and S provide R), and (2) P conforms to the specification S (that is, 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 P violates the assumptions W.

Link Failure The link (lower-layer protocol) below P fails to provide the services 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, the specification may 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. Implementations are too complex (have too many states) to carry out a complete proof of conformance and functionality. Specifications, however, 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). Further, 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 difficult when compared with testing sequential programs. 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 large and varied, and most testing studies are for particular well-known protocols such as TCP. 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 discussed (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 FeaVer (GS02).

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 automati-

cally (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 protocol implementations.

2.2.2 Live Network vs. Simulation

In a live network 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.

In contrast, 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 (Prob, 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. As a result, live testing is always needed 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 and 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_1 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. Active testing is a combination of test suite generation followed by passive analysis of the test outputs (BB89).

The chief advantage of active testing is its control over the test environment. As a result, an active test suite can 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 invisible to the IUT, it can analyze arbitrary traces, and it is 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 her control (LNS+97). We describe relevant passive testing research in more detail in Section 2.4.1.

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 designing a model for the inputs to the protocol, generating randomized input traces to which the protocol can respond, an checking the outputs 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, compares it with a formal model of how it should behave,

and produces 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. It is easy to run fast automated statistical tests that provide succinct results. In contrast, logical analysis is more cumbersome, takes more time, and sometimes provides a lot of extraneous information that the developer needs to manually analyze.

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 the Graphical Interval Logic (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. If the specification has been correctly designed, a conformant implementation is guaranteed to be functional. However, functionality testing is valuable when a formal specification either does not exist or is too complex for conformance testing. Testing functionality is often simpler, and can, sometimes, 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. In contrast, 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 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 for research, we restrict our attention in this dissertation 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 is 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. Moreover, the errors found by our tests are interpretable as violations of the formal specification. Hereafter, we refer to this class of testing as passive protocol monitoring.

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.

Programming a passive protocol monitor can be a challenging task because the input-output packet trace analyzed by the monitor is an incomplete representation of the execution of the protocol implementation. The packet trace contains only the communication events between protocol participants. And if there are several layers of protocols, it only contains messages sent between the lowest layer participants. The trace does not contain the commands sent by a user to the protocol implementation, or between two layers in the stack. It does not contain information about any hidden state that a protocol implementation may maintain. And if the network link between the device and monitor can delay or lose packets, then the trace does not even have a faithful record of communication events. As a result, a passive monitor can only analyze a trace for observable properties of the protocol—properties of

protocol state, timers, and higher-layer events that can be reconstructed from the messages. For a more formal treatment of observability, the reader is referred to the notion of fault observability in *fault secure* distributed systems (DJC94).

A further limitation of run-time monitors is that they can check a trace only for recursive, safety properties (Vis00). Liveness properties, such as 'a response is eventually produced for every command', cannot be checked on finite traces and are beyond our scope. This is not a serious limitation since most protocol liveness requirements typically have time-limits, such as 'a response is produced in 5s', and so can be cast as safety properties.

The most serious limitation of passive monitoring is that encrypted protocol sessions cannot be analyzed since they require knowledge of secrets known only to the protocol participants. As a result, a passive monitor will be unable to test protocols, such as SSL and SSH, that establish secure channels and only send encrypted data on the public network. In the rest of this thesis, we shall restrict ourselves to analyzing protocols that send all messages in the clear.

To better understand the challenges involved in programming passive protocol monitors, we first overview the network configurations in which the monitor is expected to be used. Then, we identify a set of requirements for an effective protocol monitoring framework.

2.3.1 Monitoring Environments

A typical protocol monitoring configuration is as depicted in Figure 2.4. The feasibility of monitoring depends crucially on the quality of the monitoring channel between the monitor and device under test (DUT). The monitor captures the packets as they go toward the DUT (inputs), and as they come out of the DUT (outputs). If the channel is perfect, the sequence of inputs and outputs are seen at the monitor exactly the same as the actual sequence at the DUT. We then say that the monitored trace has *perfect fidelity*. Unfortunately, in real-world networks, the monitored trace often fails to match the actual trace because of packet loss and

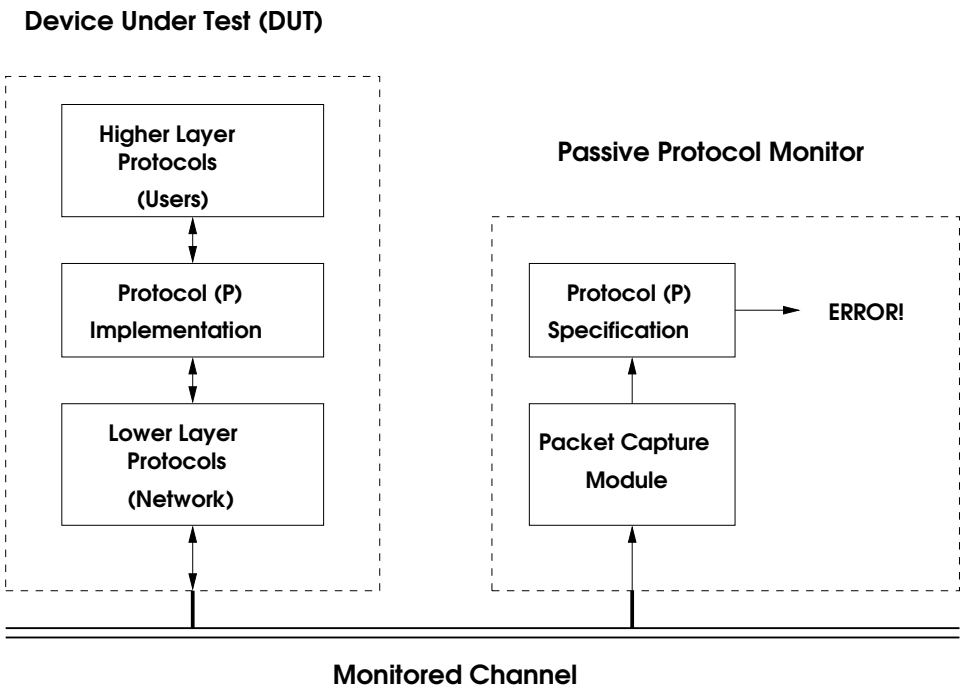

Figure 2.4 Passive Protocol Monitoring

delay between the monitor and DUT. We then say that the monitored trace has *infidelities*, or that the monitor experiences *trace infidelity*.

Trace Infidelity in IP Networks

Figure 2.5 depicts several common message sequences that result in infidelities when monitoring IP networks. We assume that a sender is trying to send an IP packet to a receiver and the monitor is trying to sniff the packet in transit. The send event is called PSENT, the receive event is called PRECV, and the monitor's capture event is called PSEEN. In the best case, the PSEEN is the same as the PSENT and the PRECV, and it occurs at exactly the same time (Figure 2.5,I).

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

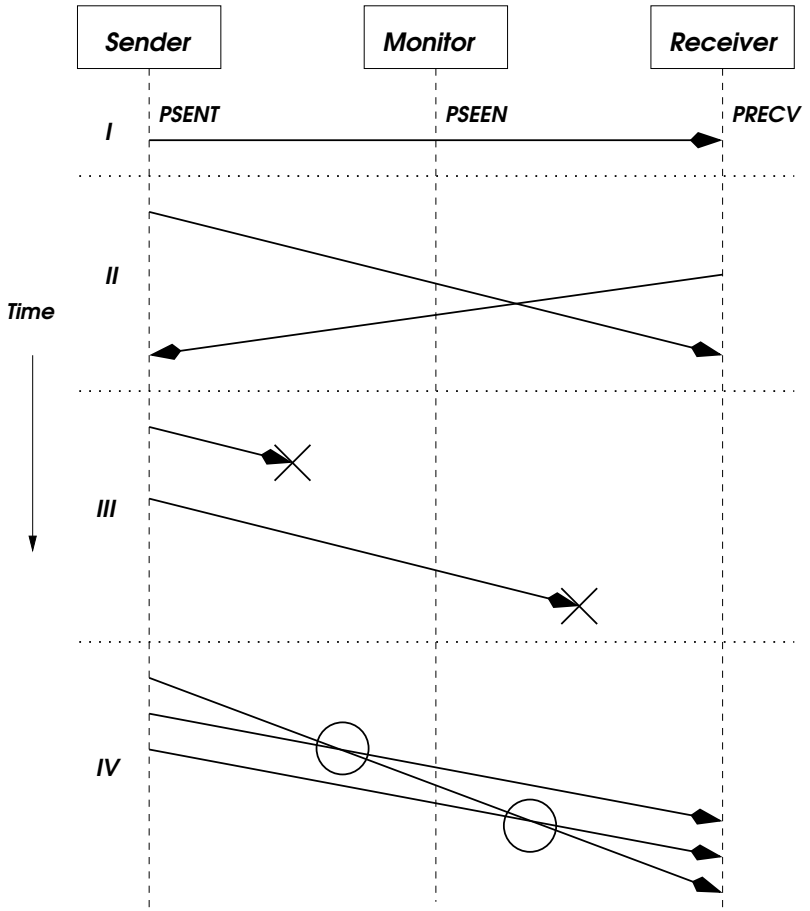

Figure 2.5 Monitoring Infidelities

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 2.5, 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 2.5, 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 2.5, 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.

In the presence of significant infidelities, protocol monitoring quickly becomes infeasible. This is because the protocol monitor 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.

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 monitors to be correct, they must be able to distinguish between infidelities and implementation errors.

The extent of these infidelities is primarily determined by the placement of the monitor with respect to the DUT. In general, the further the monitor is from the DUT, the more infidelities it will experience. Figure 2.6 depicts several possible locations for the monitor. Monitor *M1* is co-located with the device under test (DUT); it will enjoy perfect fidelity with

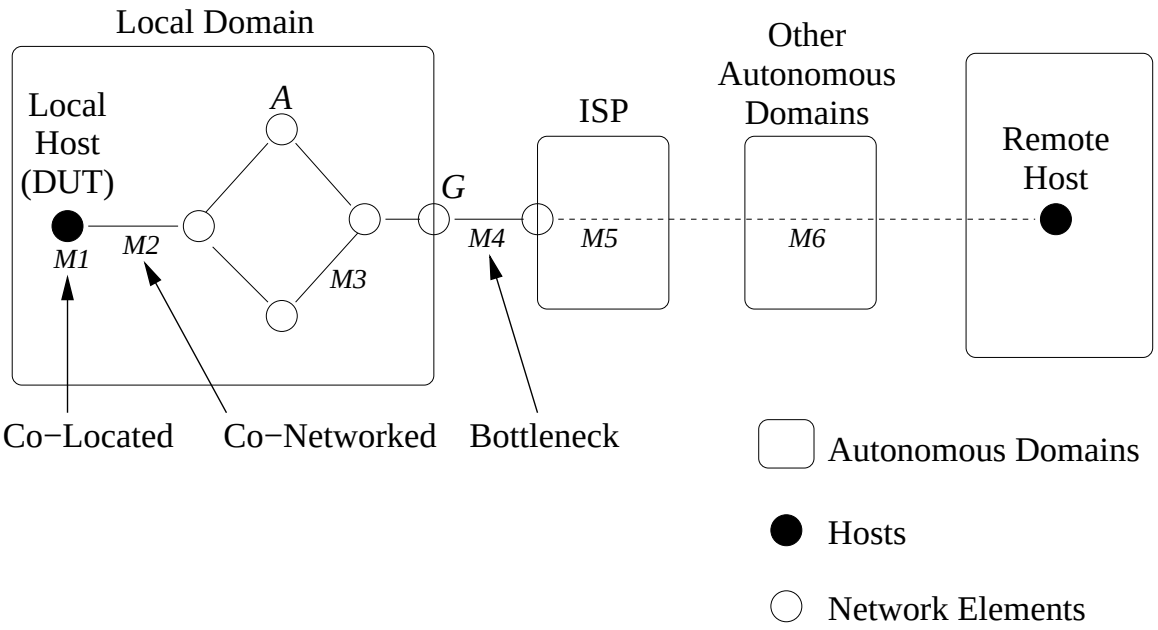

Figure 2.6 Monitor Placement

respect to the DUT. Monitor *M2* is co-networked and monitor *M4* 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 *M3* will not observe traffic passing through network element A. *M5* is located at an Internet service provider (ISP), and *M6* is located in another service provider's network. Of these, we identify co-located monitoring (M1), bottleneck monitoring (M4), and co-networked monitoring (M2) as feasible on IP networks.

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 6, we will describe passive monitoring 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 protocol being monitored runs on top 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. As a result, a protocol that runs above TCP can ignore lower-layer network er-

rors. Similarly, the monitor for a protocol above TCP can assume perfect fidelity, up to some delay, as long as the monitor correctly reconstructs TCP packets. To see why, assume that all protocol layers up to TCP are working correctly at both the sender and receiver. Then, TCP reliability implies that for every data segment S sent in a TCP session, there is a time T when both the sender and receiver agree that the data segment S has been successfully sent and received. So, by mimicking the operation of both the sender and the receiver, the monitor can conclude at time T that S was sent and S was received at some time before T. This information is enough to accurately reconstruct the events of protocol layers above TCP, as long as some delay is permitted. A concrete example of this kind of analysis is shown in Chapter 7.

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 testbed, a co-networked monitoring configuration is the best we can hope for. It is important that passive monitoring 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 when monitoring 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,

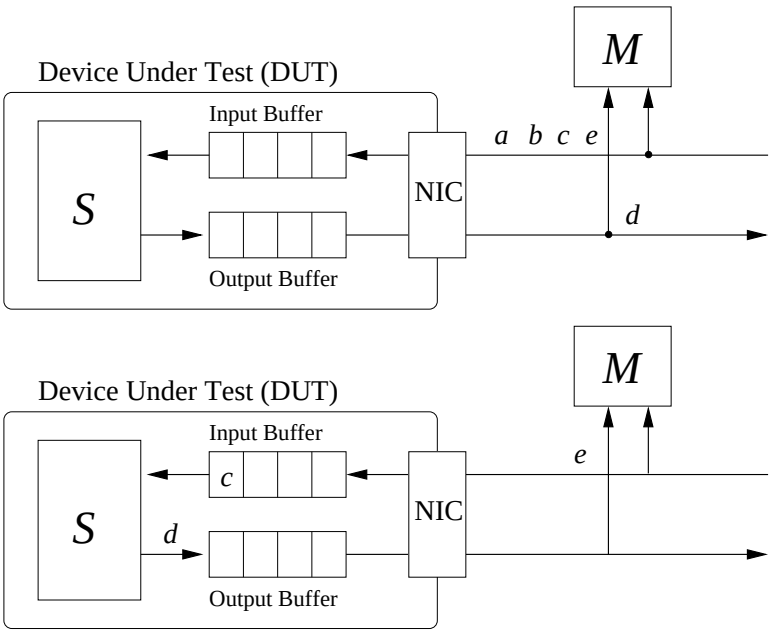

Figure 2.7 Buffers in the DUT

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 protocol participants. If a protocol S is being run by a 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 2.7. Inputs a, b, c, e go toward the device and output d is produced by the device. The monitor cannot tell which inputs were consumed before the device produced d. One possible scenario is that a, b were consumed by the device, c was buffered, and e was dropped. Several other scenarios are equally possible. These packet buffers are maintained by the operating system. For instance, the Linux ethernet drivers capture packets from the wire and hand them over to a routine 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 pro-

duce 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 phenomenon in some detail in Appendix B. To see the issue briefly, note that it is impossible for M to tell in Figure 2.7 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 occur only rarely and only 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.

We define a *co-networked* monitor as one that is placed so that it can observe all outputs produced by the device under test without delay and it can observe all inputs going toward the device, but the 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 5, we describe an algorithm that reconstructs events on the co-networked monitoring channel. Then in Chapter 8, we use this algorithm to execute a co-networked monitor for TCP and check the correctness of several TCP implementations.

2.3.2 Requirements for Programming Passive Monitors

To write a passive protocol monitor, the programmer must take a protocol specification and and write code that checks whether an implementation trace violates the specification. This programming task has several significant challenges. First, many protocol specifications allow for several different implementations. The programmer must convert such a non-deterministic specification into a deterministic monitor that can monitor all possible implementations. Second, the protocol specification will typically describe internal state and internal events that never become evident in the packet trace. The programmer must write a monitor that ignores properties that depend on this invisible information and still checks the trace for other visible properties. Third, if the protocol to be tested is layered on top of some other protocol, then the packets visible on the wire do not correspond to protocol messages. The programmer must write monitors for the lower layers that can reconstruct the high-level messages at the protocol of interest. Fourth, if the monitor is co-networked as described in the previous section, then the packet sequence on the wire is not the actual packet trace. The programmer must write a monitor that can check properties of a trace even if it has infidelities.

Given these challenges, it is unfair to expect a programmer to write correct passive monitors without any programming support. Ideally, a programming environment for protocol monitoring should provide libraries, analysis tools, and code generation tools to help the programmer do repetitive tasks quickly and complex tasks correctly. In this thesis, we design a new domain-specific language for writing protocol monitors, and we implement several such tools and libraries for programs written in this language. However, we can conceive of equally valid alternate approaches that add new libraries and tools to existing languages.

We identify five key requirements for an effective programming environment for protocol monitoring. While some of these requirements stem directly from the challenges listed above, others are the needed to make the monitoring process effective and useful.

Expressiveness Given a protocol state machine, the programming language should make it easy to write a monitor that checks for conformance. Moreover, it should not be necessary to write a monitor for the complete protocol; if needed, it should be easy to write a monitor for only a small sub-protocol. Finally, it should be easy to write monitors for several layers of protocols and put them together.

Efficiency When analyzing a live network online, the executable monitor should be able to concurrently check a number of devices that are running several layers of protocols and are participating in multiple sessions at the same time. When carrying out offline analyses, the executable monitor should be able to analyze large packet traces captured over several days.

Correctness Given the complexity of writing a monitor program, it should be possible to establish that the program faithfully encodes the formal protocol specification. In particular, the programming environment should be able to guarantee that the monitor raises an alarm only when there is a violation of the specification (no false positives).

Diagnostics Given that the executable monitor will be analyzing several thousand events, the programming environment should provide adequate diagnostics with alarms. In particular, given an alarm trace, it should be easy for the protocol analyst to know what property in the specification has been violated.

Monitoring with Infidelity The programming environment should provide support for writing monitors that analyze traces with infidelities. In particular, it should be easy to write co-networked monitors for simple properties.

In Chapter 3, we reiterate these requirements as part of the problem statement of this thesis, and in Chapter 4, we present our domain-specific language, NERL, which satisfies most of these requirements. In the next section, we survey previous attempts to write passive monitoring systems and we look at a few systems in detail. While none of the existing passive monitoring systems fully satisfies our needs, they serve to motivate our requirements and illustrate alternative approaches.

2.4 Related Work

The idea of applying passive run-time monitoring to networks is not a new one. The earliest proposed application that we can find in literature 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. If we restrict our attention to implemented passive monitoring frameworks, we can identify four lines of research in the literature. We overview each of these in order.

Passive testing systems are used to check protocol traces against specifications written using a state machine formalism, such as Estelle or LOTOS. Some of these systems have been used for online protocol monitoring. In Section 2.4.1, we sketch the design of a representative passive testing system, TETRA, and overview other approaches.

Run-time monitoring systems are used to check that programs written in some programming language are well-behaved at run-time. In Section 2.4.2, we examine a run-time monitoring language, MEDL, that has been used for protocol monitoring, and overview other systems.

Network Intrusion Detection systems (NIDS) are used to detect attacks on servers in protected networks. Although such systems can only look for simple attack patterns, they are usually well-engineered and can analyze a large amount of network traffic. In Section 2.4.3, we detail the Bro system, a sophisticated, programmable, specification-based NIDS.

The final category of passive monitoring research consists of case studies for specific protocols. While individual case studies are too many to list here, in Section 2.4.4, we survey studies for protocols similar to those analyzed in this dissertation.

2.4.1 Passive Testing

Active tests send inputs to the device under test (DUT) and analyze its outputs. In contrast, passive tests assume that some other device is sending inputs to the DUT and analyze both the inputs and the outputs. This means that passive testing techniques need to do more work; whereas an active tester can pre-compute the expected outputs for every test input sequence, the passive tester does not know what inputs to expect and so must compute the expected outputs on the fly. As a result, programming a passive tester is not easy. One approach is to automatically generate a passive tester from the protocol specification. This is the approach followed by the TETRA tool and its successors.

TETRA

Formal description techniques, such as LOTOS (Lot87), Estelle (Est89), and SDL (CCI76), have been used by protocol designers to write high-level specifications for protocols at several layers in the OSI protocol stack. These languages are based on a finite state machine model and allow non-determinism, synchronous and asynchronous communication, and modular design through layering.

In a long line of work (BDZ89, BB89, BDD+90, EB95, PYB96), Bochmann *et al.* take the approach that the best way of writing passive testers for a protocol is to generate them di-

rectly from the protocol specification written in one of these languages. Since the specification is typically non-deterministic, a important aspect of their research is the generation of a deterministic monitor.

The Test and Trace Analyzer (TETRA) (BB89) includes an offline trace analysis algorithm that is representative of this approach. It first takes an observed packet trace and translates it to a LOTOS process. The resulting *trace process* consists of a long sequence of message outputs, one for every packet in the original trace. TETRA then composes the trace process with a LOTOS specification of the protocol and asks whether the two are consistent. This query is implemented as a depth-first search for a bad state in the the state space of the composed system.

The passive testing approach taken by TETRA has several advantages. First, there is no new monitor programming language to learn. Instead, the protocol analyst can reuse existing specifications written in expressive languages. Second, the correctness question for the monitor does not arise since it is generated from the specification. Third, the analyst can use several analysis and diagnostic tools that have been written for LOTOS specifications. Fourth, while none of these tools has algorithms to handle trace infidelity, it seems straightforward to model the non-deterministic buffer as an additional process.

On the other hand, this translation approach has proved to be inefficient and unscalable. Translating large packet traces to processes is infeasible. In real networks, it is not unreasonable to analyze a packet trace with millions of packets. Representing such large traces in LOTOS is beyond the ability of TETRA. An alternate approach is to analyze the trace as it is being generated rather than translating it to a process. Ezust and Bochmann (EB95) explored this online approach in the Tango tool. Tango 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. This is due to a second limitation of the TETRA approach, namely

that depth-first search of the state space of non-deterministic protocol specifications is very inefficient.

Protocol specifications written in LOTOS or Estelle are typically non-deterministic to allow for several protocol implementations. To reuse these specifications for monitoring, tools such as TETRA determinize them on the fly, using state-space search for instance. As a result, the number of states and transitions checked by the monitor can be exponential in the size of the packet trace or exponential in the number of states in the specification, whichever is smaller. This is wasteful, especially when we are only interested in monitoring a small aspect of the protocol.

We believe that significantly better monitoring performance can be achieved by allowing the programmer to manually rewrite the specification as a deterministic monitor. This way, the programmer can optimize the determinization and can specialize the program to check only the interesting parts of the protocol. Of course, it will then be necessary to provide some tool, such as a model-checker, to check that this programmed monitor is correct with respect to the formal specification. To see the difference in performance between the two approaches, consider the following example. 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 of thousands of packets every second (see Chapter 6). We surmise that the memory cost of representing state sets probably accounts for this disparity.

Other Work

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 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. While the treatment of monitors and traces is quite formal, the EFSM specifications and tools used in these studies offer little support for protocol layering and error diagnostics.

A separate 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.

2.4.2 Run-time Monitoring

It has long been recognized that software testing does not guarantee correctness. While the guarantees provided 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 is a tool that checks whether an implementation has reached a dangerous state and alerts an administrator or calls corrective programs. Run-time monitors have been used successfully to find errors in sequential and concurrent software, but rarely in network protocols. In the following section, we examine a system that has been used to analyze and find errors in a wireless routing protocol.

MEDL

Lee *et al.* (LKK+99, KVBA+98, KVBA+99) introduced a general architecture, MaC, for the Monitoring and Checking of software executions. One instantiation of MaC is the Java-MaC system for monitoring executions of Java programs.

The Java-MaC architecture consists of two components: a source program instrumentation technique and a passive monitoring system. The input to Java-MaC consists of the source Java program, a monitoring script written in the Primitive Event Definition Language (PEDL), and a set of safety properties written in the Meta Event Definition Language (MEDL).

The PEDL script defines boolean conditions on the variables of the source program. This script is used to generate code that is inserted at key points in the source program. Whenever the monitored conditions change, the *instrumented* source program generates *primitive events* for analysis by the MEDL program. PEDL scripts represent white-box monitors that rely on being able to see the internals of the source Java program.

The MEDL program analyzes a sequence of primitive events and raises an alarm if a safety property is violated. This program is translated into a Java program that runs in parallel with the instrumented source program and monitors it for primitive events. MEDL programs represent black-box passive monitors that can be used on captured primitive event traces as well as on local or remote program executions.

A second instantiation of the MaC framework is the Verisim system (BGK+02) for monitoring protocol implementations in a simulated network. It uses the Network Simulator (NS),

instead of PEDL, to generate primitive event traces and uses MEDL to analyze these traces for errors. To use MEDL for analyzing protocol traces, the protocol analyst must first encode the protocol specification into safety properties expressed in terms of events and conditions. The core syntax of MEDL is as follows. A program consists of auxiliary variables, event definitions, and condition definitions. The variables represent the state of the monitor and can contain booleans, integers, or real numbers. A condition represents either a boolean condition on the state variables or the period between two events. An event can occur either when a primitive input event is received, or at the beginning or end of a condition, or when some event occurs during a condition. An event can have an integer attribute.

To see an example, consider the following simple MEDL program, partly taken from (BGK+02). We assume that there are two nodes 0 and 1 sending messages to each other and that each message has a sequence number. This program checks that the sequence numbers sent by a node are always greater than zero and that they are non-decreasing over time.

```
ReqSpec M

/* Primitive Input Events representing a Packet reception */

   /* 1. packet reception */
   import event pkt_rcv;

   /* 2. packet fields */
   import event src, dest, seqno;

/* Auxiliary State Variables */

   int last_seqno_0;
   int last_seqno_1;

/* Alarms */

   alarm Negative_Seqno =  pkt_rcv when
                             (value(seqno,0) < 0);


   alarm Decreasing_Seqno_0 =  pkt_rcv when
                               ((value(src,0) == 0) &&
                                (value(seqno,0) < last_seqno_0));

   alarm Decreasing_Seqno_1 =  pkt_rcv when
                               ((value(src,0) == 1) &&
                                (value(seqno,0) < last_seqno_1));
```

```
/* State Transitions */

   pkt_rcv when
     ((value(seqno,0) > last_seqno_0) &&
      (value(src,0) == 0))            -> {last_seqno_0 = value(seqno,0)};

   pkt_rcv when
     ((value(seqno,0) > last_seqno_1) &&
      (value(src,0) == 1))            -> {last_seqno_1 = value(seqno,0)};
```

In this program, the reception of a packet is represented by the input event pkt_rcv. The fields of the packet are represented by two other input events because MEDL only allows one attribute per event and we need the packet event to have three attributes. The attribute of a MEDL event E can be extracted by the expression value(E,0) and is assumed to have an integer type. The last sequence number sent by node 0 is stored in the auxiliary variable `last_seqno_0`, and the last sequence number sent by node 1 is stored in `last_seqno_1`. The state transitions at the end of the program implement this behavior. The alarms in the middle of the program check the desired safety properties. Negative_Seqno generates an alarm event if the input packet has a negative sequence number, and Decreasing_Seqno_0 (Decreasing_Seqno_1) raises an alarm when the sequence number sent by node 0 (node 1) is less than the previous value.

Monitors written in MEDL tend to be simple and easy to understand. They are translated to Java programs that can be checked for type safety. We have successfully used MEDL to analyze simulations of a wireless protocol running on 50 nodes. However, in doing such large case studies, some deficiencies of MEDL become apparent. First, as depicted in the program above, since MEDL does not support rich event attributes, every packet in MEDL must be modeled by several input events, which is wasteful and inefficient. Second, since MEDL does not directly support arrays, we must maintain copies of variables and events for every protocol participant (such as last_seqno_0, last_seqno_1). Third, there is no easy way to add diagnostic information to the alarms. In our case studies, we had to manually analyze the input

event traces to understand each alarm. Fourth, since MEDL is designed to monitor software over reliable synchronous links, it does not support monitoring a device over a buffered, lossy channel.

We believe that MEDL provides an excellent foundation for passive monitoring but it lacks some domain-specific features that are essential for monitoring networks. In Chapter 4, we present a domain-specific language, NERL, which is a direct descendant of MEDL. NERL extends MEDL with richer data types, such as records and arrays, richer event attributes, and a mechanism for modular design and layering of protocol monitors. In addition, we implement a new compiler that translates NERL programs to C, incorporates type checking, and automatically generates code for event diagnostics and monitoring with infidelity.

Other Work

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 work by Bochmann *et al.* described previously.

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.

For monitoring general-purpose software, the main challenge is to be able to observe events internal to the executing process. As a result, run-time verification research has mainly focused on techniques that instrument programs to produce external events whenever the internal state changes.

For monitoring network protocols, a lot of this research is unnecessary because the protocol standard already describes a set of observable actions that can be monitored.

2.4.3 Network Intrusion Detection

Network Intrusion Detection Systems (NIDSs) are used to monitor network traffic to look for 'bad users' in the network. Although NIDSs have not been used for correctness testing, many 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 NIDSs (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 NIDSs, 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 NIDSs have an approach that is closest to passive protocol monitoring. They check traces generated by protocol implementations against a specification. In the next section, we examine the Bro NIDS, which has a rich specification language for representing attacks and security policies.

Bro

Architecturally, the Bro intrusion detection system is divided into two components: an event engine and a policy script interpreter (Pax99). The event engine sniffs packets from a network interface, reconstructs high-level protocol messages, and generates events for the policy interpreter. The policy interpreter checks these events against a site-specific security policy. The event engine contains protocol modules for several well-known protocols. These modules are written in C++ and encode a protocol state machine to recognize high-level events. For instance, the TCP module takes a packet stream and executes the TCP state machine to produce events such as `connection_established`. The security policy scripts are written in a domain-specific language, also called Bro.

Bro programs consist of handlers for events generated by the event engine. The contents

of these events can be stored in tables or arrays and correlated with subsequent events to identify an attack. For instance, the following simple Bro program checks the same property as the MEDL program in the previous section:

```
/* State Declations */

global log =  open("log.txt");
global last_seqno: table[addr] of int;

/* Event Handlers */

event pkt_rcv(src:addr, dest:addr, seqno: int){
  if (seqno < 0) {
    print log, fmt("Negative sequence number %d sent by %s",seqno,src);
  }

  if (seqno < last_seqno[src]) {
    print log, fmt("Decreasing sequence number %d sent by %s",seqno,src);
  }

  if (seqno > last_seqno[src]) {
    last_seqno[src] = seqno

  }
}
```

Bro uses C-like syntax for statements and expressions. In this program, a variable is first declared for the log file. Then, we declare a table that stores the last sequence number seen for each source address. Bro provides a special data type addr for IP addresses. The packets captured by the monitor are represented by the pkt_rcv event. We assume that there is a protocol module in the event engine that captures packets, parses them, and generates pkt_rcv events with the packet fields represented as arguments to the event handler. The body of the event handler prints out error statements whenever a packet sent by a node has a sequence number that is negative or is less than the last sequence number stored for the node. If the received sequence number is greater than the last one, it is stored in the table.

Writing policy scripts in Bro is made simple by the domain-specific datatypes and library functions such as fmt. Moreover, Bro is strongly-typed and simple mistakes in coding are easily detected. However, Bro programmers are expected to encode all protocol-specific analyses at the event engine layer. There are two reasons for this design choice. First, Bro

is an interpreted language and programs are not optimized very aggressively, whereas protocol modules in the event engine are written in C++ and compiled to machine code. For high-volume monitoring, the performance hit of interpreting Bro is too high to consider doing protocol analyses at that layer. Second, the design philosophy of Bro is to separate the low-level protocol analysis that is independent of the site being protected from the high-level policy analysis that is specific to the site. It is expected that new protocol modules will be added rarely and only by experts, who can be trusted to write correct C++ programs. This architecture is adequate for intrusion detection, where only a few low-level protocols need to be analyzed. However, it is not suitable for a passive protocol testing system where a major motivation is the testing of new protocols. To have any confidence in the analyses, the protocol modules would have to be checked for correctness. To add diagnostic support and implement algorithms for co-networked monitoring, we would need to perform source-level transformations on the module code. Analyses and transformations on C++ programs are known to be difficult to implement because of the flexibility of the programming language. Our approach is to write the protocol modules in a Bro-like domain-specific language so that we can design and implement analyses and transformations for this restricted language.

Limitations of NIDS

A recent body of work has discovered several limitations of the ability of NIDS to discover attacks. Some of these limitations are applicable to passive protocol monitoring in general. Ptacek and Newsham (PN98) demonstrate that an intruder can often fool the NIDS either because of errors in the protocol monitor or because of ambiguities in the protocol. Their discovery of monitor errors reinforces our requirement that the protocol monitor should be proved correct for its results to have any significance. Specification ambiguities are more difficult to combat. 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 packets and normalizes them to resolve ambiguities. A similar approach has been advocated as *protocol scrubbing* (MWJH00). Since these are active testing techniques, we do not adopt them in our work. Instead, we rely on the programmer to write monitors that understand specification ambiguities. To aid in this task, we provide tools to check that the resulting monitor is correct.

A separate limitation is the inability of NIDS to monitor a network in the presence of packet loss and re-ordering. Paxson (Pax97, Pax99) shows that in the presence of network errors, NIDS cannot distinguish between errors and attacks, and therefore produce false positives. No solutions have been presented for this problem; the best a NIDS can do is to detect network errors and stop monitoring the session. We present a uniform partial solution to this problem by implementing the trace-search algorithm for monitoring with infidelity.

2.4.4 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), which 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 device and attempts to find

errors in the TCP 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).

Routing. Testing routing protocols is harder than testing two-party protocols 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 without the effort of maintaining distributed monitors. Simulators, such as the Network Simulator (NS), take a protocol implementation and execute it over a virtual network configuration producing an event trace (Prob). 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 fine state machine (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. In our work, we check several SMTP implementations for conformance with the standard. We also check that our SMTP monitor is correct and that it does not produce false positives.

2.5 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 indicates that protocol software can fail because of three reasons: the implementation may violate the specification, the specification may fail to satisfy the requirements, or the assumptions made on users and the network may prove to be incorrect.

In (BOG02), we showed that protocol verification techniques can be used to prove the correctness of protocol specifications with respect to their requirements. We demonstrated several errors in a routing protocol, suggested some corrections, and provided formal proofs of correctness for fixed versions of the protocol. However, current verification technology is inadequate for proving the correctness of protocol implementations.

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. We proposed several novel testing techniques to improve the scalability of our monitors and to reduce false alarms.

In (BGO00), we introduced a novel analysis technique called Fault Origin Adjudication.

We showed that when both conformance and functionality 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).

In (BCMG01), we identified and formalized the problem of trace infidelity. We presented a general algorithm, called the Trace Search algorithm, for co-networked monitoring. We also presented more efficient algorithms for the co-networked monitoring of specific classes of protocol properties.

In (BG02b), we detailed the requirements for a passive protocol monitoring language. We argued that existing programming languages are inadequate for our purposes and advocated the design of a new domain-specific language. This work served to motivate the design of NERL.

CHAPTER 3

NETWORK EVENT RECOGNITION: A PROBLEM STATEMENT

In the previous chapter, we identified five key requirements for programming protocol monitors. In this dissertation, we design and implement a protocol monitoring framework, called Network Event Recognition, that satisfies these requirements. The Network Event Recognition framework consists of a domain specific language, NERL, and several analysis tools and techniques for passive protocol monitoring. In this chapter, we seek to precisely define the problem that Network Event Recognition seeks to address. We define our goals, our methodology, and our evaluation strategy.

3.1 Goal

The goal of this dissertation is to develop a flexible, programmable, passive protocol monitoring system and apply it to the analysis of a variety of network protocol implementations in diverse monitoring environments.

3.1.1 Protocols

Our objective is to be able to analyze a wide variety of protocols. We aim to write monitors for analyzing protocols at several layers—network, transport, and application—in

the Internet protocol stack. We aim to analyze both multi-party protocols and two-party protocols. However, we do not aim to analyze protocols that exchange encrypted messages.

3.1.2 Properties

Our objective is to monitor implementations for a wide variety of safety properties. For each protocol, we aim to test implementations for both conformance and functionality. In particular, we aim to check implementation traces for deviations from the protocol state machine and for deviations from the logical requirements. However, we do not aim to monitor liveness properties or protocol properties that are unobservable in a packet trace.

3.1.3 Monitoring Environments

Our objective is to design a passive monitoring system that is flexible enough to be applicable in several monitoring environments. It should be able to analyze offline packet traces captured from the network. It should be able to analyze event traces generated by network simulations. It should be able to capture packets from a live network and analyze them online. We aim to write programs for co-located monitoring, for bottleneck monitoring over reliable streams, and for co-networked monitoring. We do not aim to address any other monitoring environment.

3.2 Methodology

We design and implement a programmable passive protocol monitoring framework called Network Event Recognition. The framework consists of a domain-specific language, NERL, for programming protocol monitors and several analysis tools and techniques for programs written in this language.

NERL programs are meant to represent monitorable versions of formal protocol spec-

ifications. NERL provides domain-specific constructs that make it easy to express protocol state machines, protocol message formats, and protocol layering. The NERL compiler automatically generates code for common protocol monitoring tasks that are otherwise repetitive and error-prone. The compiler includes a *model checking* feature to check the correctness of a monitor program with respect to the protocol specification. NERL monitors can be compiled with an *event tracing* feature that provides important diagnostic information in the form of the history of events that caused an error. NERL monitors can be also compiled with *channel transformations* that enable them to distinguish between network errors, such as packet loss and delay, and protocol errors, such as deviations from the specification.

To help find errors from protocol traces, we introduce two novel techniques: *tuning* and *fault origin adjudication* (FOA). Tuning allows the protocol analyst to extract several errors from the same packet trace. FOA allows the analyst to use multiple tests to distinguish errors in implementation from errors in specification.

3.3 Evaluation Strategy

To evaluate the effectiveness of the NERL language and associated tools and techniques, we carry out case studies that demonstrate our methodology for a variety of protocols and monitoring environments. We have chosen the following three case studies:

Routing Protocol Simulations

- Protocol: AODV, a new, network layer, multi-party protocol
- Properties: Conformance with the standard and requirements
- Monitoring Environment: Network simulations

Internet Mail Forwarding

- Protocol: SMTP, an established, application layer, two-party protocol

- Properties: Conformance with the standard
- Monitoring Environment: Live test network

Reliable In-order Transport

- Protocol: TCP, an infrastructural, transport Layer, two-party protocol
- Properties: Conformance with the standard
- Monitoring Environment: Offline packet traces captured from a lossy, buffered link

To evaluate the results of the case studies, we use 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?

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?

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?

Monitoring with Infidelity Can we write programs that can monitor a protocol session even when packets are lost or delayed between the monitor and the device?

CHAPTER 4

PROGRAMMING IN NERL

This chapter describes a domain-specific language, NERL, for writing protocol event recognizers. The language is introduced through a series of example NERL programs. For the detailed syntax and usage rules for all the NERL programming constructs, see Appendix A.

First, we fix some terminology and overview the structure of a NERL program. Then, we introduce a simple protocol, Ping, and write a NERL program to monitor Ping sessions for errors. We then extend this monitor to perform tasks that are slightly more complicated. Finally, we show how the NERL tools can be used to simplify programming and ease protocol analysis.

4.1 Preliminaries

A NERL program consists of a set of *recognizer modules* strung together by a *main module*. It represents a monitoring configuration such as the one shown in Figure 4.1. In the figure, there are 4 active instances: 2 instances of the recognizer module P and 1 each of Q and R. The packet stream is multiplexed into the input events of P and Q. The output event of Q is sent to R, and the output events of P and R are reported to the user as alarms.

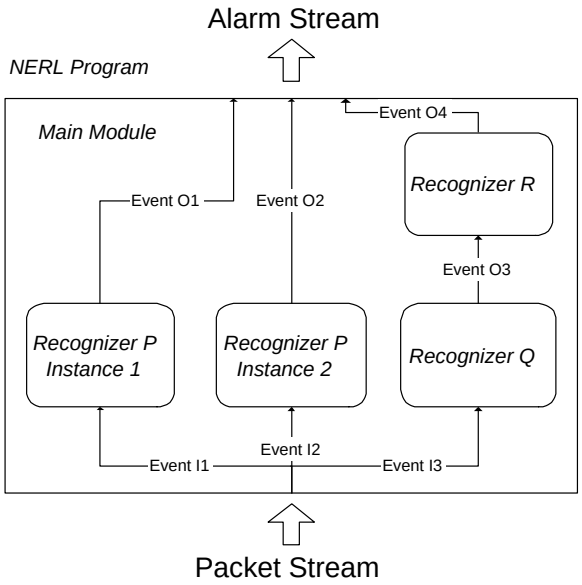

Figure 4.1 A NERL program in execution

A recognizer module represents the monitor for a particular protocol. Each recognizer has local variables; an assignment of values to these variables is called a *state*. The recognizer responds to *input events* and produces *output events*. Events have *attributes* that contain additional information about them. Figure 4.2 depicts a single event recognizer in execution: It is useful to think of a recognizer as monitoring a single protocol participant engaged in a single session. The recognizer gets input events that correspond to messages sent and received by this participant during this session. At the lowest layer, these input events are packets that have been captured from the network; at higher layers, these are messages that have been reconstructed by some lower-layer recognizer. In a single round, a recognizer takes a single input event, executes a state transition, and produces zero or more output events. These output events may represent errors or meta-events for some higher-layer protocol. In this way, each recognizer can be seen as a transducer that takes a low-level event stream and transforms it to a high-level event stream. Error events in the high-level stream are called *alarms*.

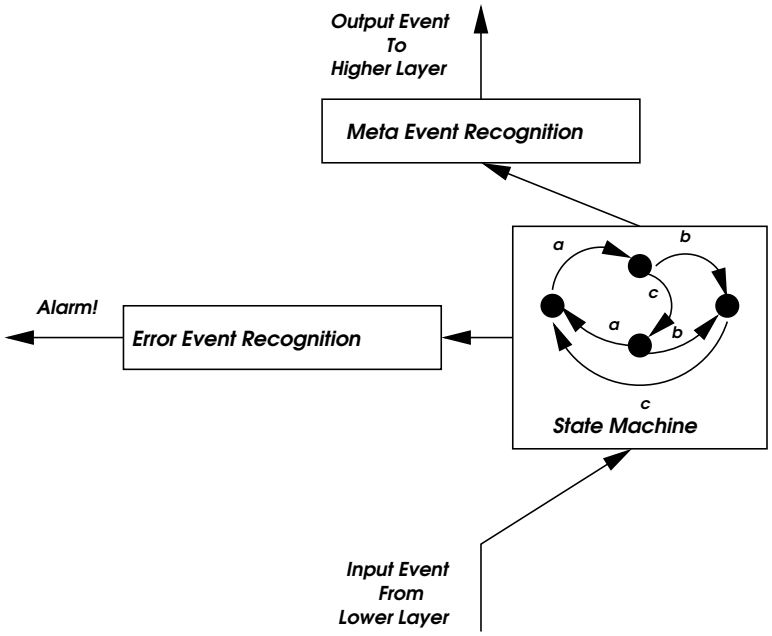

Figure 4.2 Anatomy of a Recognizer

At any time, there can be several active *instances* of a recognizer, one for each protocol session being monitored. An instance is identified by the recognizer module name and an *index*. Each instance maintains its own copy of the recognizer state. No state is shared between different instances of the same recognizer.

The main module manages the active recognizer instances and the flow of events between them. When a new protocol session is detected, it creates and initializes a new recognizer instance to monitor this session. Thereafter, it diverts the packet events corresponding to this session to the new instance, and captures the output events produced by the instance for further processing. Finally, when the session terminates, it deletes the instance.

The NERL compiler translates a NERL program to an executable *monitor*. In addition, there are three programmer tools implemented as variations of the compiler. When a program is annotated with loss and delay information for the monitored channel, the compiler applies *channel transformations* to account for the corresponding trace irregularities. When the program is compiled with the *model-checking* option, it produces Promela code that can

be analyzed for errors by the SPIN model-checker. When the program is compiled with the *event tracing* option, the resulting monitor executes a dynamic slicing algorithm that enhances each output event with the trace of causally preceding input events. We shall see examples of all these variations in the following sections.

4.2 The Ping Protocol

The Ping protocol is inspired by the ICMP Echo feature (Pos81). When a user asks node A whether node B is alive, A sends a request message to B. If B is alive, it sends back (echoes) the same message as a reply to A. When this reply reaches A, A informs the user that B is alive. The message exchange is shown in Figure 4.3. A is called the Ping sender and B is called the Ping receiver. The EchoRequest message contains two attributes: a sequence number s, and

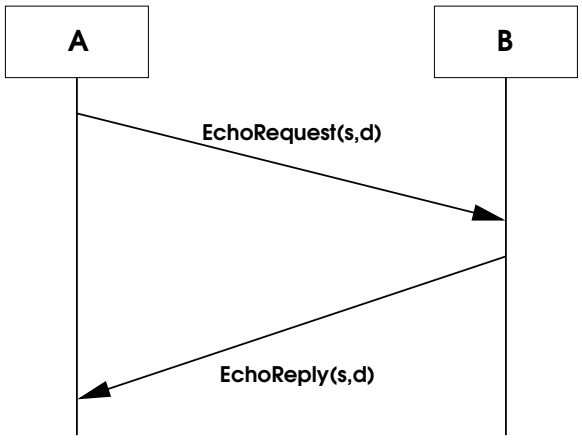

Figure 4.3 Ping Message Exchange

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

The state machine in Figure 4.4 represents the process at node B.

It receives an EchoRequest and replies with an EchoReply that contains the same sequence number and data. In the figure, input events are suffixed by question marks while output

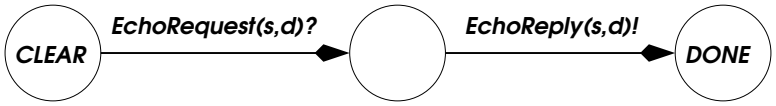

Figure 4.4 Ping Receiver Specification

events end in exclamation marks. Transient states—when the process has received an input event and is on the verge of producing an output—are unnamed.

On the other hand, the process at node A sends a request, waits for the reply, and if they match, generates an IsAlive event for the higher-layer. The corresponding state machine is shown in Figure 4.5.

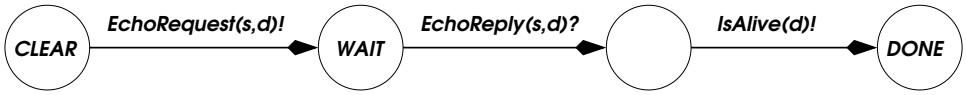

Figure 4.5 Ping Sender Specification

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

1. Check that the receiver follows the state machine in Figure 4.4. In particular, does the receiver ever produce an EchoReply(s,d) without a sender having sent it an EchoRequest(s,d)?

2. Reconstruct the IsAlive higher-layer event. Whenever the Ping sender A produces an IsAlive(d) event for the user, the monitor should produce an IsAlive(d) event as well. This monitor event may be of interest to some higher-layer recognizer.

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

We claim that the state machine shown in Figure 4.6 performs exactly these two tasks.

Notice that this state machine is essentially an extension of the Ping sender (A) state machine, except that both the lower-layer messages (EchoRequest,EchoReply) are considered inputs to the monitor, and an additional error checking transition is added. The monitor looks at a

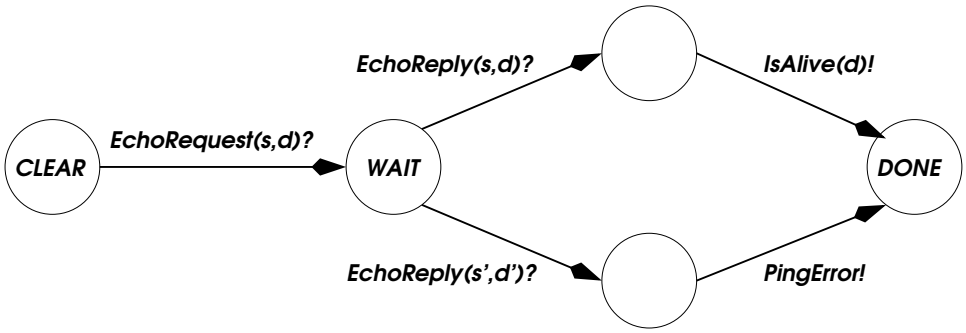

Figure 4.6 Ping Monitor Specification

message exchange on the wire, such as the one in Figure 4.3. It checks that the reply contains the same sequence number and data as the request, and if so produces an IsAlive meta-event indicating that node B successfully replied. If the reply does not match the request, a PingError error-event is generated. As before, transient states are unnamed.

For now, the relationship between the three state machines described above is left informal. We will see later in this chapter that we can use model checking to formally demonstrate that the monitor specification (Figure 4.6) correctly analyzes message sequences between Ping senders (Figure 4.5) and receivers (Figure 4.4).

4.3 The Recognizer Module: ping.nerl

We can now write a NERL program implementing the Ping monitor state machine from Figure 4.6. First, we define a variable status to store the state of the monitor; status 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:

```
#define CLEAR 0
#define WAIT 1
#define DONE 2
int status;
int seq;
string data;
```

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

Next, we declare the inputs and outputs of the recognizer. The Ping recognizer has two primary inputs:

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

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

```
typedef {
  int seq;
  string data;
} ty_echo;
```

Then, we declare the two primary outputs:

```
output event string IsAlive;
output event string PingError;
```

The string attribute of the IsAlive event contains the data exchanged, and the string attribute of PingError contains an error message.

In addition to these inputs and outputs, every NERL recognizer has a distinguished input event called **Init** that is used to initialize the state, and a distinguished output event called **Done** that indicates that the session being monitored is over. By default, the **Init** input event does nothing, and the **Done** event is never triggered. For now, we shall ignore these events.

Next, we program each state transition shown in Figure 4.6. When an EchoRequest is seen in state CLEAR, we update the seq and data variables and the state changes to WAIT:

```
transition EchoRequest(e)
           OccurredWhen (status == CLEAR) -> {
                                              status = WAIT;
                                              seq = e.seq;
                                              data = e.data;
                                             }
```

The **transition** keyword introduces the state transition; the expression to the left of –> is the guard—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 –>) contains a statement that transforms the state by assigning values to variables. In general, the statement can contain **if** conditionals, and **while** loops, which use 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, PingError:

```
event EchoReply(e)
      OccurredWhen ((status != WAIT) &&
                    ((e.seq != seq) ||
                     (e.data != data))) -> PingError(e)
                                           WithAttributes
                                           {e= "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 –> 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.seq,e.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 –> is true, then the event to the right is generated. In this case, the

PingError (error) event is generated. The **WithAttributes** construct attaches a string attribute e to this event indicating the cause of the error.

Finally, if the reply is correct then we generate the IsAlive meta-event:

```
event EchoReply(e)
      OccurredWhen ((status == WAIT) &&
                    (e.seq == seq) &&
                    (e.data == data))  -> IsAlive(a)
                                          WithAttributes
                                          {a = e.data};
```

In this case, the **WithAttributes** construct is used to copy the data content, which may be of interest to a higher-layer recognizer. For instance, the data often contains a timestamp that can be used to compute the round trip time between Ping sender and receiver.

When the Ping session is over, we tell the NERL runtime to stop the recognizer and delete its state. This is done by triggering the **Done** output event as follows:

```
event (IsAlive | PingError)  -> Done;
```

We use the event disjunction operator (|) to check whether either the `IsAlive` event or the `PingError` output event has been generated. In both cases, the work of the recognizer is over and the `DONE` event is generated.

NERL also provides an event conjunction operator (&) that is used to check whether multiple events have occurred. However, event disjunction and conjunction behave quite differently with respect to attributes; while conjuncts are allowed to bind attribute variables, disjuncts are not. For instance, the following event conjunction is legal in NERL:

```
transition (IsAlive(a) & PingError(e))  -> {x = a; y = e;}
```

Here, the variable `a` is bound to the attribute of `IsAlive` and the variable `e` is bound to the attribute of `PingError`. On the other hand, the corresponding disjunction is illegal:

```
transition (IsAlive(a) | PingError(e))  -> {...}
```

In the above transition, suppose `IsAlive` has occurred but `PingError` has not; then `e` would be undefined in the right hand side. To avoid this situation, NERL prohibits the

use of attribute variables in disjunctive event patterns. This is not a serious limitation, since transitions like the above can always be split into two separate transitions.

This completes the specification of the NERL recognizer for the simple Ping monitor shown in Figure 4.6. The complete event recognizer for Ping includes an **Init** input event that initializes the status variable, a **Done** output event indicating the end of the Ping session, and an additional error check for EchoRequest events. The complete NERL specification of the Ping monitor is shown in Figure 4.7. The recognizer is enclosed by the **Recognizer** and **EndRecognizer** keywords and preceded by the type definitions.

4.4 Main Module: pingmod.nerl

In the last section, we wrote an event recognizer for the Ping protocol. However, in itself this recognizer module cannot be executed because it does not specify where the input events come from or where the output events are sent. As such, the Ping recognizer is like a C function; we still need to write a 'main' function that closes it and makes it executable. In this section, we shall write such a main module in NERL.

A main module has two functions: it declares and instantiates all the recognizer modules that the monitor will use and it manages the inputs and outputs between these modules. While reading this section, it will be useful to remember that while the syntax of the main module has been designed to put NERL modules together, it will work just as well if the individual recognizers are written in C, Lex, or Perl, as long as they obey the input-output event interface.

To build an executable monitor for the Ping protocol, the Ping event recognizer must be put together with a packet capture module. The packet capture module 'sniffs' a stream of packets from a capture source, such as an Ethernet adapter or packet log, and when it sees an ICMP packet, it extracts the ICMP fields from the packet and produces an output event ICMP.

```
Recognizer Ping =
  typedef {
    int seq;
    string data;
  } ty_echo;

  input event bool Init;
  input event ty_echo EchoRequest;
  input event ty_echo EchoReply;

  output event string IsAlive;
  output event string PingError;
  output event bool Done;

  #define CLEAR 0
  #define WAIT 1
  #define DONE 2
  int status;
  int seq;
  string data;

  transition Init(i) -> {status = CLEAR};

  event EchoRequest(e)
        OccurredWhen (status != CLEAR)  -> PingError(e)
                                            WithAttributes
                                            {e = "Multiple Echo Requests"};
  event EchoReply(e)
        OccurredWhen (status != WAIT)   -> PingError(e)
                                            WithAttributes
                                            {e = "Unexpected Echo Reply"};
  transition EchoRequest(e)
             OccurredWhen (status == CLEAR) -> {seq = e.seq;
                                                data = e.data;
                                                status = WAIT};
  event EchoReply(e)
        OccurredWhen ((status == WAIT) &&
                      (e.seq == seq) &&
                      (e.data == data)) -> IsAlive(a)
                                            WithAttributes
                                            {a = e.data};
  event EchoReply(e)
        OccurredWhen ((status == WAIT) &&
                      ((e.seq != seq) ||
                       (e.data != data))) -> PingError(e)
                                              WithAttributes
                                              {e = "Incorrect Echo Reply"};
  transition (IsAlive | PingError) -> {status = DONE};

  event (IsAlive | PingError)  -> Done;
EndRecognizer;
```

Figure 4.7 A Complete Ping Recognizer

The main module maintains one Ping instance for each session. When it sees the ICMP event, it detects whether the event contains an Echo Request or Reply, it finds out which Ping session the packet belongs to, and sends the event to the appropriate Ping instance. This hierarchy is depicted in Figure 4.8.

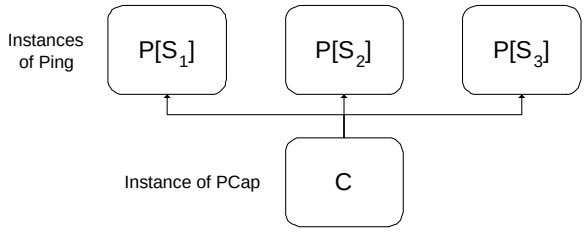

Figure 4.8 Ping Monitoring Modules

Let us write the NERL main module corresponding to Figure 4.8. First, we declare the input-output *signature* of each box. The *signature* of an event recognizer consists of all its input and output event declarations. For instance, the Ping recognizer has the following signature:

```
recognizer Ping : {
  input  event ty_ping Init;
  input  event ty_echo EchoRequest;
  input  event ty_echo EchoReply;
  output event string IsAlive;
  output event string PingError;
  output event bool Done;
}
```

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

```
recognizer PCap = {
  input event string Init;
  output event ty_icmp_pkt ICMP;
}
```

Since this is at the bottom of the monitoring stack, there are no real input events 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 a string containing the name of the file.

Note that although the packet capture module is written in C, this signature provides enough information for the main module to create and instantiate it, and control its interaction with the Ping recognizer.

Next, we declare and initialize recognizer instances. The monitor stack for Ping contains exactly one instance of PCap:

```
instance PCap        C(init)
                       WithAttributes
                       {init = "icmp.pcap"};
```

We declare that C is an instance of the recognizer PCap, and has the initialization parameter init, standing for the attribute of its **Init** event. By setting this parameter, C is instructed to read packets from the file "icmp.pcap".

We declare one instance of Ping for each session:

```
instance Ping        P[ty_ping_session s](init)
                       WithAttributes
                       {init = true};
```

P is an array of instances of the Ping recognizer, indexed by the session identifier s. For each new connection s, the main module will instantiate a new Ping recognizer P[s] and initialize it. The session identifier for Ping consists of the source and destination IP addresses, and the icmp_id field that distinguishes different packets between the same hosts:

```
typedef {
  int ip_src;
  int ip_dst;
  int icmp_id;
} ty_ping_session;
```

Finally, we define the flow of events between the packet capture module and the Ping instances. The arrows in Figure 4.8 are programmed as event forwarding definitions. These are similar to the NERL event definitions described earlier in this chapter, but are less powerful because main modules do not have any state variables. For instance, consider the following event forwarding definition for EchoRequest events:

```
event C.ICMP(p) OccurredWhen
       (e.icmp_type == 8)    -> P[x].EchoRequest(e)
                                  WithIndex {
                                    x.ip_src = p.ip_src;
                                    x.ip_dst = p.ip_dst;
                                    x.icmp_id = p.icmp_id; }
                                  WithAttributes {
                                    e.seq = p.icmp_seq;
                                    e.data = p.icmp_data; }
```

This definition is executed whenever the output event ICMP, with attributes p, is generated by the instance C, and the type field of the ICMP packet contains 8—the value corresponding to Echo Requests. When the guard to the left of –> is true, the event to the right is generated. In this case, the input event EchoRequest in the Ping instance P[x] is triggered. The **WithIndex** construct tells us which Ping instance this packet should be forwarded to. We specify that an ICMP event with attributes p should be forwarded to the instance monitoring the connection defined by <p.ip_src,p.ip_dst,p.icmp_id>. Finally, the **WithAttributes** construct behaves the same as in recognizer modules—it assigns attributes to the input event.

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

```
event E[x].IsAlive(a) -> PRINT
event E[x].PingError(e) -> PRINT
```

The **PRINT** module prints out the event to the user's terminal (stdout) using a uniform event format. We shall see several examples of the output produced by this module in the next section.

The complete NERL program for Ping monitoring is shown in Figure 4.9.

```
begin
 typedef { int ip_src;
           int ip_dst;
           int icmp_id;
           int icmp_type;
           int icmp_seq;
           string icmp_data; } ty_icmp_pkt;

 typedef { int ip_src;
           int ip_dst;
           int icmp_id; } ty_ping_session;

 recognizer PCap : { input event string Init;
                     output event ty_icmp_pkt ICMP; };

 recognizer Ping : { input event bool Init;
                     input event ty_echo EchoRequest;
                     input event ty_echo EchoReply;
                     output event ty_echo IsAlive;
                     output event string PingError;
                     output event bool Done; };

 instance PCap      C(init)
                     WithAttributes {init = "icmp.pcap"};

 instance Ping      P[ty_ping_session s](init)
                     WithAttributes {init = true};

 #define REQUEST  8
 #define REPLY    0

 event C.ICMP(p) OccurredWhen
        (e.icmp_type == REQUEST) -> E[x].EchoRequest(e)
                                    WithIndex {
                                      x.ip_src = p.ip_src;
                                      x.ip_dst = p.ip_dst;
                                      x.icmp_id = p.icmp_id }
                                    WithAttributes {
                                      e.seq = p.icmp_seq;
                                      e.data = p.icmp_data };
 event C.ICMP(p) OccurredWhen
        (e.icmp_type == REPLY)  -> E[x].EchoRequest(e)
                                    WithIndex {
                                      x.ip_src = p.ip_src;
                                      x.ip_dst = p.ip_dst;
                                      x.icmp_id = p.icmp_id }
                                    WithAttributes {
                                      e.seq = p.icmp_seq;
                                      e.data = p.icmp_data };
 event E[x].PingError(e) -> PRINT;
 event E[x].IsAlive(a) -> PRINT;
end
```

Figure 4.9 Ping Modules Description

4.5 Monitoring Ping Sessions

By putting together the C code for the PCap module, the NERL code for Ping, and the NERL code for the main module, we now have a complete NERL program to analyze Ping sessions. We can compile this program and execute it on the packet trace shown in tabular form below. This trace 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.

Pkt No.	IP src	IP dst	ICMP id	ICMP type	ICMP seq	ICMP data
1	158.130.012.217	158.130.012.004	13043	8 (REQ)	0	"saul-a"
2	158.130.012.004	158.130.012.217	13043	0 (REP)	0	"saul-a"
3	158.130.012.217	158.130.012.004	13044	8 (REQ)	1	"saul-b"
4	158.130.012.004	158.130.012.217	13044	0 (REP)	1	"saul-b"
5	158.130.012.217	158.130.012.004	13045	8 (REQ)	2	"saul-c"
6	158.130.012.004	158.130.012.217	13045	0 (REP)	2	"saul-c"

There are three pairs of events, each amounting to one ping. Each line is prefixed by a packet number indicating the order in which the event was seen by the monitor, then the input events for Ping are indicated with the relevant IP and ICMP fields. When this trace is fed to our NERL program, the following output trace is produced for the user:

```
2: IsAlive Event <``saul-a''>
2:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
4: IsAlive Event <``saul-b''>
4:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13044]
6: IsAlive Event <``saul-c''>
6:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13045]
```

The NERL program identified that there were three correct replies in the trace, resulting in three IsAlive events. The **PRINT** module outputs the packet number, the event name and attributes, and the name and session identifier of the recognizer instance that produced the output event.

To see the flow of events in more detail, we can use a verbose output mode to display the intermediate events between the PCap and Ping modules:

```
1: EchoRequest Event <seq:0, data:''saul-a''>
1:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
2: EchoReply Event   <seq:0, data:''saul-a''>
2:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
3: EchoRequest Event <seq:1, data:''saul-b''>
3:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13044]
4: EchoReply Event   <seq:1, data:''saul-b''>
4:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13044]
5: EchoRequest Event <seq:2, data:''saul-c''>
5:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13045]
6: EchoReply Event   <seq:2, data:''saul-c''>
6:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13045]
```

When the packet trace is fed to the PCap module, each packet is translated to an EchoRequest or EchoReply input event for a specific instance P[x]. For instance, the third packet produces an EchoRequest event with attributes {seq = 1; data = ''saul-b''} at the Ping instance P with session-id {ip_src = 158.130.012.217; ip_dst = 158.130.012.004; icmp_id = 13044}

Next, we introduce an error into the input packet trace. We modify packet 2 (the reply to packet 1) so that it has an incorrect sequence number, and we delete packets 3-6. The intermediate event trace now looks as follows:

```
1: EchoRequest Event <seq:0, data:''saul-a''>
1:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
2: EchoReply Event   <seq:2, data:''saul-a''>
2:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
```

Note that the seq attribute has been changed from 0 to 2. The output trace seen by the user has an error event:

```
2: PingError Event <''Incorrect Echo Reply''>
2:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
```

The monitor program recognizes the error and prints the appropriate error message.

4.6 Further Programming Examples

In the previous section, we wrote a NERL program to monitor the simple, two-message Ping protocol. The program exemplifies the standard pattern for programming monitors:

1. Declare the input and output events and local variables
2. Identify relevant input events using event patterns
3. To remember an event, trigger a state transition
4. To notify the user about an event, generate an output event
5. Attach attributes to events to give further information about them

For many protocols, the programming constructs we have seen so far will suffice. However, in some multi-message protocols with complicated packet formats we need to use arrays and while-loops, in state transitions as well as event definitions. In the following sections, we will see some examples of using these programming constructs in the context of monitoring.

4.6.1 Using Arrays

In the simple Ping protocol, the sender sends one request and waits for the reply. This means that if a request is lost, the sender must wait for the reply to time out before sending another request. To avoid this latency, a realistic Ping program sends several requests and waits for the first reply. Let us assume that the sender sends at most MAXSEQ requests with sequence numbers between 1 and MAXSEQ. Although the receiver can reply to any of the requests it receives, only the first reply will be considered.

To monitor this new protocol, we modify the Ping recognizer as follows: in the WAIT state, additional requests are allowed and all requests are put in an array. When a reply arrives, it must match one of the requests seen so far, otherwise, an error is raised.

The state is now modified to include an array of requests:

```
#define CLEAR 0
#define WAIT 1
#define DONE 2
int status;

#define MAXSEQ 5;

typedef {
  bool recd;
  string data;
} ty_req;

ty_req requests[MAXSEQ + 1];
```

Here, requests is an array of the last MAXSEQ requests indexed by sequence number. (The array size is MAXSEQ + 1 because NERL arrays are indexed from 0, not 1; so we have added an extra array position, to index it based on sequence numbers.) For each sequence number, we record a boolean, recd, that indicates whether a request with that sequence number has been seen, and we record a string, data, that contains the data seen in the last such request.

Now, when an EchoRequest is seen, it is inserted into the array:

```
transition EchoRequest(e)
           OccurredWhen
           ((status != DONE) &&
            (e.seq <= MAXSEQ)) -> {status = WAIT;
                                   requests[e.seq].recd = true;
                                   requests[e.seq].data = e.data};
```

When an EchoReply for one of the received requests is seen, the IsAlive event is produced.

```
event EchoReply(e)
        OccurredWhen
        ((status != DONE) &&
         (e.seq <= MAXSEQ) &&
         (requests[e.seq].recd == true) &&
         (e.data == requests[e.seq].data))  -> IsAlive(a)
                                               WithAttributes
                                               {a = e.data};
```

On the other hand, if the EchoReply is for a request that has not been seen then a PingError event is produced.

```
event EchoReply(e)
        OccurredWhen
        ((status != DONE) &&
         (e.seq <= MAXSEQ) &&
         ((requests[e.seq].recd == false ) ||
          (e.data != requests[e.seq].data)))  ->
                                              PingError(m)
                                              WithAttributes
                                              {m= "Incorrect Echo Reply"};
```

4.6.2 Using Variable-size Arrays

In the previous example, the receiver was only required to respond to one request. But, if we assume that no request gets lost in transit, but messages can get delayed, then the receiver should respond to all the requests that it receives, and the replies must be in the same order as the requests. Indeed, the Unix ping program uses this correlation between requests and replies to find the average round-trip-time between the sender and receiver. In this section, we shall write a NERL program to monitor this protocol. Further, we assume that there is no limit (MAXSEQ) or order to the sequence numbers that can appear in the requests.

For this protocol, we use a variable-size array to store the received requests.

```
#define CLEAR 0
#define WAIT 1
#define DONE 2
int status;

typedef {
  bool done;
  int seq;
  string data;
} ty_req;

ty_req requests[];
int next;
```

The requests array is a variable-size array (initially empty) that contains the sequence of requests seen in the session (in order). The index next (initially -1) indicates the next request to be processed.

Now, when an EchoRequest event is received, it is inserted at the end of the variable array.

```
transition EchoRequest(e)
           OccurredWhen
            (status != DONE)  -> {status = WAIT;
                                  push(requests,1);
                                  int last;
                                  last = requests#length - 1;
                                  requests[last].done = false;
                                  requests[last].seq = e.seq;
                                  requests[last].data = e.data;
                                  if (next == -1) then
                                     {next = last};};
```

We make space at the end of the array for another request using the **push** command. The local variable last refers to this new array position. We copy the parameters of the EchoRequest to requests[last]. If there are no pending requests, then next is -1, and it should be modified to reflect the new request.

Next, we modify the IsAlive event definition to check whether the EchoReply is the correct reply for the next request in the requests array:

```
event EchoReply(e)
        OccurredWhen
         ((status != DONE) &&
          (next >= 0) &&
          (requests[next].done == false) &&
          (requests[next].seq == e.seq) &&
          (requests[next].data == e.data)) -> IsAlive(a)
                                              WithAttributes
                                              {a = e.data};
```

Otherwise, we generate a PingError:

```
event EchoReply(e)
        OccurredWhen
         ((status != DONE) &&
          (next <= 0) ||
          (requests[next].done == true) ||
          (requests[next].seq != e.seq) &&
          (requests[next].data != e.data)) ->
                                          PingError(m)
                                          WithAttributes
                                          {m= "Incorrect Echo Reply"};
```

Finally, if the reply was correct, we move the next pointer to the next request, or if no requests are left set it to -1.

```
transition IsAlive(a) -> {requests[next].done = true;
                          if (requests#length - 1 > next) then {
                            next = next + 1;
                          } else {next = -1;}
                         }
```

4.6.3 Annotations for Channel Transformations

In the previous sections, we dealt in an ad-hoc way with the possibility of messages being lost or delayed as they traveled to their destination. As we saw, we needed some careful book-keeping to remember which messages have been seen and to correlate replies with requests. Loss and delay in the network are ubiquitous concerns in monitoring; they appear in most protocols that do not operate over reliable channels. NERL provides a feature called a *channel transformation* that automatically generates code for monitoring a protocol in spite of packet loss and delay.

To see an example, we consider a variation of the Ping protocol similar to the one in the previous section. The sender sends a sequence of requests and the receiver replies to the ones that it receives in the same order. If there is no delay or loss of packets, then this protocol has a simple request-reply-request-reply-... pattern. The NERL recognizer, PingRep, for this protocol is a small variation of the program in Figure 4.7; instead of terminating the recognizer after one request-reply pair, we reset the recognizer to accept more such pairs.

However, suppose that the Ping requests from the sender can be delayed or lost before they reach the receiver. Concretely, assume that up to 2 requests can be lost in a row, and up to 5 requests can be buffered before one must reach the receiver. This information can usually be gleaned from network performance statistics. For simplicity, we assume that replies are never lost or delayed. (For this protocol, the delay or loss of a reply can be treated as the delay or loss of the corresponding request.) Further, assume that the monitor is placed very close to the sender, so that it has perfect knowledge of all the messages that the sender sends or receives. In this scenario, there are two alternatives for the NERL programmer: (1) write a

new NERL program that allows for the delay and loss using techniques similar to the previous sections, or (2) annotate the PingRep recognizer with the buffering and loss information and let the inbuilt channel transformations generate the appropriate code. In the remainder of this section, we illustrate an example of the second alternative.

To trigger channel transformations, we annotate the signature of PingRep as follows:

```
recognizer PingRep : {
 channel  To[5,2];
 channel  From[0,0];

 input event bool Init;

 input event To   Ty_echo EchoRequest;
 input event From Ty_echo EchoReply;

 output event string IsAlive;
 output event string PingError;
 output event bool Done;
}
```

We declare that the Ping recognizer monitors two channels: a channel called To (from sender to receiver) that can buffer up to 5 packets and lose up to 2 packets in a row, and a channel called From (from receiver to sender) that has no delay or loss. Further, we specify that the EchoRequest message travels on the To channel and the EchoReply travels on the From channel.

Given these annotations, the NERL compiler automatically accounts for the channel characteristics. To see the result of the channel transformations, consider the following intermediate event trace:

```
1: EchoRequest Event <seq:0, data:``saul-a''>
1:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
2: EchoRequest Event <seq:1, data:``saul-b''>
2:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
3: EchoRequest Event <seq:2, data:``saul-c''>
3:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
4: EchoReply Event   <seq:1, data:``saul-b''>
4:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
```

All 4 events are for the same Ping session. The first 3 are requests, and packet 4 is a reply to request 2. Looking at the trace, we can surmise that request 1 was lost and request 2

was buffered before reaching the receiver, so the receiver did not violate the Ping specification. But when this trace is fed directly to PingRep without any channel transformation, it raises several PingErrors: at packets 2 and 3, since it does not expect to see two requests in a row, and at packet 4 since it has not seen a reply to request 1.

When we feed the trace to the monitor generated from the NERL prorgam with channel annotations, the output trace looks as follows:

```
2: Possible PingError Event  <''Too Many Echo Requests''>
3: Possible PingError Event  <''Too Many Echo Requests''>
4: Possible PingError Event  <''Incorrect Echo Reply''>
```

The channel transformations distinguish between *possible errors* and *definite errors*. A possible error means that there is some interpretation of the trace that leads to an error. For instance, if there was no buffering or loss then seeing two requests in a row is indeed an error. A definite error, on the other hand, is signalled only when all interpretations fail. Since there is one interpretation of the above trace that succeeds, namely that request 1 was lost and request 2 was buffered, so no definite error is signaled.

However, if we modify packet 4 as follows, there will be a definite error:

```
4: EchoReply Event   <seq:5, data:''junk''>
4:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
```

The sequence number of the reply does not match any of the requests, so no amount of buffering or loss can explain the behavior of the receiver in this case. The resulting output trace confirms this reasoning:

```
4: Possible PingError Event  <''Incorrect Echo Reply''>
4: Definite Error
```

Thus, channel transformations relieve us of having to reason explicitly about buffering and loss in our NERL programs. Instead, the programmer simply annotates the input events, and the compiler generates code to account for all the possible states of the monitored session.

4.7 Model Checking

In the previous sections, we learned how to write NERL programs to monitor several variations of the Ping protocol. For each of these programs, there was an implicit claim that the monitor would raise an error if and only if one of the protocol participants failed to conform to the protocol specification. In this section, we demonstrate how we can formalize such claims and prove them using the SPIN model-checker (Hol).

To use this model checking feature, the programmer must be familiar with SPIN and its modeling language Promela. Even though the NERL compiler automatically generates Promela code for a large part of the analysis, formalizing the properties to be checked and understanding the output of the model-checker require some expertise.

Suppose that for the simple two-message Ping recognizer we designed before, we want to prove the following meta-event property.

Property 4.1 *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 want to prove that the Ping recognizer does not generate any false alarms.

Property 4.2 *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 does not produce any* BadRequest *or* BadReply *event.*

To prove these properties, we first translate the NERL recognizer for Ping to a Promela model. (The NERL compiler has an inbuilt translator that will generate a Promela model for any recognizer module.) Then we model the Ping sender and the Ping receiver in Promela, by encoding the state machines from Figures 4.4 and 4.5. This complete system (sender + receiver + recognizer) is then handed over to SPIN along with the two properties expressed

as Linear Temporal Logic (LTL) formulae. The SPIN model-checker then analyzes this system to find any executions that may violate the properties.

The Promela models and LTL properties for Ping have been included in Appendix C.5. To limit the state space, our Ping sender chooses from among 10 sequence numbers and all data fields are empty strings. For this configuration, SPIN found no violations of the two properties. To check the first property SPIN analyzed 6299 states and 23780 transitions. For the second property, SPIN checked 6347 states and 8807 transitions. Therefore, even for simple recognizers and properties, there may be several cases that need to be analyzed to prove that the recognizer has been coded correctly. We find that it is useful to add an automated checking component that avoids the manual checking of each case. In many of the recognizers included in this thesis, the model checker found small logical errors that escaped our attention.

4.8 Event Tracing

When monitoring multi-message protocols, if the NERL program raises an error, it is often useful to know the exact sequence of messages that caused the error. This becomes quite difficult if the error has been raised deep into the trace when monitoring several sessions. The NERL compiler provides a special run-time feature called event tracing, based on forward dynamic slicing, that can produce the exact sequence of messages in the input trace that caused a specific error.

Overall, 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, the following is the output trace generated when the Ping recognizer is executed with tracing turned on:

```
2: IsAlive Event <''saul-a''>
2:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
2: Depends on:<1,2>
4: IsAlive Event <''saul-b''>
4:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13044]
4: Depends on:<3,4>
```

Each line is prefixed with the current packet number, and the "Depends" clause shows the packets that the current output event depends on. Here, the user knows that the IsAlive event at packet 4 was caused by the packets 3 and 4. To debug or understand this event, the user does not have to look at the other packets in the trace. Essentially, the tracing tool maintains all the dynamic control and data dependencies of state variables (and output events) on input events. In particular, in a multi-message protocol the event trace will pick out exactly those messages that belong to the session in which the output event is produced.

In some cases, the user is not interested in the control packets that set up a session and only wants to see the packets that contain the data that caused an output event. The user can then run a modified dynamic slicing tool called data tracing that only maintains data dependencies. For instance, in the Ping recognizer, the IsAlive event contains icmp_data as an attribute, and this is copied from the EchoReply event. Since the IsAlive event does not have any data dependency on the earlier request, if we execute the NERL recognizer with only data tracing turned on, we get:

```
2: IsAlive Event <''saul-a''>
2:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13043]
2: Data Depends on:<4>
4: IsAlive Event <''saul-b''>
4:   at P[ip_src:158.130.012.217,ip_dst:158.130.012.004,icmp_id:13044]
4: Data Depends on:<4>
```

This trace indicates that the icmp_data attribute in the IsAlive at packet 2 only depends on the attributes of packet 2. For multi-message protocols, data dependencies give the user a sense of the packets whose data caused an output event.

CHAPTER 5

NERL LANGUAGE IMPLEMENTATION

In Chapter 4, we introduced NERL, a domain-specific language for programming protocol monitors. For programs written in this language, the NERL compiler produces executable code for monitoring. In addition to this normal mode of operation, the compiler supports several advanced modes. In *model checking* mode, the compiler produces a formal analyzable model, instead of the executable monitor. In *event tracing* mode, the compiler automatically generates code for computing and printing diagnostic information with output events. In *channel transformation* mode, the compiler generates code for a co-networked monitor for the protocol recognizer. In the following sections, we describe the implementation of each of these features in order.

5.1 Generating an Executable Monitor

The NERL compiler translates recognizers written in NERL to recognizers in C. The C program is then compiled with NERL-specific libraries to an executable monitor. This two-step approach gains the benefits of using a domain specific language, while retaining the power of a general programming language.

Compiling to a general-purpose programming language has several benefits. First, the compiler does not have to generate machine code or implement low-level optimizations. Instead, it can rely on optimizations in the C compiler for efficient code generation. Second, the output of the compiler is readable C code. As a result, we can debug the compiler by inspecting the output. Moreover, we can use C-based tools for debugging the monitor, such as finding memory leaks, and for profiling its performance. And if the performance is found to be inadequate, we can directly modify the C program, to implement more efficient data structures for instance. Third, the generated C program is compatible with pre-existing libraries and data structures written in C. For instance, in our case studies, we use the libpcap packet capture library to capture and parse packets, and we use programs written in Lex to parse emails, and both these programs can easily be linked and compiled with NERL protocol recognizers.

On the other hand, using a domain-specific language has several benefits too. First, NERL provides a simpler, specialized syntax for writing monitors. Second, we can design and implement advanced tools that work for monitors written in this specialized syntax; we can model-check the recognizer for errors, we can compile error diagnostics into the executable monitor, and we can perform automated channel transformations. Third, the compiler automatically generates code for common, error-prone tasks and for specialized data structures. In particular, the NERL compiler provides the following:

- It provides implementations with built-in error checking for commonly used data-types: strings, arrays and variable arrays.

- It generates code to handle all pointer manipulation. For instance, it automatically generates code for all memory allocations and freeing for recognizer state variables and transient event attributes. Moreover, aliasing of event attributes is implemented using pointers.

- It generates code for printing events, along with all attributes. While this code is mainly boiler-plate, it produces a separate function for each event and can accurately format and print strings, arrays and variable arrays occurring in event attributes.

- It generates code for spawning recognizer instances on-demand and deleting them when they are no longer necessary. When monitoring an execution with several hundred sessions at several layers in the stack, this code generation is very useful since it avoids errors in pointer manipulation.

In the rest of this section we describe the translation of NERL programs to C. The section is divided into four parts. Section 5.1.1 outlines the translation of recognizer modules to C, emphasizing only the non-standard parts. Section 5.1.2 outlines the translation for main modules. Section 5.1.3 sketches the static and run-time type checking for NERL programs. Finally, Section 5.1.4 gives some implementation details.

5.1.1 Translating Recognizers

The basic building blocks of NERL recognizers are events and statement blocks. Translating NERL statement blocks to C is straightforward: conditionals, while loops, local variables, sequencing, arithmetic expressions, and boolean conditions are carried over with minor syntactic changes. Booleans are translated in a standard way to C, using the integer 0 for false and 1 for true. Type definitions similarly translate to typedefs in C, with record types represented as structs. Every atomic type, which may either be an inbuilt type or defined type variable, is associated with a default value that is used to initialize variables and array cells of that type. The default values for some sample types are assigned as shown in Table 5.1.

The main complexity of the translation, therefore, is in the translation of strings, arrays and events, and the way these are combined in a recognizer function.

Table 5.1 Default Values

NERL Variable Declaration	Default Value Assignment (C)
bool b	b = false;
int i	i = 0;
double d	d = 0.0;
string s	s.length = 0; s.value = "";
ty_foo a[10]	a.length = 10;
	for (*int* i = 0; i<10; i++)
	a.elements[i] = *ty_foo_default_elem*;
ty_foo a[]	a.length = 0; a.elements = null;

Strings NERL strings are translated to a new string datatype in C. We define a string in C as a pair consisting of a character array and an integer length. Strings can be copied: string_copy(&t,s) copies the string s to the location t. Two strings can be concatenated: string_concat(&t,s) concatenates s to the end of t; if t does not have enough space, then more space is allocated. Finally, two strings can be compared for equality using strncmp. The following table represents the translation of NERL strings to C:

NERL Construct	↪	C Translation
string s;	↪	*struct* {*int* length; *char*∗ value} s;
s = "foo";	↪	string_copy(&s,"foo",strlen("foo"))
s = t;	↪	string_copy(&s,t.value,t.length)
s == "foo"	↪	(s.length == strlen("foo")?
	↪	(strncmp(s.value,"foo",s.length)==0): false)
s == t	↪	(s.length == t.length?
		(strncmp(s.value,t.value,s.length)==0): false)
concat(s,t);	↪	string_concat(&s,t)

The translation differentiates between literal strings and memory allocated strings. Literal strings ("foo") are represented as constant strings in C, but for concatenation and copying, they are treated as if they had a length field with the appropriate value (strlen("foo")).

Arrays We implement an array library in C that extends the usual arrays in two ways: it manages variable-sized arrays by allocating and freeing memory blocks on demand, and it checks for array bounds violations at run-time. Each NERL array is represented by a pair

consisting of a C array and an integer length field; for fixed-size arrays, this pair stays constant. The operator a.**length** extracts the length field, and a.elements[n] extracts the nth element. For run-time safety, array accesses are guarded by a bounds check, and if the index n exceeds the array length, then a run-time error ArrBnd is raised and the recognizer instance terminates (but other instances can continue). For variable arrays, we provide the function array_extend(&a,d,n) that extends the array a at the end with n elements initialized to the default element for the element type. (Each declared type has a default element). Conversely, the function array_delete(&a,n) deletes the last n elements from a. The translation of NERL arrays is depicted in the following table:

bool a[10]	$\hookrightarrow$	*struct* {*int* length; bool elements[10];} a;
ty_foo a[]	$\hookrightarrow$	*struct* {*int* length; *ty_foo*∗ elements;} a;
a#**length**	$\hookrightarrow$	a.length
a[n]	$\hookrightarrow$	(n >= a.length? err(ArrBnd) : a.elements[n])
push(a,n)	$\hookrightarrow$	array_extend(&a,*ty_foo_default_elem*,n)
pop(a,n)	$\hookrightarrow$	((2 > a.length)? err(ArrBnd): array_delete(&a,n))

Events Events are represented by a pair consisting of a boolean flag and a pointer that refers to the event attributes. When an event E(r) **WithAttributes** S is generated, E.flag is set to true, and its attributes r = E.attrib are assigned according to S.

event *ty_foo* E	$\hookrightarrow$	*struct* {bool flag; *ty_foo*∗ attrib;} E;
E **WithAttributes** {}	$\hookrightarrow$	E.flag = true
E(r) **WithAttributes** {r.x = 0}	$\hookrightarrow$	E.flag = true;
		{*ty_foo*∗ r; r = E.attrib; r−>x = 0;}

Event Patterns An event pattern P is compiled to a partial statement that is a conditional guarding a hole: **if** P' {...}. This hole is filled by the statement that the pattern is guarding; for instance, the state transition P −> S is compiled to **if** P' {S'}. The event pattern E(r) **OccurredWhen** C is compiled to a conditional expression that checks whether E has occurred (E.flag is true), and if it has, then it defines r and points it to E.attrib, and it checks whether C is true. The other patterns are defined similarly:

E(r)	$\hookrightarrow$	**if** (E.flag) {*ty_foo*∗ r;r = E.attrib;...}
E | F	$\hookrightarrow$	if (E.flag) {...} else if (F.flag) {...}
E & F	$\hookrightarrow$	if (E.flag) { if (F.flag) {...}}
E(r) **OccurredWhen** (r.b)	$\hookrightarrow$	if (E.flag) {*ty_foo*∗ r;r = E.attrib; if (r−>b) {...}}

Recognizers A snapshot or *instance* of a NERL recognizer consists of the values of its state variables and events. In C, we represent an instance of a recognizer R by a *control block*: a record cb of type R_cb∗ that consists of the state variables cb−>**state**, input events cb−>inputs, and output events cb−>outputs. The commands of the recognizer are translated to a function R_recognize(&cb) that modifies the control block.

A single round of the recognizer instance cb executes as follows:

- Check whether an input event cb−>inputs.I has occurred
- Execute commands in R_recognize(&cb), possibly modifying cb−>**state**
- Produce output events cb−>outputs

To see a simple example of the translation, consider the NERL recognizer defined as follows:

```
Recognizer R =
 state int lastfoo
 input event bool Init;
 input event int Foo;
 output event string Bar;
 output event bool Done;

 transition Foo(n) -> {lastfoo = n};
 event Foo(n)
        OccurredWhen (n == 0) -> Bar(s)
                                 WithAttributes
                                 {s = "foo with zero"};
 event Bar -> Done;
EndRecognizer
```

It gets translated to the C program fragment that follows:

```
1  typedef struct { int lastfoo; } R_state;
2
3  typedef struct {
4   struct R_state* state;
5   struct { struct {bool flag; bool attrib;} Init;
6            struct {bool flag; int attrib;} Foo; } inputs;
7   struct { struct {bool flag; string attrib;} Bar; } outputs;
8  } R_cb;
9
10 void R_recognize(R_cb* cb) {
11  if (cb->inputs.Init.flag == true) {
12      cb->state = (R_state*) malloc(sizeof(R_state));
13      cb->state->lastfoo = 0;
14  }
15
16
17  if (cb->inputs.Foo.flag == true) {
18      int n;
19      n = cb->inputs.Foo.attrib;
20      lastfoo = n;
21  }
22
23  if (cb->inputs.Foo.flag == true) {
24      int n;
25      n = cb->inputs.Foo.attrib;
26      if (n == 0) {
27        cb->outputs.Bar.flag = true;
28        {string s;
29         string_copy(&s,"foo with 0");
30        }
31      }
32  }
33
34  if (cb->outputs.Bar.flag == true) {
35      cb->outputs.Done.flag = true;
36      /* Main module executes: free(cb->state); */
37  }
38 }
```

First (Line 1), the declarations of the state variables get compiled to the type definition of R_state. Then (Lines 3—8), the event declarations are translated to the type definition of control blocks Rcb+. Lines 10—37 depict the recognizer function that contains all the translated commands. Since the NERL recognizer had no transition for **Init**, the default transition is produced (Lines 11—14): it allocates memory for the state variables, and initializes them. Each of the commands is translated in order: first the state transition for Foo (Lines 16—20), then the event definition for Bar (Lines 22—31), then the event definition for **Done**.

In Appendix C, we present a detailed example of the translation for a more complicated recognizer module. Section C.1 lists the NERL code for the Ping recognizer module, while Section C.2 lists the corresponding C translation.

5.1.2 Translating Main Modules

The main module is translated to a main() function that manages all the active recognizer instances: it creates and deletes instances on demand, and in every round, it captures one packet, executes the stack of active instances, and prints out the output events.

For instance, consider the Ping main module in Chapter 4 (Section 4.4), corresponding to the recognizer stack in Figure 4.8. We assume a packet capture module PCap with a single instance called C. C feeds events to instances P[S] of the recognizer Ping, where the index S is the protocol session identifier. Recall that the main module for Ping consists of commands that forward each output event ICMP of C to the input event of some instance P[S]. When this instance is executed on the input event, its state is modified and an output event may be produced.

The NERL compiler translates this main module to a program that executes the algorithm in Table 5.2 Implementing this translation is mostly straightforward; we detail some of the interesting steps below.

Signatures Recall that the main module consists of signature declarations, instance declarations, and event forwarding definitions. The signature declarations summarize the state and event variables needed to control a recognizer instance. Signatures get compiled to control block types exactly as in the recognizer module translation (Section 5.1.1, Recognizer).

Table 5.2 Ping Main Module Execution

Data Type C one instance of the packet capture module.
P[] a table of instances, one for each session of Ping.
Initially P is empty, C is created and initialized.

Algorithm

1. Execute the packet capture module C to produce the next captured packet
2. If no packet has been captured, return to 1
3. Let the captured packet event be ICMP with attributes a (written ICMP(a))
4. Find the recognizer session S to which $ICMP$ must be sent
5. If P[S] does not exist, create and initialize it
6. Execute the instance P[S] on input ICMP(a)
7. If any error events are generated, print them and delete P[S]
8. Return to 1

Recognizer Instances Each instance corresponds to one control block. Declarations of single instances are initialization operations for the corresponding control block. Declarations of instance tables translate to initializing a hash table of control blocks, with instance indices translating to keys. Initially, none of the instances in the table are generated or initialized. When an event forwarding definition triggers the first input event for an instance, then the instance is generated (its control block is allocated), it is initialized, and it is inserted into the hash table.

Output Event Patterns The output event patterns in main modules are similar to the event patterns in recognizers, with an additional variable for the instance index. Consequently, their translations are similar:

```
I.E(r)     ↪   if (I->outputs.E.flag == true)
                 {ty_foo* r; r = I->outputs.E.attrib;...}
I[n].E(r)  ↪   if (I->outputs.E.flag == true)
                 {int* n; ty_foo* r; r = I->outputs.E.attrib;...}
```

Event Forwarding Definitions An event forwarding definition consists of an output event pattern guarding an input event; it gets translated to the following sequence of commands

- Check if an output event for A.O (A−>outputs.O) has occurred

- Set an input event B.I (B−>inputs.I)

- Execute B (R_recognize(&B))

This translation is described as follows:

```
J.F(s) WithAttributes {s = r.x}   ↪   J−>inputs.F.flag = true;
                                      { int* s; s = J−>inputs.F.attrib;
                                        *s = r−>x; R_recognize(J);}
```

In Appendix C, we present a more detailed example of the translation. Section C.3 lists the NERL code for the main module for the Ping protocol, while Section C.4 lists the corresponding C translation.

5.1.3 Type Checking

State and local variables in a NERL program may have a basic type **int**, **double**, **bool**, **string**, or a record type, or an array type. For each type, only a specific set of operations is well-defined. For instance, concatenation works only for strings, arithmetic operations for ints and doubles, array indexing works for arrays, push and pop for variable-sized arrays, and field projection for records. If any of these operations is used on variables or expressions of the wrong type, we consider it a type error. The NERL type checker points out such simple errors in NERL programs.

In addition to this static type checking, the NERL compiler inserts run-time checks into the executable monitor so that the recognizer does not crash unexpectedly.

- Every array access is guarded by an array bounds check; if it fails, an error event ($ArrBndErr$) is triggered and the recognizer instance is shut down.

- Every array pop operations is guarded by an empty array check; if it fails, an error event ($EmptyArr$) is triggered and the recognizer instance is shut down.

- If the same event is generated twice in the same instance in the same round, an error event ($RepeatedEvt$) is triggered and the recognizer instance is shut down.
- If the same state variable is modified twice in the same instance in the same round, an error event ($WriteErr$) is triggered and the recognizer instance is shut down.

The first two run-time errors denote illegal accesses of array locations; they should be seen as graceful segmentation faults. The last two denote artificial restrictions on the run-time behavior of recognizers that, in our opinion, make the recognizer easier to read and understand.

5.1.4 Implementation Details

We have written a fully functioning NERL compiler; it was used to generate code for all the recognizers described in this thesis. However, it is still under development and often produces inefficient C code that needs to be optimized by hand. Automated NERL-specific optimizations are a topic of future research.

The compiler is written in the functional language OCaml. The NERL parser is written in OCamlYacc/OCamlLex and produces an abstract syntax tree. The type-checker, written in OCaml, checks this tree for errors. Finally, the code generator translates the tree into C code that is produced as output. The generated C code is linked with libraries for arrays and strings, and C programs for the packet capture module and the print module. We find that using a higher-order, strongly-typed language significantly simplifies the implementation. The compiler was written in a few thousand lines of code and contained relatively few bugs.

5.2 Model-checking NERL Programs

In model-checking mode, the NERL compiler translates a recognizer to a formal model in 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 $ArrBndErr$ exception.
- We can check that the NERL recognizer is correct with respect to the specification. In particular, we can check that
  - it produces meta-events only if a correct protocol participant would, and
  - it produces error-events only if the protocol specification is violated.

Ideally, we would like to check the entire recognizer stack for a suite of protocols to be correct. However, to reduce the problem to manageable complexity, we only check one recognizer at a time. Our approach consists of the following steps:

1. Translate recognizer for protocol P into Promela
2. Write Promela models for the participants (sender, receiver) of protocol P
3. Write the properties we want to check as assertions or Linear Temporal Logic formulae
4. Feed all of the above to SPIN and analyze the results of the verification

Although, the model-checking feature of the NERL compiler automates the second step, none of the remaining steps are automatable. The NERL programmer must manually write the models for the sender and receiver, and manually interpret the results of SPIN. This requires a relatively deep understanding of Promela and SPIN. Moreover, when the SPIN verification does not terminate, it is often incumbent on the programmer to introduce abstractions into the recognizer model to make the state space more tractable.

For instance, Appendix C.5 lists the Promela models for the Ping protocol. The models for the sender, the receiver, and the main module are listed as `pingmain.spin`. This code was

hand-written. The model corresponding to the Ping recognizer is listed as ping.spin. This code was automatically generated by the NERL compiler.

In the remaining part of this section, we overview the key aspects of the translation from recognizers to Promela models as implemented by the NERL compiler.

Abstractions The Promela translation is similar to the C translation in many ways, primarily because Promela itself has a C-like syntax. One important difference is that since Promela is a modeling (not a programming) language, it does not support data types such as strings and floating-point numbers. So, the translation must introduce *abstractions* for variables with these types and operations on such variables.

For instance, 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. 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 Arrays are translated to Promela arrays, and variable sized arrays translate to large fixed-size arrays of size AMAX (another abstraction) with an attached length attribute (initially zero). The dynamic checks that we inserted in the NERL compiler are inserted as assert statements that can be checked by the SPIN model checker.

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

event ty_foo E $\hookrightarrow$ **chan** E = [1] **of ty_foo**;

Here, E is defined to be a channel with buffer size 1; values of type **ty_foo** can be sent and received on E. Event generation is translated to send operations on channels (written E!x). To

check whether an event has occurred, the channel must be polled (written E?[x]). Because of restrictions in Promela syntax, the dynamic channel initialization in the above translation is not entirely correct. Instead, the translated recognizer defines a table of available channels in the beginning and each event is assigned one of these channels.

Event Patterns An event pattern translates, as in the C compiler, to a conditional expression guarding a hole.

```
E(r)  ↪  ty_foo r;
         if
         :: E?[x] -> X?<x>; ...
         :: else -> skip
         fi
```

Here, we first check if there is any input on the E event channel; if there is, then we extract the buffered event attributes into x, and then execute the guarded statement (...).

Recognizers 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. As an example, the Promela proctype corresponding to the Ping recognizer is shown in Appendix C.5.

Implementation Details 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 7), we used it to prove that commonly-used SMTP monitor specifications are incorrect for our purposes.

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 complex 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 automate some of these abstractions, but no such strategies have been implemented.

5.3 Event Tracing

To find errors in sequential programs, programmers use debugging environments, such as gdb (http://www.gcc.org). One of the more useful features in a debugger is the ability to set *watchpoints*. Using watchpoints, the debugger can monitor the program for a simple boolean expression on its state variables and stop execution when the expression becomes true. For instance, a watchpoint for `x<0` will stop execution when the variable `x` has a value less than `0`.

However, debuggers can check an execution only for violations of state invariants. They cannot check for properties of sequences of states, without additional coding. For instance, a watchpoint cannot check that the value of the variable `x` increases over time. On the other hand, NERL programs can monitor executions for the full range of computable safety properties. This additional expressiveness comes at the cost of diagnostics. When a state invariant is violated, printing the current state is enough to understand the error. But when a NERL program detects a bad sequence of states, raising an error event at the end of this sequence is not enough, it must also provide information about the intermediate steps that caused the error event. For instance, when the property that the value of `x` is non-decreasing is violated, we need to know not only the current value of `x`, we also need to know the previous value.

This is a general diagnostic problem with any stateful monitoring framework. For instance, when an intrusion detection system signals an intrusion on the basis of a sequence of

packets, it should provide information about the entire sequence for the event to make sense to a network manager.

For a protocol recognizer written in NERL, an alarm is the immediate result of the current value of some state variables and the current input event. So, the sequence of inputs that resulted in this alarm consists of the current input event and all the current and previous input events that caused these variables to have the current values. This sequence of input events is what we call a *dynamic event trace* for the alarm. This trace is the shortest sub-trace of the monitored packet stream that represents a violation of the safety property monitored by the NERL program.

The computation of the dynamic event trace for an alarm is completely mechanical, and NERL provides a compiler feature that automates it, saving the programmer time and effort. For each variable, the compiled monitor maintains a *dependency set*: the set of past input events that affected its current value. This set encodes the causal relation between input events and state variables. When an alarm is raised, its dependency set is also output as an attribute. In the later case studies, we will see how this set is useful in diagnosing errors in the protocol implementation. The NERL compiler's event tracing feature maintains the causal relation from input events to state variables.

In the following sections, we sketch implementation details for the event tracing feature. The tracing algorithm is a version of forward dynamic slicing (KY94) and is similar to event trace slicing (SK00).

Representing Dependency Sets First, each packet that enters the system is given a unique integer identifier, called a *timestamp*. Then, to each state variable and event, we add a depends attribute that contains a set of timestamps corresponding to the packets that affected its value. For instance, the C translation of state variables and events now looks like the following:

NERL Variable Declaration	$\hookrightarrow$	C Translation
double d;	$\hookrightarrow$	**struct** {**double** value; IntSet depends;} d;
event ty_foo E;	$\hookrightarrow$	**struct** {bool flag; **int** timestamp; IntSet depends; **ty_foo**$*$ attrib;} E;

The dependency sets are represented by the IntSet datatype, implemented as a sorted list of integers. The operations that can be performed on IntSet variables are: intset_clear(s) (delete all elements in s), intset_copy(&t,s) (delete all elements in t and copy the set s to t), and intset_union(&t,s) (add elements of s to t).

Initially, the depends attribute of each variable, local event, and output event is empty. An input event has the depends attribute set by the lower layer; if it represents a packet event then the depends attribute is a singleton set containing the new timestamp assigned to it by the packet capture module. Any event that is generated as a result of this input event will contain this timestamp in its dependency set; any variable that is modified as a result of this input event will contain the timestamp in its dependency set as well.

Control and Data Dependencies In general, a state (or event) variable that appears within a state transition (or event definition) is control dependent on all state and event variables used in the guarding event pattern. In addition, if it appears within an if-then-else or while-do statement, then it is also control dependent on all state variables that appear in, and all events whose attributes appear in, the conditional expression.

On the other hand, a state variable that is the target of an assignment statement is data dependent on all state variables that appear in the source expression and on all event variables whose attributes appear in the source expression. An event variable that is the target of an event definition is data dependent on all state variables that appear in, and all events whose attributes appear in, the target attribute assignments.

Example Let us detail the translation of a simple state transition. Let I be an input packet event, and x be an state variable that is incremented every time an I event with $I.a > 0$ is seen. Then the dependency set of x is the union of all the dependency sets of such I events seen till date:

```
transition I -> {x = x+1};   ↪   if (cb->inputs.I.flag == true) {
                                    x = x  1;+
                                    intset_union(&(x.depends),cb->inputs.I.depends;}
```

We distinguish between two kinds of dependencies for x in this state transition. We say that x is *control dependent* on the input event I, meaning that I influences whether x will be modified. We say that x is *data dependent* on itself, meaning that the new value of x depends on the old value of x. The other constructs are translated similarly.

Implementation We have implemented the tracing feature as a modification of the NERL compiler. The new compilers translate recognizers to C programs that have the tracing code enclosed in #**ifdef** CTRACE ... #**endif**. So to turn the tracing on, the C recognizer is compiled with the −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 in NERL programs, there is often a group of variables that is always updated together. In such cases it is more efficient to have a single 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.

5.4 Channel Transformations

Programming a passive monitor to work in a real-world monitoring environment requires the programmer be aware of the possibility of packet loss and delay in the network. This means that when a monitor captures two packets, say P followed by Q, it is possible that there was a third packet R in between that got lost, and it is possible that Q was actually emitted before P. Since keeping track of these possibilities can easily become cumbersome for the programmer, the NERL compiler automates this process by implementing the trace-search algorithm, first introduced by Bhargavan, Chandra, McCann, and Gunter (BCMG01), which searches for all the traces that can correspond to a captured packet trace.

Bhargavan *et al.* show that even in a controlled co-networked monitoring environment, such as an Ethernet-based local area network, packet loss and delay can cause the captured packet trace to deviate from the actual packet trace at the monitored devices. They formalize this class of *trace infidelities* and propose the trace-search algorithm: a generic technique for transforming a monitor that assumes a perfect monitoring channel to a monitor that keeps track of co-networked trace infidelities. In effect, the algorithm transforms a monitor M to a monitor $T(M)$ such that given an actual trace T of packet events and a (possibly different) trace C captured by a co-networked monitor, the transformed monitor $T(M)$ will raise an error when processing C only if the original monitor M would raise the error on T. A detailed presentation of the trace search algorithm with a proof of correctness is presented in Appendix B. The time complexity of this algorithm is either exponential in the length of the captured trace or exponential in the number of states of the monitor program, whichever is smaller. In practice, this algorithm is effective as long as at any given time, the number of possible states of the monitor is quite limited.

The NERL compiler implements the trace-search for monitors written in NERL. Because of the state space limitations imposed by the algorithm complexity, we implement the algo-

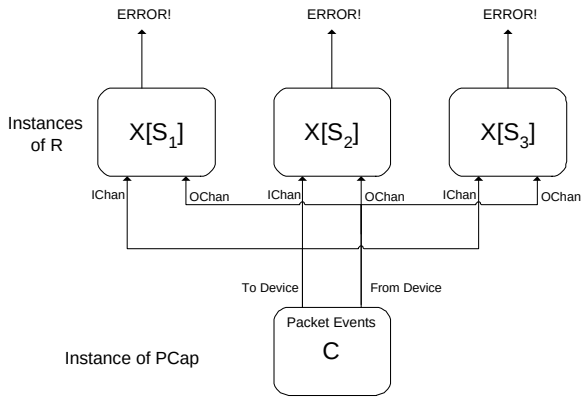

Figure 5.1 Typical Monitoring Stack for Trace Search

rithm only for the most common case of co-networked monitoring. We consider recognizer stacks with at most two levels: the packet capture module at the bottom level and instances of a single recognizer at the second level. The recognizer monitors at most two channels, called the input channel and output channel, where only the input channel is allowed to have a buffer that can lose and delay input events. Figure 5.1 shows a typical recognizer stack for which the algorithm works. This is the same configuration as in the Ping example (Figure 4.8) with some additional information about the monitored channels. As before, we assume a packet capture module with a single instance C. Here, C feeds events to instances X[S] of recognizer R, where the index S is the protocol session identifier. We assume that the recognizer R has a single output event that signifies an error.

For trace search, we assume that the signature of a recognizer R has been annotated so that the input events are divided into two categories: those on the input channel IChan, and those on the output channel OChan. The intuition behind these annotations is that the packets on IChan are going from the monitor toward the device under test, and the packets on OChan are output from the device toward the monitor. We assume that the packets on IChan can be dropped before reaching the device, but at most MAX_LOSS packets can be lost in a row. Moreover, we assume that packets on IChan can be delayed by a buffer of size MAX_BUF

before reaching the device. On the other hand, packets on OChan never get lost or delayed. As we argue in Chapter 2, these assumptions are reasonable for a co-networked monitoring environment, where the only source of packet delay and loss is the input buffer of a heavily loaded network device.

To see an example of an annotated signature, consider the following Ping signature declaration:

```
recognizer PingRep : {
 channel  IChan[5,2];
 channel  OChan[0,0];

 input event bool Init;

 input event IChan   Ty_echo EchoRequest;
 input event OChan   Ty_echo EchoReply;

 output event string IsAlive;
 output event string PingError;
 output event bool Done;
}
```

Here, MAX_BUF is 5 and MAX_LOSS is 2.

The normal execution algorithm for this NERL program, without trace search, has already been presented in Table 5.2. In normal mode, we think of each recognizer instance as modeling one participant of one session of a protocol. With trace-search, the execution algorithm is significantly more complicated, and is presented in Table 5.3. Now, the monitor must not only model every protocol participant, it must also model the non-deterministic input buffer at each participant. For each instance X[S] we maintain a set of possible states, one for each possible trace. Each possible state is a triple (Instance,input buffer,input loss streak), where the input buffer is a list of packets that have been buffered at the device, and the input loss streak is the number of packets that have been dropped before the device. For every packet seen on IChan there are three possibilities: (1) the packet reached the device immediately, (2) the packet was bufferred, (3) the packet was lost. For every possible state of X[S], these three possibilities give rise to three new possible states. On the other hand,

Table 5.3 Main Module Execution with Channel Transformations

Data Type P one instance of the packet capture module.
X[] a table containing entries of type BufSet.
Each BufSet is a set of triples (C,B,L), where C is an R instance, B is a sequence of events representing the input buffer, and L is a positive integer representing the input loss streak.
Initially X is empty, P is created and initialized.

Algorithm Execute the following:

1. Execute the packet capture module P to produce the next captured packet
2. If no packet captured, return to 1
3. Let the captured packet event be E with attributes a
4. Find the recognizer session S to which $E(a)$ must be sent
5. If X[S] does not exist, create and initialize it to a singleton set (Init,empty,0), where Init is an initialized instance.
6. If E(a) is on the input channel then for every (C,B,L) in X[S] do the following:
   (a) Delete (C,B,L) from X[S]
   (b) If B has less than MAX_BUF events then
      i. Copy C to C1
      ii. Insert (C1,B::E(a),0) into X[S] (possibility 1: E is buffered)
   (c) If L is less than MAX_LOSS then
      i. Copy C to C1
      ii. Insert (C1,B,L1)+ into X[S] (possibility 2: E is lost)
   (d) If B is empty then
      i. Copy C to C3
      ii. Execute C2 on input event E(a) (possibility 3: E has reached)
      iii. If an error event is not raised, then insert (C2,B,0) into X[S]
7. If E(a) is on the output channel then for every (C,B,L) in X[S] do the following:
   (a) Delete (C,B,L) from X[S]
   (b) Execute C on input E(a)
   (c) If an error event is not raised, then
      i. Insert (C,B,L) into X[S]
      ii. For every non-empty prefix B1 of B (B = B1 B2)
         A. Copy C to C'
         B. Execute C' sequentially on all events in B1
         C. Insert (C',B2,L) into X[S]

when an output event is seen, because of our assumptions it can be executed immediately on all the possible states. The proof of correctness of this algorithm derives from the proof of correctness for the general trace-search algorithm that is presented in Appendix B.

The trace search algorithm in Table 5.3 is implemented as an alternative translation of the main module in the NERL compiler. The compiler interprets the channel annotations on the recognizer signature to find the MAX_BUF and MAX_LOSS constants. It then automatically generates data types for the possible states. For instance, the possible states of a Ping recognizer would be represented by the following Ping_entry datatype.

```
typedef struct {
  queue ibuf;
  int   streak;
  Ping_cb* cb;
  struct Ping_entry* next;
} Ping_entry;
```

Here, ibuf is the input packet buffer, streak is the input loss streak, cb is the state of an instance. The BUFSET is represented as a linked list of such entries:

instance R X **WithAttributes** S $\hookrightarrow$ {R_entry* X_cb;}

The implementation of the rest of the algorithm is straightforward. The channel transformation feature 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 8. To monitor multiple layers, we shall need more optimizations.

CHAPTER 6

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 Mobile Ad-hoc Networking (MANET) working group at the Internet Engineering Task Force (IETF,http://www.ietf.org) 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 (BMJ+98). 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 they 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 only consists of simulation studies (BMJ+98, LH98, YEG00). We know of no previous work that tests these implementations for correctness. However, simulator implementations for protocols typically contain 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 precomputed expected performance chart for the scenario.

Testing based only on performance measures is inadequate for the careful analysis of a protocol. First, 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 violations for example. Second, 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.

In an earlier work, we developed a simulation analysis framework, Verisim (BGK+02). Verisim used a monitoring language called MEDL (LKK+99) to program protocol monitors. In this chapter, we reimplement Verisim with NERL as the monitor programming language.

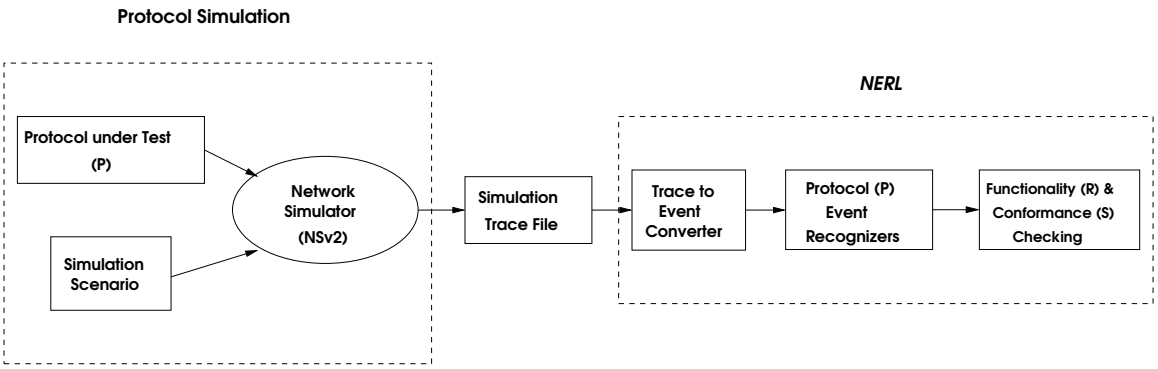

Figure 6.1 The Verisim Framework

The Verisim framework is shown in Figure 6.1. 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 the 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.

6.1 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.

6.1.1 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).

Each node maintains a table of *routes*. Each route represents the local information needed to send packets on the path leading to a destination. A route to a destination d contains the following fields:

$\mathsf{next\_hop}_d$: Next node on a path to d.

$\mathsf{hop\_cnt}_d$: Distance from d, measured in the number of nodes (hops) that need to be traversed to reach d.

$\mathsf{seq\_no}_d$: Last recorded *sequence number* for d.

$\mathsf{lifetime}_d$: 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 to a destination d is established, it is stamped with the current sequence number at d. As the topology changes, old routes become invalid and new routes are computed; More recent routes will have larger sequence numbers; this 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:

$$\mathsf{RREQ}(d, \mathsf{hops\_to\_src}, \mathsf{dest\_seq\_no}, s, \mathsf{src\_seq\_no}).$$

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 will 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 that has the following format:

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

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

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

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 depending on how long it expects to be available at that location. Every node that receives a RREP increments the value of the hops_to_dest packet 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_d 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.

6.1.2 The AODV State Machine

Network protocols represent reactive systems, which means that every action is carried out in response to an event. Although protocol standards are written in natural languages, one can typically extract the state machine that it is trying to express. In the last section, we informally described the AODV state machine. In Table 6.1.2, we present a fuller and more formal description.

The AODV specification (Per97) is an evolving document that describes the various packets and network events that an AODV process responds to. Here, we present the reactive state machine that an implementation of AODV version 0 is supposed to implement.

An AODV node runs a state machine for each destination; Table 6.1.2 depicts the state machine for the destination *dst* at node *me*. There are two control states corresponding to the presence or absence of a route at *me* to *dst*. When the node has a route, it keeps track of the best known route to *dst* using four state variables: seq_no, hop_cnt, next_hop, and lifetime. 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 of an AODV process.

Table 6.1 AODV State Machine

STATE: No Route		
Condition	**Action**	**Next State**
TimeOut	seq_no $\leftarrow 0$	No Route
Recv from p: RREQ(d, hops_to_src, dest_seq_no, s, src_seq_no) $\wedge\ d =$ dst	dest_seq_no $\leftarrow$ $\max$(seq_no, dest_seq_no); Broadcast RREQ(d, hops_to_src $+ 1$, dest_seq_no, s, src_seq_no)	No Route
Recv from p: RREQ(d, hops_to_src, dest_seq_no, s, src_seq_no) $\wedge\ s =$dst $\wedge$ src_seq_no $\geq$ seq_no	next_hop $\leftarrow p$; hop_cnt $\leftarrow$ hops_to_src $+ 1$; seq_no $\leftarrow$ src_seq_no; lifetime $\leftarrow$REV_ROUTE_LIFE;	Has Route
Recv from p: RREP(hops_to_dest, d, dest_seq_no, route_lifetime) $\wedge\ d =$dst $\wedge$ dest_seq_no $\geq$ seq_no	next_hop $\leftarrow p$; hop_cnt $\leftarrow$ hops_to_dest $+ 1$; seq_no $\leftarrow$ dest_seq_no; lifetime $\leftarrow$ route_lifetime	Has Route
STATE: Has Route		
Condition	**Action**	**Next State**
TimeOut	seq_no $\leftarrow$ seq_no $+ 1$; next_hop $\leftarrow 0$; hop_cnt $\leftarrow 255$ Send to active neighbors: RREP(255,dst, seq_no, BAD_LINK_LIFETIME)	No Route
Recv from p: RREQ(d, hops_to_src, dest_seq_no, s, src_seq_no) $\wedge\ s =$dst $\wedge$ [src_seq_no, hops_to_src] is better than [seq_no, hop_cnt]	next_hop $\leftarrow p$; hop_cnt $\leftarrow$ hops_to_src $+ 1$; seq_no $\leftarrow$ src_seq_no; lifetime $\leftarrow$REV_ROUTE_LIFE	Has Route
Recv from p: RREP(hops_to_dest, d, dest_seq_no, route_lifetime) $\wedge\ d =$dst $\wedge$ [dest_seq_no, hops_to_dest] is better than [seq_no, hop_cnt]	next_hop $\leftarrow p$; hop_cnt $\leftarrow$ hops_to_dest $+ 1$; seq_no $\leftarrow$ dest_seq_no; lifetime $\leftarrow$ route_lifetime	Has Route
Recv from p: RREP(255, d, dest_seq_no, route_lifetime) $\wedge\ d =$dst $\wedge$ dest_seq_no $>$ seq_no	next_hop $\leftarrow 0$; hop_cnt $\leftarrow 255$; seq_no $\leftarrow$ dest_seq_no; lifetime $\leftarrow$BAD_LINK_LIFETIME	Has Route
Recv from p: RREQ(d, hops_to_src, dest_seq_no, s, src_seq_no) $\wedge\ d =$ dst $\wedge$ dest_seq_no $\leq$ seq_no	Unicast from me for s : RREP(hop_cnt, d, seq_no, MY_ROUTE_TIMEOUT)	Has Route
Recv unicast from p for dst: DATA	Send to next_hop : DATA	Has Route
Recv unicast from p for dst: RREP(hops_to_dest $+ 1$, d, dest_seq_no, route_lifetime)	Send to next_hop : RREP(hops_to_dest, d, dest_seq_no, route_lifetime)	Has Route

6.1.3 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 *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 6.2.

Table 6.2 AODV Requirements: Loop Freedom

Property Name	Property Description
Loop Invariant	Along every AODV route to a destination d, the pair $(-\mathsf{seq\_no}_d, \mathsf{hop\_cnt}_d)$ strictly decreases in the lexicographic ordering.

To monitor conformance with the AODV standard, an event recognizer check that the event trace generated by the implementation adheres to the rules of the state machine in Table 6.1.2. Table 6.3 shows some of the properties that test adherence to the state machine. Notice how each property contains an event in its description (denoted by a phrase of the form *when*... or *if*...). We should point out that the set of standard properties listed in Table 6.3 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.

6.2 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 (Table 6.1.2) in NERL. The state of the recognizer

Table 6.3 AODV Specification: Properties from the Standard

Property Name	Property Description
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 hops_to_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 ($\mathsf{seq\_no}_d$).
Detect Route Error	If a node detects a broken route, it should use $\mathsf{dest\_seq\_no} = 1 + (\text{its own})\ \mathsf{seq\_no}_d$ in the unsolicited RREP.
Forward Route Error	When a node forwards an unsolicited RREP, it should forward the same sequence number that it received.

consists of the routing tables at each node. The routes are represented by three arrays:

```
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 $\mathsf{seq\_no}_j$ at node `i`.

We then define state transitions that modify the routing tables by 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:

```
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:

$$\mathsf{next\_hop}_{dst}(at) = nxt.$$

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:

$$(s(at) \leq s(nxt)) \ \wedge \ (s(at) = s(nxt) \ \Rightarrow \ h(at) > h(nxt)).$$

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

$$(s(at) > s(nxt)) \ \vee \ (s(at) = s(nxt) \ \wedge \ h(at) \leq h(nxt)).$$

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`.

```
event (Pkt(p) & Send(s))
        OccurredWhen
            ((next_hop[s.at][s.dst] > 0) &&
```

```
        ((seq_no[s.at][s.dst] >
          seq_no[next_hop[s.at][s.dst]][s.dst]) ||
         ((seq_no[s.at][s.dst] ==
           seq_no[next_hop[s.at][s.dst]][s.dst]) &&
          (hop_cnt[s.at][s.dst] <=
           hop_cnt[next_hop[s.at][s.dst]][s.dst])))) ->
    LoopInvFails(l)
     WithAttributes {
       l.node = s.at;
       l.next = next_hop[s.at][s.dst];
       l.dest = s.dst;
       l.seq_at_node = seq_no[s.at][s.dst];
       l.hc_at_node = hop_cnt[s.at][s.dst];
       l.seq_at_next = seq_no[next_hop[s.at][s.dst]][s.dst];
       l.hc_at_next = hop_cnt[next_hop[s.at][s.dst]][s.dst]
      }
```

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 D gives the complete NERL script for AODV. Table 6.4 shows the AODV properties and their corresponding NERL alarm names.

Table 6.4 NERL Alarms for AODV

Property	NERL alarm
Monotone Sequence Numbers	NotSeqMono
Destination Stops	DestForwards
Correct Forwarding	BadFwd
Destination Reply	BadDestRep
Node Reply	BadNodeRep
RREQ Sequence Number	BadRReq
Detect Route Error	BadRerr
Forward Route Error	BadRerrFwd
Loop Invariant	LoopInvFails

6.2.1 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 a Promela model and analyze it using the SPIN model checker (Hol). 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 and does not produce any false positives:

Property 6.1 *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 6.2 *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.

6.3 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). The code is already instrumented to produce a packet trace for every packet generated, forwarded and dropped by the protocol. We have earlier analyzed and found errors in this implementation using the Verisim tool (BGK+02). In this chapter, we are performing the analysis again to evaluate the effectiveness and efficiency of NERL.

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 6.2.

Topology: There are 5 nodes initially arranged as in Figure 6.2 (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).

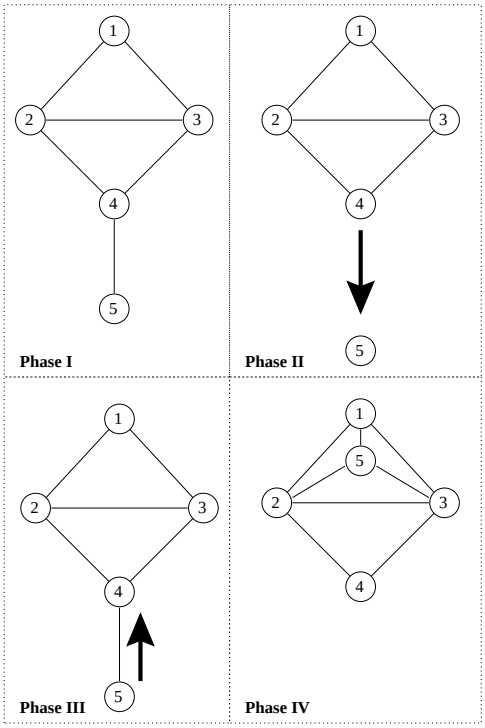

Figure 6.2 Scenario S

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 retried.

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 6.5. When a packet send or receive event happens at a node N, there is a line in the trace with the format:

<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.

6.4 Analysis

We then execute the NERL recognizer on such a trace (T); the recognizer checks if T satisfies the AODV properties ϕ, and produces a meta-trace T^{ϕ} 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. The first row of Table 6.6 shows statistics for the alarms produced by NERL for one of these simulation traces, with 49542 events. (The other rows will be explained later.)

6.4.1 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:

```
19: Pkt Event <eventty:1, pktty: 2, atnode:5, fordest:5, src:5,
               src_seq:2, src_hc:1, dest:3, dest_seq:0, bcastid:0,
               prev:0, next_hop:4, ttl:32, life:600>
19: BadDestRep Event <at:5, dst:5, best_seq:0, best_hc:0, best_next:0>
19: Depends on: <19>
```

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 5 sent an incorrect RREP. Indeed, when we look at the fields we see that SrcHops is set to 1, when it should be 0.

Having found this error, we have two options. We can either inspect the simulator code for AODV, correct this error, and generate a new trace, or we can attempt to find more errors from the same trace. In the following section, we describe a novel technique, called tuning, for the second option. We then return to our simulation analysis and demonstrate its use.

Table 6.5 AODV Simulation Trace T

```
s 0.000000000 _1_ RTR  --- 0 AODV 52 [0 0 0 0 0] -------  [1:255 -1:255 32 0] [0x2 0 1 [5 0] [1 1]] (REQUEST)
s 0.000000000 _2_ RTR  --- 0 AODV 52 [0 0 0 0 0] -------  [2:255 -1:255 32 0] [0x2 0 1 [5 0] [2 1]] (REQUEST)
s 0.000000000 _3_ RTR  --- 0 AODV 52 [0 0 0 0 0] -------  [3:255 -1:255 32 0] [0x2 0 1 [5 0] [3 1]] (REQUEST)
r 0.000519784 _2_ RTR  --- 0 AODV 52 [20 0 ffff 1 800] ---  [1:255 -1:255 32 0] [0x2 0 1 [5 0] [1 1]] (REQUEST)
r 0.000535386 _3_ RTR  --- 0 AODV 52 [20 0 ffff 1 800] ---  [1:255 -1:255 32 0] [0x2 0 1 [5 0] [1 1]] (REQUEST)
r 0.002002991 _1_ RTR  --- 0 AODV 52 [20 0 ffff 3 800] ---  [3:255 -1:255 32 0] [0x2 0 1 [5 0] [3 1]] (REQUEST)
r 0.002006118 _2_ RTR  --- 0 AODV 52 [20 0 ffff 3 800] ---  [3:255 -1:255 32 0] [0x2 0 1 [5 0] [3 1]] (REQUEST)
r 0.002014489 _4_ RTR  --- 0 AODV 52 [20 0 ffff 3 800] ---  [3:255 -1:255 32 0] [0x2 0 1 [5 0] [3 1]] (REQUEST)
s 0.002360210 _4_ RTR  --- 0 AODV 52 [20 0 ffff 3 800] ---  [4:255 -1:255 31 0] [0x2 1 1 [5 0] [3 1]] (REQUEST)
r 0.002689325 _1_ RTR  --- 0 AODV 52 [20 0 ffff 2 800] ---  [2:255 -1:255 32 0] [0x2 0 1 [5 0] [2 1]] (REQUEST)
r 0.002700822 _4_ RTR  --- 0 AODV 52 [20 0 ffff 2 800] ---  [2:255 -1:255 32 0] [0x2 0 1 [5 0] [2 1]] (REQUEST)
r 0.002708053 _3_ RTR  --- 0 AODV 52 [20 0 ffff 2 800] ---  [2:255 -1:255 32 0] [0x2 0 1 [5 0] [2 1]] (REQUEST)
s 0.002777804 _4_ RTR  --- 0 AODV 52 [20 0 ffff 2 800] ---  [4:255 -1:255 31 0] [0x2 1 1 [5 0] [2 1]] (REQUEST)
r 0.003439172 _2_ RTR  --- 0 AODV 52 [20 0 ffff 4 800] ---  [4:255 -1:255 31 0] [0x2 1 1 [5 0] [3 1]] (REQUEST)
r 0.003449342 _5_ RTR  --- 0 AODV 52 [20 0 ffff 4 800] ---  [4:255 -1:255 31 0] [0x2 1 1 [5 0] [3 1]] (REQUEST)
s 0.003449342 _5_ RTR  --- 0 AODV 44 [0 0 0 0 0] -------  [5:255 3:255 32 4] [0x4 1 [5 2] 600] (REPLY)
```

Table 6.6 AODV Property Violations Detected by NERL

Meta-trace	BadDestRep	BadFwd	BadNodeRep	BadRerr	LoopInvFails	Total
T^{ϕ}	3	2807	9	7	4	2830
T^{ϕ_1}	0	2807	9	7	4	2827
T^{ϕ_2}	0	16	9	4	0	29
T^{ϕ_3}	0	16	9	0	0	25

6.4.2 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* (RFB) 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 rerun the implementation and NERL analyses, and repeat. This is clearly inapplicable to live network monitoring. Even in the case of a network simulation, each rerun 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 mimic this incorrect action. We call this modification *tuning*. The tuned NERL recognizer mimics the implementation. We then rerun 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 to be noise generated by the inconsistency between the NERL recognizer and the implementation. To test our guess of where the error is, we modify our recognizer to tune into the implementation state machine and we know that we have succeeded when the noise disappears.

6.4.3 Tuning out BadDestRep Events

Returning to the case study, the BadDestRep errors show that the AODV code for the destination is not implemented correctly. So, let us try to guess the bug in the code.

In all three 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 assume that the destination will always emit a hop-count of 1 (instead of 0). Let us call this new NERL program ϕ_1. When we run this recognizer on the same simulation trace, all 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.

In this way, we have tuned the AODV specification to simulate the error. The idea 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 arriving at the right answer.

6.4.4 Nodes Ignore Hop-count in Routes

The remaining errors are shown in the second row of Table 6.6. 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:

```
82: Pkt Event <eventty:1, pktty: 0, atnode:3, fordest:5, src:3,
               src_seq:0, src_hc:0, dest:5, dest_seq:0, bcastid:0,
               prev:0, next_hop:1, ttl:32, life:0>
82: BadFwd Event <at:3, dst:5, best_seq:2, best_hc:4, bext_next:2>
82: 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 be using node 2. The 'depends on' clause indicates that we should look at the events in packets 80 and 81 to understand the reason. 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 third version of the recognizer ϕ_2. On running ϕ_2 on the simulation trace T, we find that most of the BadFwd errors disappear, validating our guess. Note that the LoopInvFails error has disappeared as well. Error events are often interrelated 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.

6.4.5 *Node Sends Incorrect* RERR

Of the remaining errors in the meta-trace (third row of Table 6.6) the first is a BadRerr event at packet 3592:

```
3592: Pkt Event <eventty:1, pktty: 2, atnode:4, fordest:5,
                src:5, src_seq:3, src_hc:255,
                dest:2, dest_seq:0, bcastid:0,
                prev:0, next_hop:0, ttl:1, life:0>
3592: BadRerr Event <at:4, dst:5, best_seq:3, best_hc:2, bext_next:5>
3592: 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 $(\mathsf{seq\_no} = 3, \mathsf{hop\_cnt} = 1)$. 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 $\mathsf{seq\_no} = 3$, 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 ϕ_3. Indeed, the fourth row of Table 6.6 indicates that all BadRerr error events disappear.

6.4.6 *Nodes Send Expired Routes*

At this point, there are only 25 errors remaining (last row of Table 6.6) 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 events 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 are all the bugs we found.

6.5 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 6.7 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

Table 6.7 FOA outcomes and their interpretations

		Requirements check	
		$T \in R$	$T \notin R$
Conformance check	$T \in S$	E	A
	$T \notin S$	D	C

Region	Interpretation and remedy
E	Everything OK.
D	Incorrect implementation of the standard. Correct the program.
C	Incorrect implementation of the standard. Correct the program.
A	Incorrect standard. Revise the standard and the program accordingly.

for free, in addition to the implementation errors that NERL points out anyway.

6.5.1 FOA for the AODV Simulation Analysis

In section 6.4, when we first analyzed the simulation trace T we generated the LoopInvFails error indicating a violation of the requirements and several 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 6.7).

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.

```
3672: Pkt Event <eventty:1, pktty: 2, atnode:1, fordest:5,
                src:5, src_seq:3, src_hc:4, dest:1, dest_seq:0,
                bcastid:0, prev:1, next_hop:3,
                ttl:32, life:0>
3672: LoopInvFails Event <node:1, next:2, dest:5, seq_at_node:3,
                          hc_at_node:4, seq_at_next:0, hc_at_next:255>
3672: 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 is in category A. It violates the requirements but conforms to the standard, and therefore indicates an error in the AODV standard. It demonstrates that, when routes are deleted too early, even a standard-conformant implementation of AODV may form routing loops.

This flaw in the standard was already known to us, we found it 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.

6.6 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 trace was generated by an AODV simulation on a site of size 1500×300 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 analyzed 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. However, this kind of piecemeal analysis would not work for more complex properties, such as Loop Freedom, where more than 5 nodes may interact to cause an output event.

We reran 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 Dest-

Forwards. 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. There are several reasons for this difference.

1. MEDL was not designed for networking applications, so it lacks arrays and event attributes. Encoding these in MEDL leads to a substantial performance hit, because it causes every event to be encoded as 2500 independent events.

2. MEDL was designed to monitor Java programs. As a result it was implemented in Java, and MEDL monitors expect 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.

These conclusions reinforce our design decision to using a domain-specific language for network monitors.

6.7 Results

Errors. In total, we have detected five 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 sent 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 replies 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 three 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 since been fixed. Our findings raise serious doubts about the reliability of the comparitive study of wireless protocols that was based on this implementation (BMJ+98). It is unlikely that the errors we found had no effect on the performance profile of the protocol. Moreover, it is possible that the other implementations in the study have similar errors that have not yet been discovered.

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 NERL recognizers can be integrated into protocol simulations with little

performance overhead. We argue that all simulator studies should perform a passive analysis of the generated traces to confirm the absence of errors and to improve confidence in their experimental results.

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 7

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`. (Actually, envelopes contain the complete path that the message should take to the recipient, or an error message should take back to the sender. This path includes the email addresses of the sender and receiver.) 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 STMP 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 7.2, 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 7.4, we show how to write the SMTP recognizer in NERL and discuss correctness issues. Finally, we analyze live executions of SMTP servers. Our analysis finds several flaws in popular SMTP mail servers ranging from malformed messages to incorrect transitions in the implemented protocol state machine. We present detailed results in Section 7.5.

7.1 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 7.1 where `C:` indicates client commands, and `S:` indicates server responses.

```
 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 7.1 Sample SMTP Session

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, except during a DATA command, 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 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 her 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.

7.1.1 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 of 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 whom 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 7.1.

The attributes at the beginning of the message make up 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. (The standard (Res01) describes a number of other fields that may contain addressing information as well.) 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).

7.2 Monitoring Environment: Reliable Message Stream

To monitor emails as they are transferred between MTA's we must first reconstruct SMTP commands and responses. This process of layer-wise reconstruction is depicted in Figure 7.2

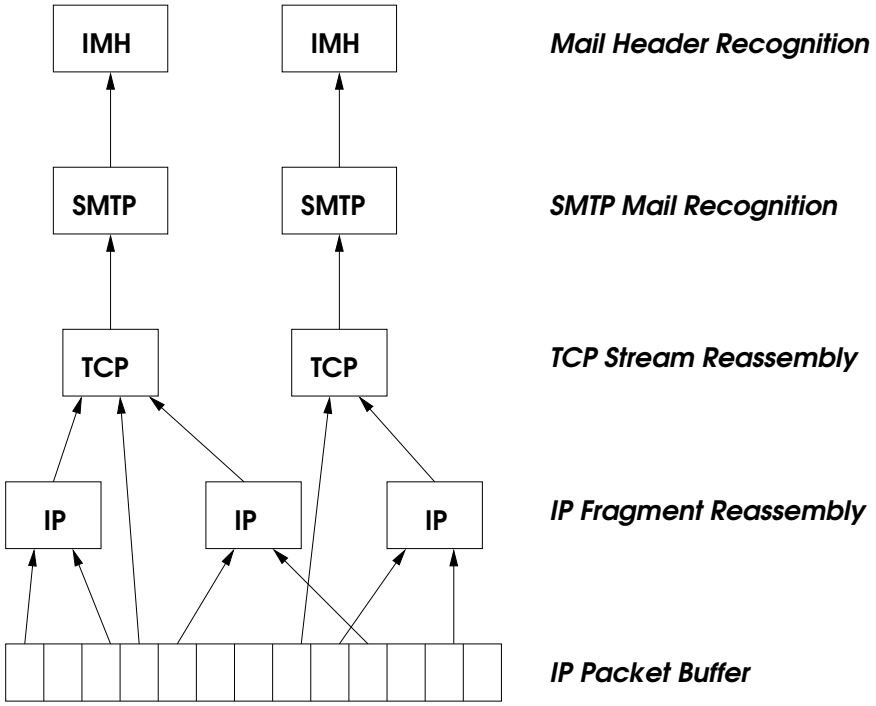

Figure 7.2 Recognizing Emails

The key idea in reconstructing events at each layer is that the receiver of the packets, segments, and commands will have to carry out this reconstruction anyway. So, all we need to do is to mimic the actions of the receiver. For this case study, we assume that the imple-

mentations of all the protocol layers below SMTP, namely the IP and TCP implementations, are correct at both the sender and receiver. Checking properties of these layers is beyond our scope. With this assumption, we can reconstruct a high-level command-response trace that is a *true* trace, even if the packet trace on the wire was not. To see why, observe that TCP reliability implies that, for every data segment S sent in a TCP session, there is a time T when both the sender and receiver agree that the data segment S has been successfully sent and received. So, by mimicking the operation of both the sender and the receiver, the monitor can conclude at time T that S was sent and S was received at some time before T. This information is enough to accurately reconstruct the events at the SMTP layer.

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

7.2.1 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 that takes `IP_Frag` input events and outputs complete `IP_Pkt` events. One instance of `IP_Reassemb` is generated for

every new <ip_src,ip_dst,ip_id,ip_p> tuple seen on a fragmented packet. The reassembly operation itself is encoded according to a simple algorithm. If efficiency of reassembly becomes an issue, we can use more sophisticated algorithms (Bra89a).

The key fragment of NERL code is shown in Appendix E.1. When a fragment is received, it is first inserted into the packet buffer `ipdatabuf`. The range of bytes it occupies is marked in `ipdatarecd`. If it is the last fragment, then we know what the length of the entire packet (`ip_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 different byte ranges that overlap.

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 IP 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. This is reasonable because fragmentation errors are rare in modern networks and only appear in unusual circumstances such as network intrusions.

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 7.1 *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. 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 this restricted system and shows that for all executions of the system the property holds true.

7.2.2 TCP Stream Reassembly

We shall describe the TCP protocol in more detail in Chapter 8. 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 implements a reliable stream. 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_Reassemb` recognizer is included as Appendix E.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 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. As with fragmentation, TCP errors are rare in modern networks except when attackers use such packets to bypass intrusion detection systems (PN98).

Recognizer Correctness

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

Property 7.2 *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_Reassemb` 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.

7.3 SMTP Main Module

Having written recognizers for IP fragments and TCP streams, we can now set up the entire monitoring stack for SMTP executions (Figure 7.3). 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.

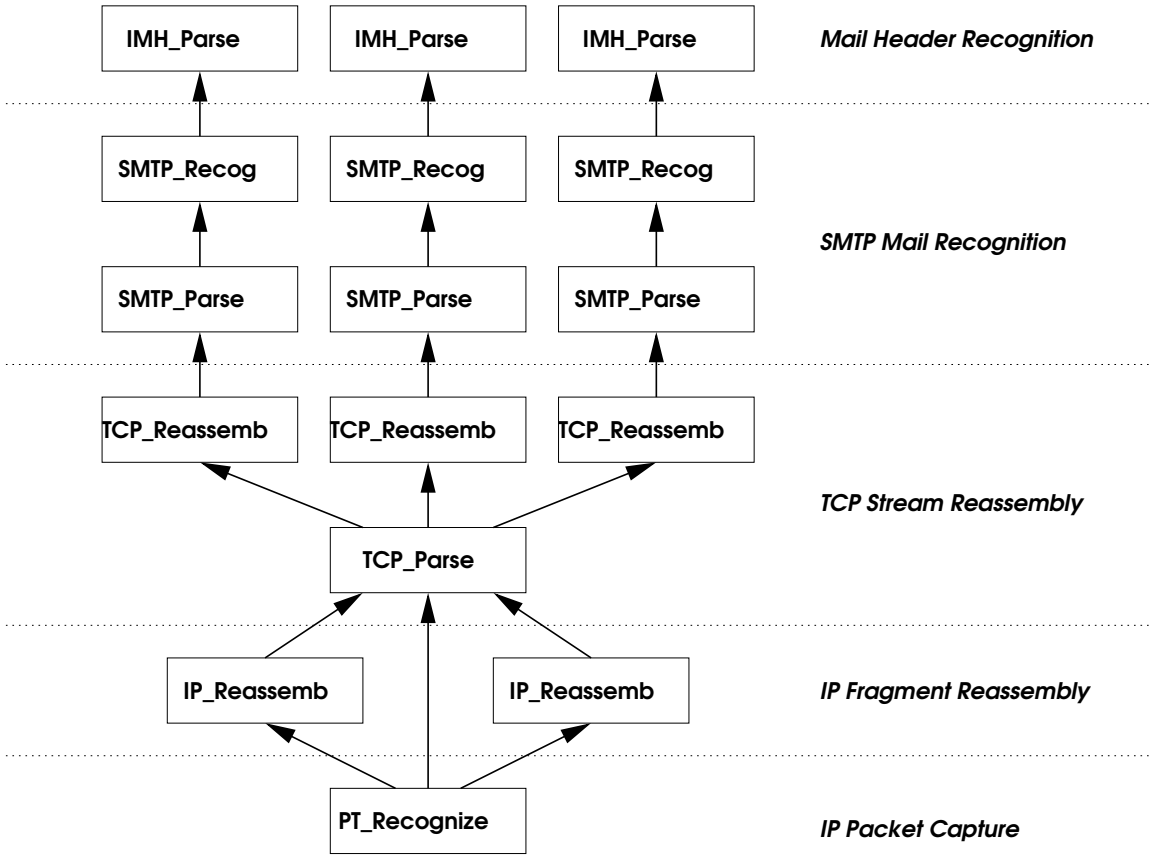

Figure 7.3 Recognizing Emails

- 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 instance of `TCP_Reassemb` 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 reconstructed `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 recognizers are written, it is straightforward to write a main module that models this stack.

7.4 The SMTP Recognizer in NERL

The complete SMTP server state machine for the minimal command set is shown in Figure 7.4. 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 or 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 RESPNONE, RESPOK or RESPDONE, 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 RESPNONE. The recognizer then waits for the server response and then commits the state transition or rolls back to `laststatus`.

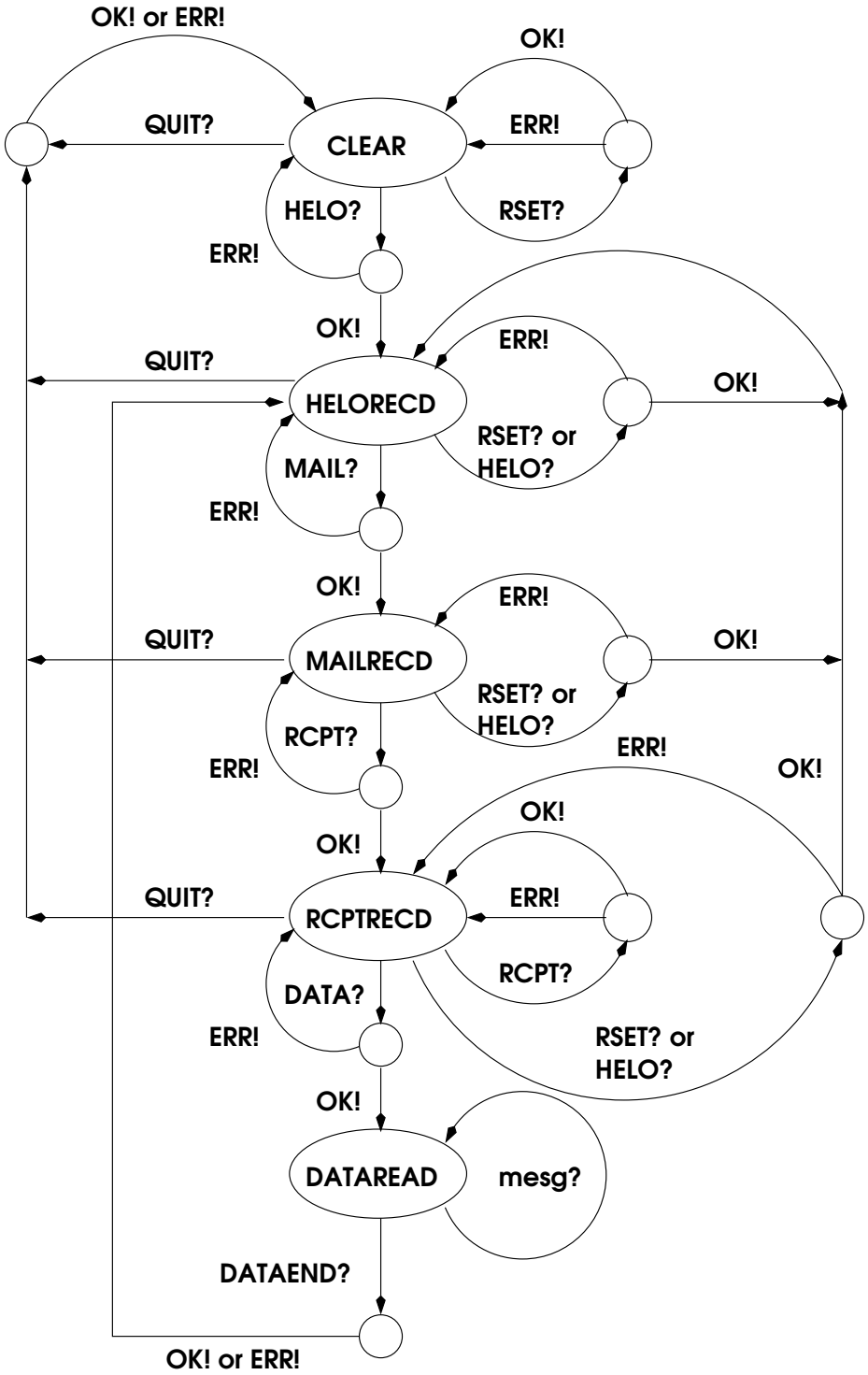

Figure 7.4 SMTP Server State Machine

Table 7.1 Events and State Transitions for the MAIL Command

```
transition m: MailFrom(m)
              OccurredWhen
                ((status == HELLO) &&
                 (respstatus == RESPDONE)) -> {laststatus = status;
                                               status = MAIL;
                                               respstatus = RESPNONE;
                                               senderseen.user = m.user;
                                               senderseen.domain = m.domain;
                                              };

event MailFrom(h)
      OccurredWhen
       !((status == HELLO) &&
         (respstatus == RESPDONE)) -> Command_Error(b)
                                      WithAttributes
                                      {b.error = "Unexpected MailFrom";
                                       b.status = status;
                                       b.respstatus = respstatus;};

transition mr2: MetaResponse(m)
                OccurredWhen ((status == MAIL) &&
                              (m.ok == true)) -> {
                                sender.user = senderseen.user;
                                sender.domain = senderseen.domain;
                                };
transition mr1: MetaResponse(m)
                OccurredWhen ((status == MAIL) &&
                              (m.ok == false)) -> {status = laststatus;};

event MetaResponse(m)
      OccurredWhen (m.ok == false) ->    Response_Error(b)
                                          WithAttributes
                                          {b.error = "Neg Resp.";
                                           b.status = status;};
```

For instance, consider the MAIL command. Its transitions are converted to the NERL code in Table 7.4. Here the state transitions are as shown in Figure 7.4. 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 (`MetaResponse(m)` with `m.ok == true`), then the transition is committed and the sender email address is confirmed. On the other hand, if the server sent a negative reply (`m.ok == false`), 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 E.3.

7.4.1 Recognizer Correctness

One aspect of programming the event recognizer that we have not detailed is the extraction of the state machine from the protocol standard. Depending on our focus, we can choose a different subset of commands and states from the same protocol specification. 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. Our NERL recognizer for SMTP is primarily concerned with server errors and so implements the state machine in Figure 7.4 that we claim is complete for this purpose.

How do we know that our monitor is adequate for our purposes? Would the Altivore or BlackICE monitor work just as well? In order to check the correctness of server software, we want the SMTP monitor to obey the following property:

Property 7.3 *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 7.4 *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 can 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 first-in-first-out because the channel is implemented by TCP.

We 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 encode this monitor in NERL as Recognizer A. Network intrusion detection systems, such as BlackICE, implement SMTP monitors as well (Gro01). We code one such monitor in NERL as Recognizer N, which contains a more sophisticated stateful analysis like the one a NIDS might use. Finally, we encode our own SMTP recognizer that follows the complete SMTP state machine as SMTP_Recog. We translate these three NERL recognizers to Promela and analyze them using the SPIN model-checker.

Recognizer A fails and SPIN produces the counter-example depicted in Figure 7.5. Each vertical line represents one process. The process on the left is the SMTP client, the process on the right is the SMTP server, and the process in the middle is the SMTP monitor A. Every message from the client to the server passes through the monitor. The monitor estimates the appropriate server response and publishes it. SPIN checks that the actual server response always matches the monitor's estimate. The message sequence in Figure 7.5 depicts an error found by SPIN. The monitor model generated by A does not correctly handle the case when the client issues two `MailFrom` commands and the first is rejected. It should treat the second `MailFrom` command as initiating a new email. Instead, it captures the second email as part of the text 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 attempt the same proof for recognizer N. Again, SPIN finds an error in the monitor model and produces the counter-example depicted in Figure 7.6. When a `Data` command results in an error response from the server, the monitor should return to the previous state and wait for a new `Data` command. Instead, the model generated from N fails to notice this

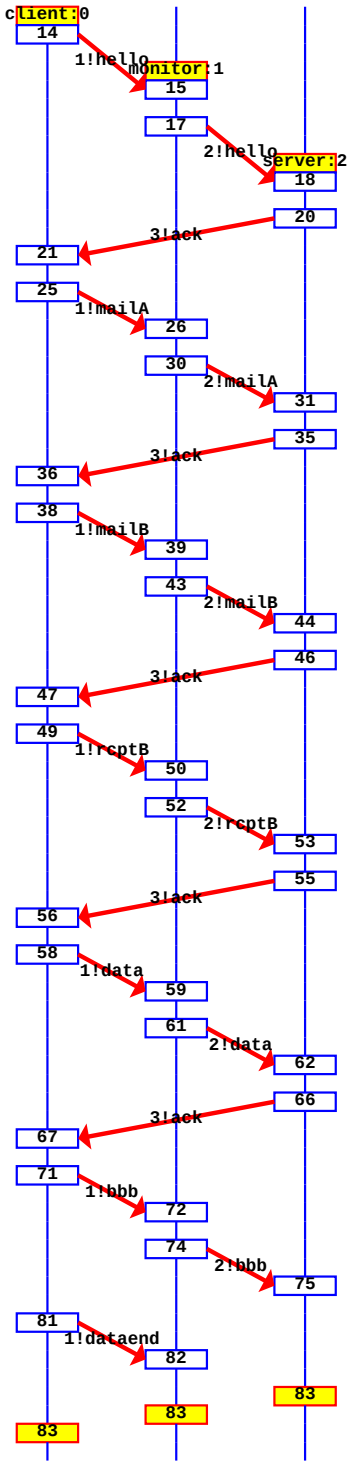

Figure 7.5 SPIN Counter-example for SMTP Recognizer A

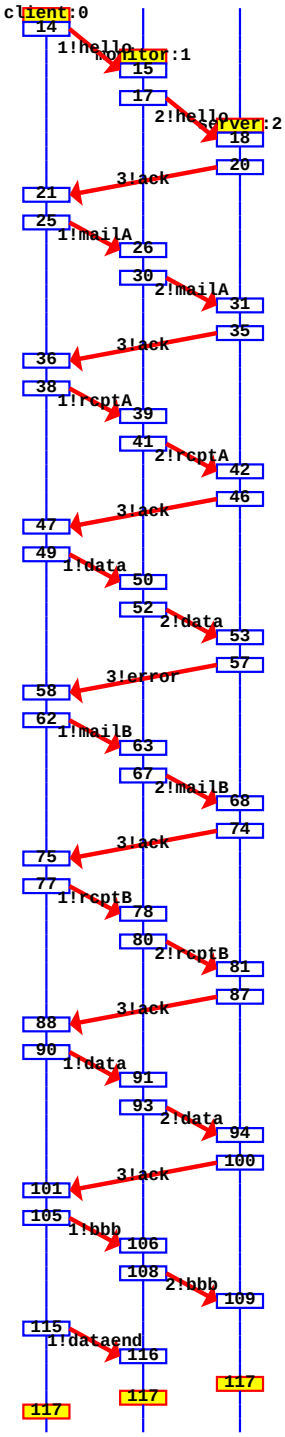

Figure 7.6 SPIN Counter-example for SMTP Recognizer N

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.

The SMTP monitors in Altivore or BlackICE are not necessarily incorrect. These systems have a different focus from ours and it would have been surprising if their monitors were adequate for our purposes. Instead, the above analyses indicate that interpreting the standard and extracting the right state machine for a specific purpose can be tricky and that model-checking can help the programmer to debug his monitor program. Indeed, as we shall see in the following sections, interpreting the SMTP state machine correctly has proved to be surprisingly difficult for implementors and there are several ways to get it wrong.

7.5 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 all the 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 deliver 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 the server either 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 7.5. 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 `Envelope_Sent`. 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 `Envelope_Accepted` event. The client then sends an email in packet 34, which is then acknowledged in packet 35 - triggering an `Mail_Sent` event followed by an `Mail_Accepted` event. The email is parsed according to the Internet Message Format

Table 7.2 Output Trace of the Mail Monitoring Stack

```
25: Envelope_Sent Event <>
25: Depends on: <14,15,16,18,19,20,22,23,24,25>
26: Response_Error Event <error: ``Neg Resp'', status:RCPT>
26: Depends on: <14,15,16,18,19,20,22,23,24,25,26>
27: Envelope_Sent Event <>
27: Depends on: <14,15,16,18,19,20,22,23,24,25,27>
28: Envelope_Accepted Event <>
28: Depends on: <14,15,16,18,19,20,22,23,24,25,26,27,28>
28: Data Depends on: <23,25,27>
28: ---------Acked Envelope Follows-------
28: Sender: bkarthik@verinet.cis.upenn.edu
28: Receiver: bkarthik@seas.upenn.edu
28: ---------Acked Envelope End-------
34: Mail_Sent Event <>
35: Depends on: <14,15,16,18,19,20,22,23,24,25,26,27,28,29,30,31,34>
35: Data Depends on: <23,25,27,29,31,34>
35: Mail_Accepted Event <>
35: Depends on: <14,15,16,18,19,20,22,23,24,25,26,27,28,29,30,31,34,35>
35: Data Depends on: <23,25,27,29,31,34>
35: ---------Acked Email Follows-------
35: Sender: bkarthik@verinet.cis.upenn.edu
35: Receiver: bkarthik@seas.upenn.edu
35: Text:
35: Received: from verinet.cis.upenn.edu (verinet.cis.upenn.edu [158.130.13.33])
35:     by verinet.cis.upenn.edu (8.11.6/8.11.6) with ESMTP id g2ELgTA05682;
35:     Thu, 14 Mar 2002 16:42:29 -0500
35: Date: Thu, 14 Mar 2002 16:42:29 -0500 (EST)
35: From: Karthik <bkarthik@verinet.cis.upenn.edu>
35: To: bkarthik@seas.upenn.edu
35: cc: bkats@seas.upenn.edu
35: Message-ID: <Pine.LNX.4.44.0203141642070.5680-100000@verinet.cis.upenn.edu>
35: MIME-Version: 1.0
35: Content-Type: TEXT/PLAIN; charset=US-ASCII
35: --
35: ----------------------------------------
35: http://www.seas.upenn.edu/~bkarthik
35: ----------------------------------------
35: .
35:
35: ---------Acked Email End-------
35: --------- Parsed Internet Message Format---------
35: Date: Thu, 14 Mar 2002 16:42:29 -0500
35: From: bkarthik@verinet.cis.upenn.edu
35: To: bkarthik@seas.upenn.edu
35: Cc: bkats@seas.upenn.edu
35: Msg-ID: Pine.LNX.4.44.0203141642070.5680-100000@verinet.cis.upenn.edu
35: --------- Parsed Internet Message Format Ends---------
```

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 `Envelope_Accepted` 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. When only the current packet contributed to an event, we do not print the data dependencies.

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.

7.5.1 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 packets), the first response error is on packet no 14

```
14: Response_Error <error:``Neg Resp.'', status:EXPAND>
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 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. Although this is a violation of the standard, we consider it to be a minor one.

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: Response_Error <error:``Neg Resp.'', status:HELLO>
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 recom-

mended that servers reject them. However, the new version of SMTP (since 2001) explicitly allows multiple HELO's.

This error was independently discovered as a bug in Sendmail and has been corrected in the latest version (8.12) of the Sendmail 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 unparseable. On closer inspection, all the badly formatted responses were of the same kind. For instance, the first such error event says

```
23: 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>
```

Packet no 23 contained the data

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

This response was unparseable, 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 introduced in Section 6.4.2, 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 events 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
47: Response_Error <error:``Neg Resp.'', status:MAIL>
```

The error statment says that a HELO is required before sending a MAIL, but the event indicates that the current state is MAIL, 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
1592: Response_Error <error:``Neg Resp.'',status:DATA>
```

Here the response claims that no MAIL command was sent while the error event indicates that the current state is DATA, 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.

7.5.2 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 issuinng 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: Response_Error <error:``Junk Resp is OK.'', status:JUNK>
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 command 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 mechanisms to increase the security of the server. However, as we found, the `needmalhelo` flag is in fact 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: Response_Error <error: ''Junk Resp is OK.'', status: JUNK>
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 (For brevity, we have not shown the intermediate commands that are not significant to the state machine.)

```
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
```

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 resets 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

Table 7.3 Time and memory usage for Expect trace analysis

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

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 received, 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.

7.6 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 7.6 we present our results for several large SMTP traces with full 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 7.6, 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

Table 7.4 Time and memory usage for Multimail trace analysis

Trace	Packets	Sessions	Emails	Execution(s)	Analysis(s)	Memory(MB)	Output(MB)
4	12960	486	486	50.5	1.78	5.83	6.81
5	21261	600	600	136.06	2.53	7.4	9.53
6	28793	1000	1000	71.28	3.7	12	14.88

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 traces 4, 5, and 6 in Table 7.6. Notice that for these traces the period of execution is quite small - the Multimail clients are flooding 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.

7.7 Results

Errors. 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 found 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.

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. Its 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. 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 NERL 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 main module successfully combines recognizers written in NERL, Lex and C to execute the SMTP monitor.

Correctness. We demonstrate the model-checking procedure in detail for the SMTP recognizer. We show that some versions of the recognizer, inspired by email surveillance systems and intrusion detection systems, can be proved to be inadequate for testing SMTP servers. We find no errors in our SMTP recognizer.

CHAPTER 8

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, and 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 monitor protocols when the monitor is co-located with the device, 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 demonstrate how TCP implementations can be monitored 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 because 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 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 2.4.4). 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, he found errors in all major operating system implementations (Pax97). However, this work failed 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 one choosing a different 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 5. 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 environment. We analyze TCP implementations of well-established operating system implementations, and, unsurprisingly, do not find any significant errors. However, we demonstrate that offline traces of TCP-like transport protocols can be feasibly checked by NERL programs, and we evaluate the effectiveness of co-networked monitoring for different safety properties.

8.1 Transmission Control Protocol

TCP has been described in 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. Our TCP state machine is shown in Figure 8.1.

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 an OPEN command with a local port, remote host and remote port number. TCP then sets up a socket that connects `<local_host, local_port>` with `<remote_host, remote_port>`. This initial setup is achieved by a handshake consisting of a SYN packet to, a SYNACK packet back and an ACK packet to the remote host.

Once the socket is open, the application layer protocol simply writes data onto it using the SEND command, while the remote host reads data using the RECEIVE command. These are treated like file write and read at each host, while TCP carries out the actual data transfer using DATA packets. TCP receivers acknowledge receipt of DATA, FIN and SYNACK packets by sending ACK packets.

TCP allows data transfer in both directions - 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 transferred only in one direction, from client to server, the client needs to wait for a server OK after almost every line that 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 have 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 8.1 contains the *observable* part of the state machine: TCP states and the 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.

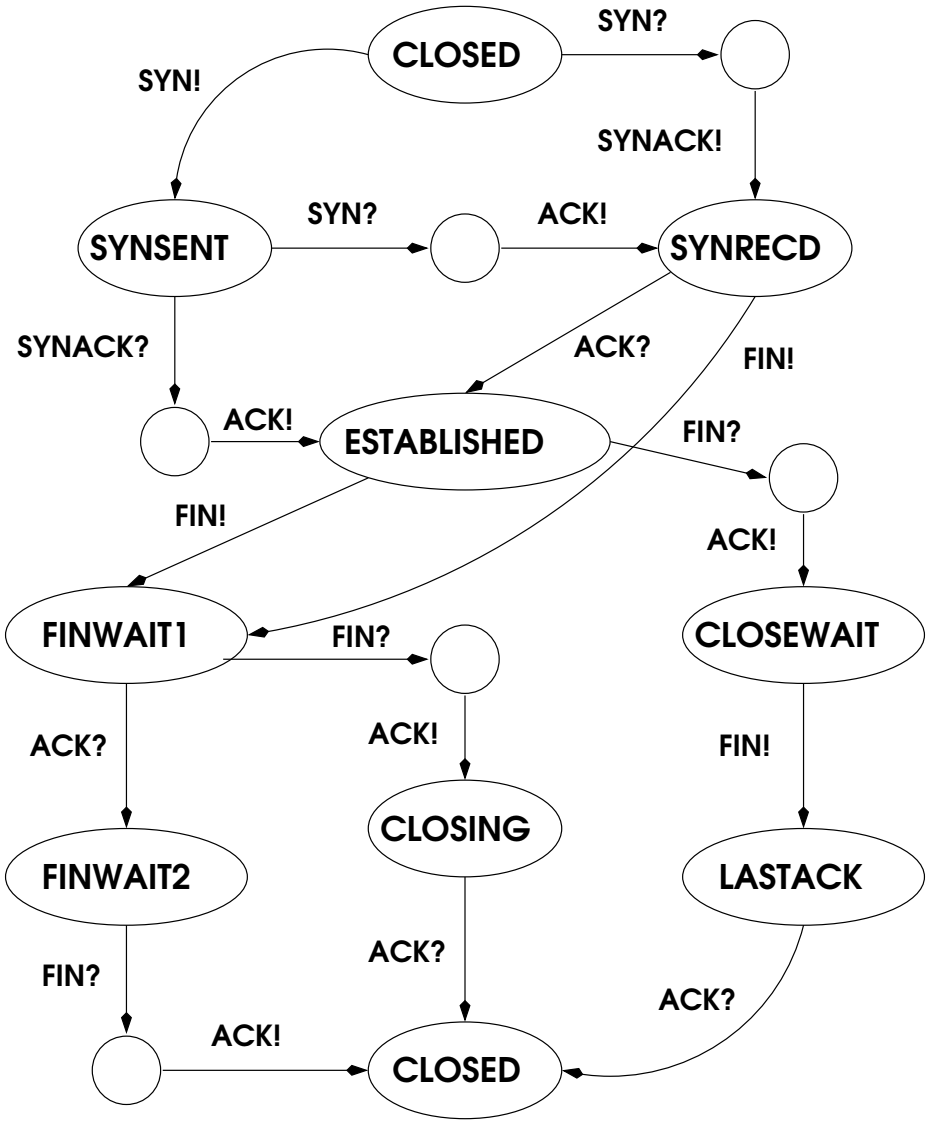

Figure 8.1 TCP State Machine

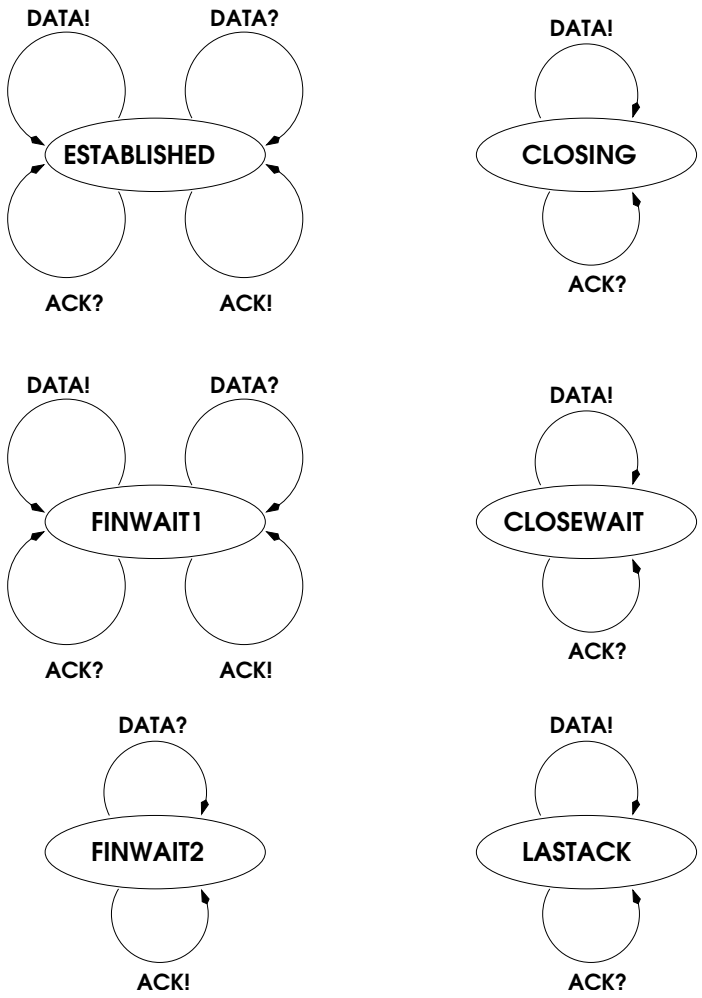

Figure 8.2 TCP Data Transfer States

8.1.1 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 8.2 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 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, whereas 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 $S_{\max}$, the receiver has received contiguous data up to sequence number S_{Cont}, and the last acknowledgment it sent had sequence number A_L; then the next acknowledgment A_N produced by the receiver must follow

$$A_N = S_{Cont} + 1.$$

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

$$A_N \geq A_L.$$

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

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

When the sender sends the next segment with sequence number S_N and length L, this

segment must obey the limitations of the receivers' window, based on the last ACK A_L it received:

$$A_L \leq S_N < A_L + W$$

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

The receiver accepts this segment it it satisfies its window:

$$S_{Cont} + 1 \leq S_N \leq S_{Cont} + W$$

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

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.

8.2 TCP Properties in NERL

Given the TCP state machine in Figure 8.1, we write a NERL recognizer to check deviations from it. Encoding the state machine is straightforward - the recognizer maintains a variable - `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 F.

8.3 Initial Analysis

To start with, let us ignore may 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 as depicted in the following trace. For readability, in all the following traces, we shall replace IP addresses by names and only show the relevant TCP fields.

```
62: TCP Event <ip_src:verinet, ip_dst:saul,
               tcp_sport:44878, tcp_dport:25,
               tcp_seqno:188034011, tcp_ackno:188033049>
62: TCP_Error Event <``10:ACK sent with bad ackno''>
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: TCP Event <ip_src:verinet, ip_dst:saul,
               tcp_sport:44878, tcp_dport:25,
               tcp_seqno:188033049, tcp_ackno:37513231>
61: Depends on:<61>
```

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 next packet (63) is an ACK sent by the receiver:

```
63: TCP Event <ip_src:saul, ip_dst:verinet,
               tcp_sport:25, tcp_dport:44878,
               tcp_seqno:37513231, tcp_ackno:188034012>
```

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. On almost all of the SMTP traces we analyze, our NERL recognizer generates error events of the following types:

- 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 will model the IP buffer and distinguish between these cases.

8.4 Monitoring Environment: Co-networked Unreliable Channel

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 8.3) actually monitors 4 channels

1. The IP packets sent from A that are seen at M: the output channel at A

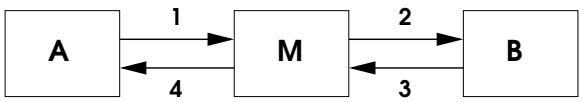

Figure 8.3 TCP Monitor Channels

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 always translate to an immediate 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.

A NERL programmer can deal with this non-deterministic channel behavior in two ways. The NERL compiler includes a special feature called a *channel tranformation* that uses annotations in the signature of a recognizer to automatically generate code for searching through all valid traces corresponding to an observed packet trace. This search corresponds to the trace search algorithm developed by Bhargavan *et al.* (BCMG01) (reproduced in Appendix B.

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 sits on IP, or NFS 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.

8.5 Trace Search for TCP

To apply the trace search algorithm to the TCP recognizer, let us first look at its signature. The original signature has a single TCP input event that represents packets travelling in both directions and assumes that each packet is sent and instantaneously received.

```
input event ty_tcpflow Init;
input event ty_tcppkt TCP;

output event bool Done;
output event ty_tcpdata TCP_Data;
output event string TCP_Error;
```

Assuming that two TCP participants are A and B. Then, the TCP event above actually consolidates four separate events: (1) packet sent from A, (2) packet received at B, (3) packet sent from B, and (4) packet received at A. And each of these packet events occurs on a separate monitoring channel; the first and third are on output channels and the second and fourth are on input channels. So, we define four channels, and modify the signature as 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.

```
ichannel ITo[5,0]
ichannel IFrom[5,0]
ochannel OTo
ochannel 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 TCP_Data_To;
output event ty_tcpdata TCP_Data_From;
output event string TCP_Error;
```

The trace search algorithm outlined in Chapter 5 only works for one input channel and one output channel. To use this algorithm, we should split the signature and the TCP recognizer into two halves, one for each protocol participant. This results in a slightly inefficient configuration where every session is monitored by two separate instances, one for the sender and one for the receiver. However, when the two input channels are on two different nodes, as is the case here, their buffers are independent of each other. In this case, the trace-search algorithm can be modified slightly to model both buffers at the same time. In the rest of this section, we will use this optimized version of the trace-search algorithm.

We use the channel transformation mode of the NERL compiler to generate a co-networked TCP monitor. The monitor models 2 input buffers for each TCP connection to generate all plausible sequences of TCPSent and TCPRecd events. It then checks each sequence against the original TCP recognizer. 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.

8.6 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:

```
62: Possible TCP_Error Event <``5b:ACK sent with bad ack''>
62: Depends on:<45,46,47,49,50,51,52,53,54,55,56,57,58,59>
```

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 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:

```
449: Possible TCP_Error Event <``2a:Syn sent in bad state''>
449: Depends on:<>
449: Possible TCP_Error Event <``2a:Syn sent in bad state''>
449: Depends on:<448>
449: Definite TCP_Error Event
```

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:

```
ichannel lTo[5,1]
ichannel lFrom[5,1]
```

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 major 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.

8.7 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-nco-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) of 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: Definite TCP_Error Event
```

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: Possible TCP_Error Event <``11c:Too many unacked packets''>
```

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* (Section 6.4.2), 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.

This phenomenon of delaying ACKs has been observed earlier (Pax97) for several operating systems. Paxson argues that although delaying ACKs not considered to be a serious violation of the standard, it is network-unfriendly and results in provably sub-optimal performance when packets are being lost by the network.

Our study demonstrates the usefulness of packet buffer modeling. For instance, let us look at a fragment of the TCP trace of a Linux implementation

```
57: TCP Event <ip_src:verinet, ip_dst:bangalore,
               tcp_sport:36188, tcp_dport:5001,
               tcp_seqno3257542433, tcp_ackno:433220911>
---------------------------------------------------------
58: TCP Event <ip_src:verinet, ip_dst:bangalore,
                tcp_sport:36188, tcp_dport:5001,
                tcp_seqno3257543881, tcp_ackno:433220911>
---------------------------------------------------------
59: TCP Event <ip_src:verinet, ip_dst:bangalore,
               tcp_sport:36188, tcp_dport:5001,
               tcp_seqno3257545329, tcp_ackno:433220911>
---------------------------------------------------------
60: TCP Event <ip_src:verinet, ip_dst:bangalore,
               tcp_sport:36188, tcp_dport:5001,
               tcp_seqno3257546777, tcp_ackno:433220911>
---------------------------------------------------------
61: TCP Event <ip_src:bangalore, ip_dst:verinet,
               tcp_sport:5001, tcp_dport:36188,
               tcp_seqno433220911, tcp_ackno:3257522161>
```

Here the Linux TCP receiver (bangalore) 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

```
11: TCP Event <ip_src:verinet, ip_dst:bangalore,
               tcp_sport:36206, tcp_dport:5001,
               tcp_seqno62127937, tcp_ackno:3827060927>
---------------------------------------------------------
12: TCP Event <ip_src:verinet, ip_dst:bangalore,
               tcp_sport:36206, tcp_dport:5001,
               tcp_seqno62129385, tcp_ackno:3827060927>
---------------------------------------------------------
13: TCP Event <ip_src:verinet, ip_dst:bangalore,
               tcp_sport:36206, tcp_dport:5001
               tcp_seqno62130833, tcp_ackno:3827060927
---------------------------------------------------------
14: TCP Event <ip_src:verinet, ip_dst:bangalore,
               tcp_sport:36206, tcp_dport:5001,
               tcp_seqno62132281, tcp_ackno:3827060927>
---------------------------------------------------------
15: TCP Event <ip_src:bangalore, ip_dst:verinet,
               tcp_sport:5001, tcp_dport:36206,
               tcp_seqno3827060927, tcp_ackno:62133729>
```

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.

8.8 Scalability

The analysis times for the traces described in the previous section are shown in Table 8.8. 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 8.8, 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 dissertation. 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.

Table 8.1 Time and memory usage for TCP trace analysis

Trace	OS	Packets	Buffer	Unacked	Execution(s)	Analysis(s)	Memory(MB)
1	Linux	4930	5	2	47.46	11.6	22.23
2	Linux	9804	5	2	95.36	23.21	21.89
3	Win XP	4752	5	5	55.2	12.38	14.62
4	Win XP	9474	5	10	113.96	25.23	13.16
5	Solaris	4930	5	5	47.46	13.92	18.11
6	Solaris	9814	5	5	95.36	27.75	21.64

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 initial handshake, 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 ($\Gamma \times 2^B$). In effect, the states in an finite state machine define an equivalence relation on the set of plausible instances.

Responsiveness A protocol is said to be responsive if a participant must produce an output for every C inputs received. For instance, for TCP $C = 2$, and for SMTP $C = 1$ (since SMTP servers must respond to every input).

Responsive protocols limit the number of buffering cases that need to be checked. For instance, if N SMTP commands are seen in a row, since at most 1 of them may be consumed, there are only two cases to consider: either all N messages are buffered or

the last $N-1$ are bufferred. Without the responsiveness assumption, we may have to consider all N cases.

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.

8.9 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 to speed up the monitoring of the TCP properties we have described earlier.

8.9.1 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 8.9.1). Both the memory usage and analysis time are halved when we eliminate this one error check.

Table 8.2 Time and memory usage for Remaining TCP Properties

Trace	OS	Packets	Buffer	Execution(s)	Analysis(s)	Memory(MB)
1	Linux	4930	5	47.46	5.51	12.79
2	Linux	9804	5	95.36	11.23	12.74
3	Win XP	4752	5	55.2	5.84	4.29
4	Win XP	9474	5	113.96	11.79	4.29
5	Solaris	4930	5	47.46	6.15	5.85
6	Solaris	9814	5	95.36	12.19	5.85

Table 8.3 Time and memory usage for TCP Monotonic ACK property

Trace	OS	Packets	Buffer	Execution(s)	Analysis(s)	Memory(MB)	Result
1	Linux	4930	5	47.46	0.23	0.071	Pass
2	Linux	9804	5	95.36	0.46	0.071	Pass
3	Win XP	4752	5	55.2	0.22	0.071	Pass
4	Win XP	9474	5	113.96	0.42	0.071	Pass
5	Solaris	4930	5	47.46	0.23	0.071	Pass
6	Solaris	9814	5	95.36	0.45	0.071	Pass

8.9.2 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 collected traces 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 8.9.2. Notice the low analysis time and memory usage.

When we first analyze a protocol, we always assume the absence of the input buffer. Only if the implementation fails this analysis do we assume input buffering and loss. This is safe because, if the co-located monitor does not produce an error then the co-networked monitor will not produce an error. (For a proof, see Lemmas B.5 and B.11 in Appendix B.)

8.9.3 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:

Property 8.1 (TCP Counting) *The receiver generates an ACK for at least every other DATA packet it receives.*

This property falls in a class that we call P1:*Counting properties*. Such properties do not look at the content of packets; they only count the number of inputs and outputs. A general co-networked monitoring algorithm for counting properties is presented in Appendix B. In Table 8.9.3, we describe an instantiation of this algorithm for the TCP counting property. Two parameters of the algorithm are the constants $c_{\min} = 0, c_{\max} = 2$ representing the minimum and maximum number of DATA packets allowed between ACKs. In particular, $c_{\max}$ is the same as Unacked in Table analyses. A third parameter is the buffer size B. This algorithm maintains two integers, $buf_{\min}$ and $buf_{\max}$, representing the minimum and maximum number of DATA packets that are currently buffered on the channel between the monitor and the device under test. If $buf_{\min}$ ever grows too large, it indicates that the DATA and ACK sequence seen so far cannot reflect a valid execution without additional buffering between the monitor and the device. That is, too few ACKs have been seen to account for all the DATAs seen so far. Similarly, if $buf_{\max}$ ever becomes too small, it indicates that the DATA and ACK sequence seen so far cannot reflect a valid execution because even if each ACK has consumed the maximum number of DATAs, there have not been sufficient DATAs to account for every ACK. In each case, an error flag e is set.

When we use this algorithm to check the TCP traces generated before, we find that the

Table 8.4 Co-networked Monitor Algorithm for TCP Counting

Constants. $c_{\min} = 0, c_{\max} = 2$ are integers representing the minimum and maximum number of DATA packets allowed between ACKs.
B is an integer representing the estimated size of the input buffer.

Data Type. $buf_{\min}$ and $buf_{\max}$ are integers estimating the minimum and maximum number of packets in the input buffer.
e is a boolean representing the error state.
Initially, $buf_{\min} = buf_{\max} = 0$ and e = false.

Event Handlers. On receiving

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

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

Table 8.5 Time and memory usage for TCP Counting property

Trace	OS	Pkts	B	$c_{\max}$	Execution(s)	Analysis(s)	Memory(MB)	Result
1	Linux	4930	5	2	47.46	0.19	0.071	Pass
2	Linux	9804	5	2	95.36	0.51	0.071	Pass
3	Win XP	4752	5	2	55.2	0.22	0.071	Fail
4	Win XP	9474	5	2	113.96	0.37	0.071	Fail
5	Solaris	4930	5	2	47.46	0.2	0.071	Pass
6	Solaris	9814	5	2	95.36	0.41	0.071	Fail

performance has improved by several orders of magnitude. Table 8.9.3 contains the time and memory usage for checking the TCP counting property.

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 DATA packets, so it allows for both possibilities. This means that our specialized algorithm misses some errors that would be caught by a more complete recognizer.

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 (BCMG01) and aim to automate these strategies in a future version of NERL.

8.10 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.

In an earlier passive testing study, Paxson observed this delayed ACK behavior for the Solaris and IRIX implementations of TCP (Pax97). Paxson calls these *strech acks* and explains that they have several drawbacks in wide-area networks. TCP connections with stretch acks are brittle with respect to packet loss and are provably sub-optimal in transmitting data (Pax97). Instead of sending a steady stream of data, the sender ends up sending a burst of packets and waits for an ACK

It is interesting to note that the passive testing tool, Tcpanaly, that Paxson used for

his study does not take packet buffers into account. As a result, the tool found stretch acks in almost all the TCP implementations, many of them false positives. To account for buffering, Paxson manually discounted the results where the ACKs spanned over only a few DATA packets. He found that Solaris and IRIX consumed so many DATA packets before an ACK that they could not be ignored. Our co-networked monitor performs this analysis automatically and provably produces errors only when a real stretch ack has been sent.

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.

Monitoring with Indifelity. We demonstrate that monitors written in NERL can be effectively used for co-networked monitoring. We apply channel transformations on a TCP monitor written in NERL and automatically generate a co-networked monitor from co-located one. This emphasizes the value of using a domain-specific language for monitoring; it enables a structured transformation of the execution semantics.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

In this thesis, we presented and evaluated the design and implementation of a programmable passive protocol monitoring framework called Network Event Recognition. We presented a domain-specific language, NERL, for writing protocol monitors. We described several analysis tools and techniques and implemented them for NERL programs. Using NERL monitors, we analyzed protocol implementations of well-known protocols. In this chapter, we evaluate our results, draw general conclusions, and speculate on future work.

9.1 Evaluating NERL

The primary measure of effectiveness for a protocol testing framework is the ability to find errors in implementations. We used NERL monitors to analyze implementations of three protocols and in all three case studies, we found implementation errors. We found five different flaws in the simulator implementation of AODV. In addition, we found one flaw in the AODV (version 0) standard. We found six different flaws in the SMTP implementations of three popular mail servers. We confirmed the existence of one flaw in two TCP implementations and its absence in a third implementation. We conclude that network event recognition is an effective protocol testing framework.

To further evaluate the effectiveness of the NERL language and its analysis tools, we return to the evaluation criteria set out in Chapter 3.

Expressiveness. We showed that NERL can be used to program monitors for a wide variety of network protocols. We wrote event recognizers for AODV monitoring, IP fragment reassembly, TCP stream reassembly, SMTP monitoring, and TCP state machine and reliability monitoring. We demonstrated that NERL monitors can be used to analyze multi-party protocols, such as AODV, as well as two-party protocols, such as SMTP. We showed that NERL recognizers can express a variety of safety properties. They can check for conformance with full protocol state machines, as in SMTP, with logical requirements, as in AODV, and with simple counting properties, as in TCP. In the SMTP case study, we showed that NERL main modules can express and manage several layers of protocol recognizers.

Flexibility. We demonstrated that NERL monitors are applicable in network simulations (of AODV), in live network monitoring (of SMTP), and in offline analysis of (TCP) packet traces. We demonstrated that NERL programs can be effectively used for co-located monitoring, for bottleneck monitoring over reliable message streams, and for co-networked monitoring.

Correctness. For every recognizer in the thesis, we formalized the desired correctness properties and showed that they can be checked using SPIN. In the SMTP case study, we used the model-checking feature of NERL to show that two kinds of monitors were inadequate for our purpose. We demonstrated how a programmer can use model checking to find subtle errors in a protocol recognizer or to show that a recognizer is consistent with a protocol specification.

Diagnostics. In all three case studies, we demonstrated that the event tracing feature of NERL provides enough information to interpret implementation errors. For AODV, we showed that event tracing and tuning can be used together to guide the programmer directly to errors in the implementation without inspecting the source code.

Monitoring with Infidelity. We demonstrated that the channel transformation feature of NERL can generate monitors that account for trace infidelity in a co-networked monitoring environment. For TCP traces, we showed that co-networked monitors generated by NERL can distinguish between network errors and protocol errors. We showed that NERL can be used to program specialized co-networked monitors for simple properties.

Efficiency. We demonstrated that NERL can keep up with large network simulations (as in AODV) as well as several concurrent live protocol sessions (as in SMTP). In the presence of significant trace infidelities, the performance of automatically generated TCP co-networked monitors can be inadequate. For such cases, we demonstrated that efficient co-networked monitors for simple properties can be written directly in NERL.

In summary, NERL satisfies most of the requirements that we identified for passive monitoring. The only significant limitation is the poor performance of NERL monitors in some cases, such as co-networked monitoring. We plan to address this limitation by optimizing the NERL compiler to improve the performance of the event tracing and channel transformation components.

9.2 General Conclusions

Our case studies found several instances where a popular implementation deviates from its protocol specification. There can be several reasons for such deviations. TCP implementations violate the standard in an attempt to glean the maximum network performance. However, as Paxson illustrates (Pax97), such optimizations are often misguided and lead to suboptimal performance as well as implementation incompatibility. SMTP implementations deviate because they differ in their interpretations of the standard. Although some interpretations can be justified by security concerns, many interpretations are clearly incorrect and can only

lead to undesirable server behavior. The AODV implementation deviated from the specification because of simple programmer errors, possibly due to frequent modifications of the draft standard. We believe that protocol implementors should formally state the version and portion of the specification that they aim to support. Each implementation should then be tested against its specification. This dissertation demonstrates that formal tools like network event recognition can significantly improve the reliability of protocol implementations.

We have shown that passive protocol monitoring can effectively find errors in protocol implementations. We strongly advocate the use of passive protocol monitoring in all stages of protocol development. Passive monitoring adds little overhead to an existing testing framework, such as a network simulator. In lightly-loaded networks with no loss or delay, passive monitors can be used permanently to check that all the devices on the network are operating correctly. In networks with packet loss and delay, passive monitors can be used to analyze packet traces captured from a co-network monitor; and for some simple properties, passive monitors can be used online even for such networks.

9.3 Future Work

We envision three directions of future research based on this dissertation: improving the NERL implementation, extending the NERL language, and investigating the use of NERL monitors for network monitoring applications.

9.3.1 Implementing Optimized Trace Search Algorithms

The NERL compiler has a channel-transformation mode that implements the trace-search algorithm. Trace search performs a brute-force search through the state space of the composition of the protocol recognizer with a non-deterministic buffer. This algorithm is known to be very inefficient in the worst case. Even for specific properties of TCP, we showed that the algorithm does not scale to large traces.

However, for special classes of NERL programs that check simple safety properties, we can design specialized algorithms that are equivalent to trace search but significantly more efficient. For instance, in Chapter 8, we demonstrated an efficient algorithm for monitoring the TCP counting property in a co-networked environment.

In an earlier work, Bhargavan *et al.* (BCMG01) have identified 11 classes of protocol properties for which efficient co-networked monitoring is possible. For each of these property classes, they describe efficient channel transformation algorithms that are equivalent to trace search. Figure 9.1 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.)

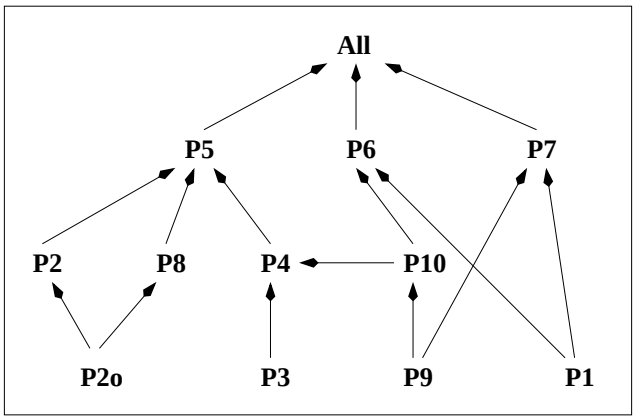

Figure 9.1 Co-networked Monitorable Properties

To generate efficient co-networked monitors from NERL programs, we intend to incorporate these specialized algorithms into the NERL compiler. This would entail two new extensions to the compiler: an algorithm for checking whether a NERL program lies in a given property class, and an automated channel transformation for each property class.

A further goal of our research is to automatically generate bottleneck monitors from NERL programs. Bottleneck monitors are important because they can easily monitor a large local network and can be used for intrusion detection. Recall that bottleneck monitoring

is vulnerable to additional trace infidelities such as output loss and output buffering. It is unknown whether efficient algorithms can be designed for bottleneck monitoring, but we believe that this is can be an area of productive research.

9.3.2 Language Extensions

The current design of NERL was motivated by the need to monitor the three protocols in the case studies. We envision that as we use NERL to analyze new protocols for new properties, we might need to extend the language. In particular, we think the following three extensions will significantly improve the design of NERL.

Real-time Properties. None of the case studies we presented analyzed real-time protocol properties. To extend the design of NERL for real-time properties, we would have to model timer events in the language. There are well known techniques for modeling and implementing timers in run-time verification systems and intrusion detection systems that we hope to reuse for NERL. Future case studies can then incorporate real-time properties. 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 or 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 a correct TCP handshake:

```
event (Syn;Synack;Ack) -> TCP_HandShake;
```

This event can be translated to the following NERL fragment:

```
transition Init   -> {tcp_handshake_state = 0;}

transition Syn    -> {tcp_handshake_state = 1;}

transition Synack
           OccurredWhen
           (tcp_handshake_state == 1)   -> {tcp_handshake_state = 2;}

event Ack
      OccurredWhen
      (tcp_handshake_state == 2)   -> TCP_Handshake

event Ack
      OccurredWhen
      (tcp_handshake_state == 2)   -> {tcp_handshake_state = 0;}
```

Such automatic translations would be even more convenient for complex regular expressions like (Syn;Synack;(Data | Ack)*;Fin).

A second extension would be to allow events expressed using past temporal operators. The past temporal operators are: So-far p ([−]p), Once p(<−>p), p Since q, p BackTo q, and Previous p ((−)p). For instance, the following formula checks that a reply matches a request sometime in the past

```
event EchoReply
       OccurredWhen !(<->EchoRequest) -> BadEchoReply
```

This can also be translated to a NERL fragment

```
transition Init -> {once_echorequest = false;}

event EchoRequest -> {once_echorequest = true;}

event EchoReply
       OccurredWhen !(once_echorequest == true) -> BadEchoReply
```

Temporal logic has been used in many run-time verification languages, and regular expressions are essential components of network intrusion detection systems. We believe that both these features can significantly enhance the programmer experience for NERL.

Formal Semantics In this thesis, we did not present a formal semantics for NERL. Instead, we described its informal semantics through examples and through two translations from NERL programs to protocol monitors. The first translation generated C programs that were executable, and the second translation generated formal models in the Promela language.

However, there are several interesting questions about protocol monitoring that cannot be answered by an informal semantics. For instance, we cannot establish theorems about the expressiveness of NERL. Such questions are not only of academic interest. A protocol monitor that runs in a real network is susceptible to denial-of-service attacks (PN98). To combat such attacks, we would like to establish a resource bound on NERL programs. For instance, we would like to prove that each recognizer instance generated by NERL corresponds to an active session and will use finite resources for a finite time period. To prove such theorems, we will need a formal operational semantics for NERL. We intend to develop a formal operational semantics and a formal type system for NERL. Our ultimate goal is to use the formal semantics to establish theorems about passive monitoring in general.

9.3.3 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. We believe that NERL can be used as a programming language for a variety of monitoring tools. We outline two possible applications of NERL to existing network monitoring applications.

Surveillance We are using NERL to build an open-source email surveillance tool called Open-Warrants (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 fil-

ter out the parts of the email that are not warranted. In contrast, surveillance tools such as Carnivore cannot provide these guarantees. We have yet to fully evaluate the effectiveness of NERL for network surveillance, but preliminary results are promising.

Detecting Infidelities Most network surveillance, intrusion detection and network management systems fail to account for trace infidelities. NERL provides a generic monitoring algorithm that may be applicable for all these systems. For instance, a version of the trace search algorithm could be used as a stand alone monitor that computes the amount of packet loss and delay on a local network and raises an alarm when a threshold is crossed. Such a monitor could also be used as one module of an intrusion detection system.

REFERENCES

[Axe99] Stefan Axelsson. Research in intrusion-detection systems: A survey. Technical report, Chalmers University of Technology, 1999.

[BB89] G.v. Bochmann and O. Bellal. Test result analysis with respect to formal specifications. In *Proc. 2-nd Int. Workshop on Protocol Test Systems, Berlin*, pages 272–294, October 1989.

[BCMG01] Karthikeyan Bhargavan, Satish Chandra, Peter J. McCann, and Carl A. Gunter. What packets may come: Automata for network monitoring. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'01)*, pages 206–219. ACM Press, January 2001.

[BDD+90] G.v. Bochmann, D. Desbiens, M. Dubuc, D. Ouimet, and F. Saba. Test result analysis and validation of test verdicts. In *Proc. Workshop on Protocol Test Systems (IFIP)*, 1990.

[BDZ89] G.v. Bochmann, R. Dssouli, and J.R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Tr. on Soft. Eng.*, 15(11):1347–1356, November 1989.

[BG02a] Karthikeyan Bhargavan and Carl A. Gunter. Network Event Recognition for Packet Mode Surveillance. Submitted for publication, 2002.

[BG02b] Karthikeyan Bhargavan and Carl A. Gunter. Requirements for a practical network event recognition language. In *Proceedings of the Runtime Verification Workshop*, July 2002.

[BGG+98] Karthikeyan Bhargavan, Carl A. Gunter, Elsa L. Gunter, Michael Jackson, Davor Obradovic, and Pamela Zave. The Village Telephone System: A case study in formal software engineering. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics 11th International Conference TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*, pages 49–66, Canberra, Australia, September 1998. Springer.

[BGK+00] Karthikeyan Bhargavan, Carl A. Gunter, Moonjoo Kim, Insup Lee, Davor Obradovic, Oleg Sokolsky, and Mahesh Viswanathan. Verisim: Formal analysis of network simulations, August 2000. International Symposium on Software Testing and Analysis.

[BGK+02] K. Bhargavan, C.A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal Analysis of Network Simulations. *IEEE Transactions on Software Engineering*, 28(2):129–145, February 2002.

[BGO00] Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic. Fault origin adjudication, August 2000. Formal Methods in Software Practice.

[BHW97] Randy Barrett, Steven Vonder Haar, and Randy Whitestone. Routing snafu causes internet outage. Interactive Week, Available online http://www.info-sec.com/denial/denial_3.html-ssi, April 1997.

[BMJ+98] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, October 1998.

[BOG02] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols, 2002. Journal of the ACM, to appear.

[BP94] Gregor V. Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 international symposium on Software testing and analysis*, pages 109–124. ACM Press, 1994.

[Bra89a] R. Braden. Requirements for internet hosts – application and support. Technical Report RFC 1123, IETF, 1989.

[Bra89b] R. Braden. Requirements for internet hosts – communication layers. Technical Report RFC 1122, IETF, 1989.

[CCI76] CCITT. Specification and Description Language (SDL), 1976.

[CEM98] John R. Callahan, Stephen M. Easterbrook, and Todd L. Montgomery. Generating test oracles via model checking, 1998. Unpublished, working paper. Available at http://research.ivv.nasa.gov/projects/blurbs/FormalTesting.html.

[CFP93] A. Cavalli, J.P. Favreau, and M. Phalippou. Formal methods in conformance testing: Results and perspectives. In *Proceedings of the 6th International Workshop on Protocol Test System*, Pau, France, September 1993.

[Cri96] M. Crispin. Internet Message Access Protocol - Version 4rev1. Technical Report RFC 2060, IETF, 1996.

[Cro82] D. Crocker. Standard for the Format of ARPA Internet Text Messages. Technical Report RFC 822, IETF, 1982.

[CS96] J. Callahan and F. Schneide. Specification-based testing using model checking. In *1996 SPIN Workshop, Rutgers University*, August 1996.

[DJC94] M. Diaz, G. Juanole, and J.-P. Courtiat. Observer-a concept for formal on-line validation of distributed systems. *IEEE Transactions on Software Engineering*, 20(12):900–913, 1994.

[DJM96] Scott Dawson, Farnam Jahanian, and Todd Mitton. ORCHESTRA: A fault injection environment for distributed systems. Technical Report CSE-TR-318-96, University of Michigan, 26 1996.

[DJM97] Scott Dawson, Farnam Jahanian, and Todd Mitton. Experiments on six commercial TCP implementations using a software fault injection tool. *Software Practice and Experience*, 27(12):1385–1410, 1997.

[DR96] Laura K. Dillon and Y.S. Ramakrishna. Generating Oracles from Your Favorite Temporal Logic Specifications. In *Proc. 4th ACM SIGSOFT Symp. Foundations of Software Engineering, San Francisco*, pages 106–117, October 1996.

[DY94] Laura K. Dillon and Q. Yu. Oracles for Checking Temporal Properties of Concurrent Systems. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'94)*, volume 19(5), pages 140–153, December 1994. Proceedings published as Software Engineering Notes.

[EB95] S.A. Ezust and G.v. Bochmann. An Automatic Trace Analysis Tool Generator for Estelle Specifications. *Computer Communication Review*, 25(4):175–184, October 1995. Proceedings of ACM SIGCOMM 95 Conference.

[Est89] The Extended State Transition Language (Estelle), 1989.

[FP76] David J. Farber and J. B. Picken. The overseer, a powerful communications attribute for debugging and security in thin-wire connected control structures. In *Proceedings of International Computer Communications Conference*, 1976.

[FV00] Kevin Fall and Kannan Varadhan. *ns Notes and Documentation*. The VINT Project, February 2000.

[GGJZ00] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, May 2000.

[God97] Patrice Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of the 9th Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 476–479, Haifa, Israel, June 1997. Springer-Verlag.

[Gro01] NSS Group. Intrusion detection systems - group test, December 2001.

[GS02] G.J.Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. on Software Engineering*, 28(4):364–377, April 2002.

[HE98] Ahmed Helmy and Deborah Estrin. Simulation-based 'STRESS' Testing Case Study. In *Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, July 1998.

[HEG98] Ahmed Helmy, Deborah Estrin, and Sandep Gupta. Fault-oriented Test Generation for Multicast Routing Protocol Design. In *Formal Description Techniques & Protocol Specification, Testing, and Verification (FORTE/PSTV) IFIP*, November 1998.

[HKL+95] M. Hlady, R. Kovacevic, J.J. Li, B.R. Bekilis, D. Prairie, T. Savor, R.E. Seviora, D. Simser, and A. Vorobiev. An approach to automatic detection of software failures. In *Proc. International Symposium on Software Reliability Engineering (ISSRE'95)*, pages 314–323. IEEE Computer Society Press, October 1995.

[HLSV00] R. Hao, D. Lee, R. Sinha, and D. Vlah. "testing ip routing protocols: From probabilistic algorithms to software tool". In *Proceedings of FORTE/PSTV*, Pisa, Italy, October 2000.

[Hol] Gerard J. Holzmann. SPIN-formal verification. Web Page. Available at http://netlib.bell-labs.com/netlib/spin/whatispin.html.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991. http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html.

[How] Rick Hower. Recent major computer system failures caused by software bugs. Available online http://www.softwareqatest.com/qatfaq1.html.

[HPK01] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium 2001*, 2001.

[Ins81a] Information Sciences Institute. Internet Protocol. Technical Report RFC 791, IETF, 1981.

[Ins81b] Information Sciences Institute. Transmission Control Protocol. Technical Report RFC 793, IETF, 1981.

[JPO95] L. Jagadeesan, C. Puchol, and J. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In *Proceedings of the 7th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 127–140, July 1995.

[JPP+97] L. J. Jagadeesan, A. Porter, C. Puchol, J. C. Ramming, and L. G. Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the International Conference on Software Engineering*, May 1997.

[JZ92] Michael Jackson and Pamela Zave. Domain descriptions. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 56–64. IEEE Computer Society Press, 1992.

[JZ95] Michael Jackson and Pamela Zave. Deriving specifications from requirements: an example. In *Proceedings of the Seventeenth International Conference on Software Engineering*, pages 15–24. IEEE Computer Society Press, 1995.

[Kle01] J. Klensin. Simple Mail Transfer Protocol. Technical Report RFC 2821, IETF, 2001.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, second edition, 1988.

[Kva99] Hakan Kvarnstrom. A survey of commercial tools for intrusion detection. Technical report, Chalmers University of Technology, 1999.

[KVBA+98] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. A framework for run-time correctness assurance of real-time systems. Technical Report MS-CIS-98-37, University of Pennsylvania, 1998.

[KVBA+99] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *Proceedings European Conference on Real-Time Systems*, 1999.

[KY94] Bogdan Korel and Satish Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the 1994 international symposium on Software testing and analysis*, pages 66–79. ACM Press, 1994.

[LH98] Tony Larsson and Nicklas Hedman. Routing protocols in wireless ad-hoc networks - a simulation study. Master's thesis, LuleåUniversity of Technology, Stockholm, 1998.

[LKK+99] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M.Viswanathan. Runtime assurance based on formal specifications. In *Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[LNS+97] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *Proc. International Conference on Network Protocols (ICNP '97)*, Atlanta, USA, October 1997.

[Lot87] LOTOS: a formal description technique, 1987.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quaterly*, 2(3):219–246, 1989.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[MF99] Daniel Massey and Bill Fenner. Fault detection in routing protocols. In *Proceedings of the Seventh Annual International Conference on Network Protocols*, Toronto, Canada, October 1999.

[MJ98] G. Robert Malan and Farnam Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, 1998.

[MR96] J. Myers and M. Rose. Post Office Protocol - Version 3. Technical Report RFC 1939, IETF, 1996.

[MWJH00] G. Robert Malan, David Watson, Farnam Jahanian, and Paul Howell. Transport and application protocol scrubbing. In *Proceedings of the IEEE INFOCOM 2000 Conference*, March 2000.

[ORD96] T.O. O'Malley, D.J. Richardson, and L.K. Dillon. Efficient Specification-Based Test Oracles. In *Second California Software Symposium (CSS'96)*, April 1996.

[PAD+99] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known tcp implementation problems. Technical Report RFC 2525, IETF, 1999.

[Pax97] V. Paxson. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM '97*, September 1997.

[Pax99] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31:2435–2463, 14 December 1999. This paper is a revision of paper that previously appeared in Proc. 7th USENIX Security Symposium , January 1998.

[Per97] Charles Perkins. Ad Hoc On-Demand Distance Vector (AODV) Routing. Internet-Draft Version 00, IETF, November 1997.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., 1998.

[Pos81] J. Postel. Internet Control Message Protocol. Technical Report RFC 792, IETF, 1981.

[Pos82] Jonathan B. Postel. Simple Mail Transfer Protocol. Technical Report RFC 821, IETF, 1982.

[PR99] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, pages 90–100, February 1999.

[Proa] The GNU Project. *The C Preprocessor*. Free Software Foundation. http://gcc.gnu.org/onlinedocs/cpp/.

[Prob] The VINT Project. The ns Network Simulator. Web Page. Available at http://www-mash.CS.Berkeley.EDU/ns.

[PS98] S. Parker and C. Schmechel. Some testing tools for tcp implementors. Technical Report RFC 2398, IETF, 1998.

[PYB96] A. Petrenko, N. Yevtushenko, and G.v. Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. In *Proc. of 9th International Workshop on Testing of Communicating Systems (IWTCS'96)*, pages 125–140, 1996.

[RAO92] D.J. Richardson, S. Leif Aha, and T.O. O'Malley. Specification-Based Oracles for Reactive Systems. In *14th International Conference on Software Engineering*, May 1992.

[Res01] P. Resnick. Internet Message Format. Technical Report RFC 2822, IETF, 2001.

[Ric94] D.J. Richardson. TAOS: Testing with Analysis and Oracle Support. In *Proceedings of the International Symposium on Software Testing and Analysis*, August 1994.

[SAN] The SANS Institute. http://www.sans.org.

[SCA] Self-checking active networks (scan). http://www.cis.upenn.edu/~jms/SCAN.html.

[SCB91] Deepinder Sidhu, Anthony Chung, and Thomas P. Blumer. Experience with formal methods in protocol development. *ACM SIGCOMM Computer Communication Review*, 21(2):81–101, 1991.

[SCHP+00] Stephen P. Smith, J. Allen Crider, Jr. Henry Perrit, Mengfen Shyong, Harold Krent, Larry L. Reynolds, and Stephen Mencik. Independent review of the carnivore system - final report. Technical report, IIT Research Institute, December 2000.

[SK00] Raymond Smith and Bogdan Korel. Slicing event traces of large software systems. In *Proceedings of Automated and Algorithmic Debugging (AADEBUG'00)*, 2000.

[SS96] T. Savor and R.E. Seviora. An architectural overview of a software supervisor. In *Proc. 8th Euromicro Workshop on Real-Time Systems*, pages 52–56, June 1996.

[SS97] T. Savor and R.E. Seviora. An approach to automatic detection of software failures in real-time systems. In *Proc. 1997 Real-Time Technology and Applications Symposium*, pages 136–146, June 1997.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, Massachusetts, 1994.

[TCI99] Marine Tabourier, Ana R. Cavalli, and Melania Ionescu. A GSM-MAP protocol experiment using passive testing. In *Proceedings of World Congress on Formal Methods in development of Computing Systems (FM'99)*, pages 915–934, Toulouse, France, September 1999.

[Vis00] Mahesh Vishwanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, 2000.

[YEG00] Yan Yu, Deborah Estrin, and Sandeep Gupta. Worst case performance analysis of wireless ad-hoc routing protocols: Case studies. Technical Report 00-730, USC, 2000. Submitted for Publication.

[Zim80] H. Zimmerman. OSI reference model - the ISO model of architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, COM-28(4), April 1980.

[ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology, 6(1):1–30, January 1997.

[ZWY02] Yxin Zhao, Jianping Wu, and Xia Yin. From active to passive: Progress in testing of internet routing protocols. *Journal of Computer Science and Technology*, 17(2):264–283, 2002.

APPENDIX A

NERL SYNTAX REFERENCE

A.1 Tokens

The following lexical tokens appear in NERL programs.

Var	::=	`[a-z][a-z0-9\_]*`	state variables
EVar	::=	`[A-Z][a-z0-9\_]*`	event variables
TVar	::=	`[a-z][a-z0-9\_]*`	type variables
Int	::=	`[0-9]+`	integers
Double	::=	`[0-9]*.[0-9]+`	floating-point numbers
String	::=	`"[^"]*"`	strings
Bool	::=	`true | false`	booleans
RVar	::=	`[A-Z][a-z0-9\_]*`	recognizer names
IVar	::=	`[A-Z][a-z0-9\_]*`	recognizer instance names

Notes:

- We use the names of the lexical categories (e.g. *Var*) as metavariables in this chapter.
- By convention, we use the prefix `ty_` for type variables.

Examples:

`status`, `sequence_no` are state variables.

`Packet`, `Route_Error` are event variables.

`ty_pkt`, `ty_cell` are type variables.

`13143431, 0` are integers.

`.245, 1234.0` are doubles.

`"", "foo bar"` are strings.

`true, false` are the only bools.

`Ping`, `SMTP_Recog` are recognizer names.

`P`, `S` are recognizer instance names.

A.2 Reserved Words

The reserved words in NERL are listed below:

typedef	state	event	input	output	length
bool	int	double	string	true	false
concat	push	pop	if	then	else
while	do	Tick	Init	Done	transition
Recognizer	EndRecognizer	signature	recognizer	instance	
WithAttributes	OccuredWhen	WithIndex	begin	end	
#define	#if	#endif	#include		
{	}	[	]	(	)
‖	&&	|	&	⇒	→
+	−	*	/	!	.
==	! =	<	>	<=	>=
=	#	;			

A.3 Pre-processing

We use the C pre-processor cpp to process macro definitions. Pre-processor directives, such as #define, #if, #endif, and #include, appear in NERL programs but are not part of the NERL syntax; these directives are processed and removed by cpp. In the rest of this appendix, we will only describe pre-processed NERL programs. More details on cpp can be found in C language manuals (KR88) and as compiler documentation (Proa).

A.4 Types

Every variable and event in a NERL program is given a type that determines how it may be used. Types are classified as array types and record types. Array types are divided into basic types and type variables.

Ty	$::=$	$ArrayTy$	array types
		$RecordTy$	record types

A.4.1 Basic Types

Variables containing numbers, booleans or strings are said to have a basic type.

BTy	$::=$	`bool`	booleans
		`int`	32 bit integers
		`double`	64 bit floating point nos
		`string`	byte strings

Notes:

- The default values for these types are as follows:

Type	Default Value
bool	false
int	0
double	0.0
string	""

- `int`s can be used in place of `double`s; the other basic types are distinct.
- `string`s have a length parameter; they are not NUL-terminated.
- `string`s need not be printable.

Examples:

```
int x;
bool y;
double z;
string w;
x = 12345;
y = true;
if (y == false) then {z = x + 0.245;}
else {w = ``hello world''}
```

See Also: Values (A.6.1)

A.4.2 Atomic Types

Type variables are used as abbreviations for complex types.

ATy	::=	BTy	basic types
		$TVar$	type variables

Notes:

- There are no recursive types; so type variables can be replaced at will with their expanded forms.
- When a type variable is used to abbreviate a complex type, the default value of the new type is the same as the default value of the complex type.

Examples:

```
typedef int ty_int;  /* type variable ty_int is
                        another name for int */
ty_int n;
n = 1234;
```

See Also: Record Types (A.4.4), Type Declarations(A.5.2)

A.4.3 Arrays

NERL programs can have multi-dimensional arrays of cells, where each cell has an atomic type. Arrays may also have variable length, in which case they behave like stacks; cells must be allocated at run-time, using push.

$ArrayTy$	::=	ATy	atomic types
		$ArrayTy[Int]$	integer indexed arrays
		$ArrayTy[]$	variable-size integer-indexed arrays

Notes:

- Arrays have a `length` parameter.
- Arrays are indexed from `0` to `length-1`.
- Array variables are declared in C-style: with length appearing after the variable name.
- The default value of a fixed-size array `a` with cell type `T` and length `n` is an array of `n` cells each initialized to the default value for `T`.
- The default value of a variable-size array with cell type `T` is an array with `0` cells.

Examples:

```
int x[10];          /* x has type int[10] */
double y[];         /* y has type double[] */
x[0] = x[9] + 10;
if (y#length > 0) then {y[0] = x[0]}
else {push(y,1); y[0] = x[0]}
```

See Also: Assignable Locations (A.6.2), Array Parameters (A.6.3), Assignment (A.7.2), Variable Array Operations (A.7.4)

A.4.4 Record Types

Records are finite lists of values indexed by labels. They are similar to C-structs.

$RecordTy$	$::=$	$\{Var : ArrayTy, (Var : ArrayTy)*\}$	records

- All labels in a record type must be distinct.
- Records are defined Promela-style: the type of the record must first be abbreviated by a type variable, and then the type variable can be used in state variable declarations.
- The default value of a record-type is a record with the required fields where all fields have the default values of their respective types.

Examples:

```
typedef {
  int x;
  double y;
} ty_myrecord;

ty_myrecord r; /* r has type { x:int, y:double } */

r.x = 1;
r.y = r.x + 0.5;
```

See Also: Type Declarations (A.5.2), Assignable Locations (A.6.2), Assignment (A.7.2)

A.5 Declarations

The preamble of a NERL program contains a sequence of declarations, where the types and names of state variables and events are defined.

D	$::=$	$TyDec$	type abbreviations
		`state` $VarDec$	state variable declaration
		$EvDec$	event declaration

A.5.1 Variable Declarations

Variables declarations introduce local variables in statements as well as state variables in the recognizer preamble.

VarDec	::=	*ATy Var*	simple variables
		ATy Var[*Int*]	arrays
		ATy Var[]	variable-size arrays

Notes:

- The value of the variable is set to a default value. This is done recursively for arrays and records.

Examples:

```
string x;      /* x has type basic string */
ty_mypkt y;    /* y has type defined for variable ty_mypkt */
double z[12]; /* z has type double[12] */
ty_mypkt w[]; /* w has type ty_mypkt[] */
```

See Also: Types (A.4)

A.5.2 Type Declarations

User-defined types are introduced by type declarations.

TyDec	::=	typedef *ATy TVar*	simple type renaming
		typedef *ATy TVar*[*Int*]	array type abbreviation
		typedef *ATy TVar*[]	variable array type abbreviation
		typedef {*VarDec*; *VarDec*∗} *TVar*	record types

Examples:

```
typedef string   ty_mystr;      /* ty_mystr abbrev for string */
typedef int      ty_intarr[10] /* ty_intarr abbrev for int[10] */
typedef {
  int x;
  ty_mystr y
} ty_myrec;                     /* ty_myrec abbrev for
                                        {x:int,y:ty_mystr} */
```

```
typedef ty_myrec ty_table[]   /* ty_table abbrev for ty_myrec[] */
```

See Also: Types (A.4)—Atomic Types (A.4.2)

A.5.3 Event Declarations

Event declarations introduce the input, local and output events of a recognizer.

EvDec	::=	`event` *ATy EVar*	local event
		`input event` *ATy EVar*	input event
		`output event` *ATy Evar*	output event

Examples:

```
input  event ty_pkt Packet          /* input event                    */
event  ty_pkt GoodPacket            /* local event, triggered by Packet */
output event ty_pkt ModifiedPacket; /* output meta-event for higher-layer,
                                       triggered by GoodPacket        */
output event string Error           /* output error event,
                                       with message for user          */
```

See Also: Types (A.4), Events (A.8), Event Definitions (A.10.1),

Signature Declarations (A.12.1)

A.6 Expressions

Arithmetic and boolean operations are used to filter events, and execute state transitions.

E	::=	*V*	values
		L	assignable locations
		Op	arithmetic operations
		Cmp	comparison operations
		Bop	boolean operations
		AOp	array parameters

A.6.1 Values

The basic values that appear in the states and events of a NERL program are numbers, strings, and booleans.

V	::=	*Int*	integers
		Double	doubles
		String	strings
		Bool	booleans

Examples:

`13143431, 0` are integer values.

`.245, 1234.0` are double values.

`"", "foo bar"` are string values.

`true, false` are the only bool values.

See Also: Tokens (A.1), Basic Types (A.4.1)

A.6.2 Assignable Locations

Locations are used to address mutable values, such as array cells and record fields.

L	::=	*Var*	state variables
		$L[E]$	array indexing (E is an integer expression)
		$L.Var$	record projection

Examples:

```
state int n;        /* x is a location */
state int a[10];    /* a[5] is a location, a[2*n] is a location */

typedef {int x; double y} ty_myrec;
state ty_myrec r; /* r.x, r.y are locations */

n = 1;
a[5] = n;
a[2*n] = 5;
z.x = 2*5 + n;
z.r = 0.9;
```

See Also: Variable Declarations (A.5.1), Assignment (A.7.2)

A.6.3 Array Parameters

The length parameter of an array returns its current length. This parameter could change over the lifetime of a variable-size array (but cannot change for fixed-size arrays).

AOp	::=	$L\#$length	length of an array

Examples:

```
state int a[10];
state int b[];

int n = 0;
while (n < a#length) {a[n] = 0; n = n + 1};
if (b#length < 10) {push(b,10-b#length)};
```

See Also: Arrays (A.4.3), Variable Declarations(A.5.1), Variable Array Operations (A.7.4)

A.6.4 Arithmetic Operations

Familiar arithmetic operations on numbers are allowed.

Op	::=	$E_1 + E_2$	Addition
		$E_1 - E_2$	Subtraction
		$E_1 * E_2$	Multiplication
		E_1 / E_2	Division
		$-E$	unary negation

Examples:

```
int n;
double d;

n = n + n*2 - n/4;
d = -0.13*n + 3.5 + 2*n/d;
```

See Also: Variable Declarations (A.5.1), Values (A.6.1)

A.6.5 Comparisons

Arithmetic expressions can be compared using standard numerical comparison operators. In addition, strings can be compared for equality and inequality.

Cmp	::=	$E_1 == E_2$	Equals
		$E_1 < E_2$	Less Than
		$E_1 > E_2$	Greater Than
		E_1 != E_2	Not equals
		$E_1 <= E_2$	Less Than or equals
		$E_1 >= E_2$	Greater Than or equals

Notes:

- String equality is tested by `strncmp`; both the size and contents of the two strings must be equal.

Examples:

```
int n;
double d;
bool b;

if (n == 0) then {b = (d > 0.5);}
while ((d/2) < 10) do {d = d+1; b = (n != 0);}
```

See Also: Variable Declarations (A.5.1), Values (A.6.1), Arithmetic Operations (A.6.4)

A.6.6 Boolean Operations

Boolean conditions can be combined by conjunction, disjunction, implication and negation.

BOp	::=	$E_1 \parallel E_2$	Or
		E_1 && E_2	And
		$E_1 \Rightarrow E_2$	Implies
		$!E$	not

Examples:

```
int n;
bool b;

if (b || (n == 0)) then {b = false;}
while (b && (n < 10)) do {n = n+1; b = b => (n > 0);}
```

See Also: Variable Declarations (A.5.1), Comparisons (A.6.5)

A.7 Statements

Statements modify state variables; they are executed during state transitions. In addition, statements are used to assign values to events.

S	::=	$VarDec$	local variable decl
		Ass	assignment
		$StrOp$	string operations
		$VAOp$	variable array operations
		$\{\}$	noop
		$\{S_1; S_2; \ldots; S_n\}$	sequencing
		`if` E `then` S_1 `else` S_2	conditional branch
		`while` E `do` S	while loop

A.7.1 Local Variable Declarations

Temporary variables can be declared within statements as follows, they are deallocated at the end of the current statement sequence (the enclosing $\{\}$).

$VarDec$	::=	$ATy\ Var$	simple variables
		$ATy\ Var[Int]$	arrays
		$ATy\ Var[]$	variable-size arrays

Notes:

- The value of the local variable is set to a default value. This is done recursively for arrays and records.

Examples:

```
{
  string x;      /* local variable x has type string  */
  int y[10];     /* local variable y has type int[10] */

  x = ``foo bar''
  y[0] = 10;
}                /* x,y are deallocated */
```

See Also: Types (A.4), Variable Declarations (A.5.1), Sequencing (A.7.6)

A.7.2 Assignment Statements

Assignment statements are used to give new values to variables.

Ass	$::=$	$L = E$	assignment

Notes:

- The location L must be valid; any record labels it uses must be defined, and any array indices must be within array-bounds.
- Both L and E should have basic types; copying reords or arrays is not allowed.
- The type of E should be the same as that of L, except that E can be a double and L can be an int.

Examples:

```
typedef {int f1; int f2} ty_rec;

string x;
int y[100];
ty_rec z[10][10];

x = ``foo bar''
y[0] = 10;
z[5][6].f1 = y[3];
```

See Also: Assignable Locations (A.6.2), Expressions (A.6)

A.7.3 String Operations

The only operation allowed on strings is concatenation. It takes a string s_1 at a location, concatenates it with another string s_2, and puts the result in the same location.

$StrOp$	::=	$\texttt{concat}(L_1, E_2)$	string concatenation

Examples:

```
string x;

x = ``foo'';          /* x contains ``foo''      */
concat(x,`` bar''); /* x contains ``foo bar'' */
concat(x,x);          /* x contains ``foo barfoo bar'' */
```

See Also: Tokens (A.1), Assignable Locations (A.6.2)

A.7.4 Variable Array Operations

Fixed-size arrays are allocated when they are declared, thereafter they are modified by assignment statements. On the other hand, variable-size arrays can also be extended and reduced in length.

$VAOp$	::=	$\texttt{push}(L, E)$	insert E empty cells at end of var. array L
		$\texttt{pop}(L, E)$	delete last E cells from var. array L

Notes:

- The value of a newly pushed element is the default value for its type.

Examples:

```
int x[];              /* x has type int[]          */
                      /* x has no cells            */
                      /* x#length = 0              */
```

```
push(x,5);          /* x has 5 cells                 */
                    /* x[0] = x[1] =...= x[4] = 0 */
x[2] = 5;           /* x[2] is modified              */
x[4] = x[2] + 1;    /* x[4] = 6                      */

pop(x,2);           /* x has 3 cells                 */
                    /* x[3], x[4] are deleted        */

push(x,3);          /* x has 6 cells                 */
                    /* x[3] = x[4] = x[5] = 0        */
```

See Also: Arrays (A.4.3), Variable Declarations (A.5.1), Array Parameters (A.6.3), Assignment (A.7.2)

A.7.5 Noops

Noops are used to signify empty statements. Usually we employ syntactic sugar to avoid writing noops.

$$S \quad ::= \quad \{\} \quad \text{noop}$$

Examples:

```
if (x == 1) then {y = 2} else {};

transition Foo(r) -> {}
event Foo(r) -> Bar WithAttributes {}
```

See Also: Sequencing (A.7.6)

A.7.6 Sequencing

Statements can be put together into a sequential block using sequencing (;).

$$S \quad ::= \quad \{S_1; S_2; \ldots; S_n\} \quad \text{sequencing } (n \geq 1)$$

Examples:

```
{
  if (x == 1) then {y = 2} else {};
```

```
    y = 2;
    concat(s, ``some string'')
  }

  transition Foo(r) -> {int i;
                        int j;
                        i = 0;
                        j = 0;
                        while (i<10) do {
                          j = j * i;
                          i = i + 1
                        };
                        n = j}
```

A.7.7 If Conditionals

We use if-then-else constructs for conditional control-flow.

S	::=	`if` E `then` S_1 `else` S_2	conditional branch

Notes:

- We omit the else clause in the conditional if the corresponding statement is a noop.

Examples:

```
if (x == 1) then {y = 2} else {z = 3};

if (a#length == z-1) then {push(a,1); a[z-1] = a[z-2]};
```

See Also: Comparisons (A.6.5), Boolean Operations (A.6.6), Sequencing (A.7.6)

A.7.8 While Loops

Standard while-loops are allowed.

S	::=	`while` E `do` S	while loop

Examples:

```
{int i;
 int j;
```

```
i = 0;
j = 0;
while (i<10) do {
  j = j * i;
  i = i + 1
};
n = j}
```

See Also: Comparisons (A.6.5), Boolean Operations (A.6.6), Sequencing (A.7.6)

A.8 Events

An event is a temporary variable that consists of a boolean flag, indicating its occurrence, and an attribute whose value is assigned by a statement.

V	$::=$	$EVar\ WithAttributes\ S$	event with attribute
		$EVar$	event with no attribute

Notes:

- The distinguished input event **Init WithAttributes** S occurs only once—when the recognizer is initialized; S carries the initialization parameters.
- The distinguished input event **Tick WithAttributes** {time = T} occurs at each round, and T is the current time.
- The distinguished output event **Done** is triggered when the recognizer has finished processing the monitored session.

Examples:

```
typedef {int x; int y} ty_rec;

input event  int Foo;
event ty_rec Bar;

event Foo -> Bar(r) WithAttributes {r.x = 0; r.y = 1};
```

See Also: Event Declarations (A.5.3), Statements (A.7)

A.9 Event Patterns

Event patterns are used as preconditions for state transitions and event definitions. They are defined as predicates on possible event occurences and the current state.

Ev	::=	$EPat$	basic patterns
		EOp	event correlation
		$EFil$	filtering events

A.9.1 Basic Patterns

Basic patterns match event variables; they check whether the flag for the specified event variable has been set, and if yes, they extract the attribute of the event.

$EPat$	::=	$EVar(Var)$	match event, extract attributes
		$EVar$	match event

Examples:

```
typedef {int x; int y} ty_rec;

state int lastfoo;
input event ty_rec Foo;
output event int Bar;

event Foo(r) -> Bar(n) WithAttributes {n = r.x + r.y};
transition Foo(r) -> {lastfoo = r.x};
event Bar -> Done;
```

See Also: Event Declarations(A.5.3), Events(A.8)

A.9.2 Event Correlation

Event patterns can be combined using disjunction and conjunction to produce complex predicates on events.

Eop	::=	$Ev \mid Ev$	match one of the two patterns
		Ev & Ev	match both patterns, extract attributes

Notes:

- Pattern disjunctions cannot extract attributes: `Foo(r) | Bar(n)` is not well-defined; only `Foo | Bar` is.

Examples:

```
typedef {int x; int y} ty_rec;

state int lastfoo;
input event ty_rec Foo;
event int GoodFoo;
output event int Bar;

transition Foo(r) & GoodFoo -> {lastfoo = r.x};
event Foo | Bar -> Done;
```

See Also: Basic Patterns (A.9.1)

A.9.3 Event Filtering

To express preconditions for state transitions and event definitions, we use filtering conditions on event patterns. These conditions check whether some boolean predicate holds of the matched event and the current state.

EFil	::=	Ev `OccurredWhen` E	Ev occurred when E was true

Examples:

```
typedef {int x; int y} ty_rec;

state int lastfoo;
input event ty_rec Foo;
output event int Bar;

transition Foo(r) OccurredWhen (r.x == r.y) -> {lastfoo = r.x};
transition Foo(r) OccurredWhen ((r.x != r.y) &&
                    (lastfoo != r.x)) -> {lastfoo = r.x + r.y};
```

See Also: Basic Patterns (A.9.1), Comparisons(A.6.5), Boolean Operations(A.6.6)

A.10 Guarded Commands

A recognizer consists of a sequence of guarded commands: event definitions and state transtions. Each command is guarded by an event pattern that defines a precondition; if it succeeds the right-hand-side of the command is executed, otherwise it is skipped.

A.10.1 Event Definitions

For event definitions, the right-hand-side generates a new event.

C	::=	event $Ev \rightarrow E$	event definitions

Notes:

- The scope of event attribute variables defined in the event pattern extends to the right-hand-side.

Examples:

```
state int lastfoo;
input event ty_rec Foo;
output event int Bar;

event Foo(r) -> Bar(n) WithAttributes {n = r.x};
```

```
event Foo(r) OccurredWhen
            ((r.x != r.y) &&
             (lastfoo != r.x)) -> Bar(n) WithAttributes
                                         {if (lastfoo != r.y) then {
                                           lastfoo = r.y;
                                          }};
```

See Also: Event Declarations (A.5.3), Events (A.8), Event Patterns(A.9)

A.10.2 State Transitions

For state transitions, the right-hand-side modifies the state variables.

C	::=	transition $Ev \rightarrow S$	state transitions

Notes:

- The scope of event attribute variables defined in the event pattern Ev extends to S.

Examples:

```
typedef {int x; int y} ty_rec;

state int lastfoo;
input event ty_rec Foo;

transition Foo(r) OccurredWhen (r.x == r.y) -> {lastfoo = r.x};
transition Foo(r) OccurredWhen (r.x != r.y) -> {
                                    int i;
                                    int j;
                                    i = 0;
                                    j = 0;
                                    while (i < r.x) do {
                                       j = j * r.y;
                                       i = i + 1;
                                    }
                                    lastfoo = j};
```

See Also: Event Patterns (A.9), Statements(A.7)

A.11 Recognizer Definitions

A recognizer consists of declarations followed by commands.

R	::=	*Recognizer* $RVar =$	
		$(D;)*$	State, Event Declarations
		$(C;)*$	Commands
		EndRecognizer	

See Also: Declarations (A.5), Guarded Commands (A.10)

Examples:

```
Recognizer R =
  typedef {int x; int y} ty_rec;

  state int lastfoo;

  input event int Init;
  input event ty_rec Foo;
  output event int Bar;

  Init(i) -> {lastfoo = i};

  transition Foo(r) OccurredWhen (r.x == r.y) -> {lastfoo = r.x};
  event Foo(r) OccurredWhen ((r.x != r.y) &&
                        (lastfoo != r.x)) -> Bar(n) WithAttributes
                                                {n = lastfoo};
EndRecognizer;
```

A.12 Main Module Declarations

The preamble of the main module contains type declarations, recognizer signature declarations, and instance declarations.

MainDec	::=	*TyDec*	New type abbreviations
		SigDec	Recognizer signatures
		IDec	Recognizer instances

A.12.1 Signature Declarations

A recognizer signature consists of input and output event declarations.

$SigDec$	::=	***recognizer*** $RVar : \{(EvDec;)*\}$	Signatures

Examples:

```
recognizer R : {
  input event int Init;
  input event ty_rec Foo;
  output event int Bar;
};

recognizer Cap: {
  input event string Init;
  output event ty_rec Ofoo
};
```

See Also: Recognizer Definitions (A.11), Event Declarations (A.5.3)

A.12.2 Instance Declarations

We can define a fixed number of instances of a recognizer, or a potentially infinite table of instances, one for each index. In both cases, the instance declaration must provide initialization parameters; it must provide the attribute for `Init`.

$IDec$	::=	`instance` $RVar$ $IVar(Var)$ $WithAttributes\ S$	single instance
		`instance` $RVar[Ty]$ $IVar[Var](Var)$ $WithAttributes\ S$	table of instances

Examples:

```
instance Cap C(init) WithAttributes {init = ``blah''};

instance R I1(init) WithAttributes {init = 0};

instance R I2(init) WithAttributes {init = -1};

instance R[int] J[index](init) WithAttributes {init = index};

typedef {bool b; int n} ty_idx;

instance R[ty_idx] K[index]{init} WithAttributes
```

```
{if (index.b == true) then
    {init = index.n}
 else {init = 0}}
```

See Also: Recognizer Definitions (A.11), Event Definitions (A.10.1)

A.13 Instance Input Events

In the main module, a particular input event is referred to by prefixing it with the instance name. In the case that we want to trigger the input of an instance in a table, we must provide the index of the instance as well.

V_I	::=	$IVar.EVar$	input event at instance
		$IVar.EVar(Var)\,WithAttributes\ S$	input event w/attribute at instance
		$IVar[Var].EVar\ \ WithIndex\ S$	input event at indexed instance
		$IVar[Var].EVar(Var)\ \ WithIndex\ S_1$ $WithAttributes\ S_2$	input event w/attribute at indexed instance

Notes:

- Although we allow instance tables to be indexed by any type, typically we only use fixed-size records, because they are easier to hash.
- There may be at most one instance with no input events other than `Init`. This instance is called the capture instance and is executed at the beginning of each round.

Examples:

```
event C.Ofoo(or) -> I.Foo(ir) WithAttributes {ir.x = or.x;
                                              ir.y = or.y};

event C.Ofoo(or) -> J[idx].Foo(ir) WithIndex {idx = ir.x}
                                   WithAttributes {ir.x = or.x;
                                                   ir.y = or.y};
```

See Also: Recognizer Definitions (A.11), Events (A.8), Instance Declarations (A.12.2)

A.14 Instance Output Event Patterns

In the main module, we take output events of one instance and feed it as input to another. To identify whether an output event has occurred, we use event patterns that match both the instance index and the event attribute.

IEv	::=	$IVar.EVar$	output event at instance
		$IVar[Var].EVar$	output event at indexed instance
		$IVar.EVar(Var)$	output event with attribute at instance
		$IVar[Var].EVar(Var)$	output event with attribute at indexed instance
		IEv `OccurredWhen` E	IEv occurred when E was true

Examples:

```
recognizer Q : {
  input event int Init;
  input event string Error
};

instance Q K(init) WithAttributes {init = 0};

I.Bar -> K.Error(e) WithAttributes {e = ''bar''};
I.Bar(n) OccurredWhen (n == 0) -> K.Error(e) WithAttributes
                                        {e = ''bar with 0''};
J[idx].Bar(n) OccurredWhen (idx != n) -> K.Error  WithAttributes
                                        {e = ''bar with bad n''};
```

See Also: Recognizer Definitions (A.11), Event Patterns (A.9), Instance Declarations (A.12.2)

A.15 Event Forwarding Definitions

Event forwarding definitions take output events from one instance and forward it as input to another. They first check whether an output event has occured, and if yes, then they trigger the input event of an instance and execute one round of the instance.

F	::=	event $IEv \rightarrow V_I$	output IEv forwarded to input V_I
		event $IEv \rightarrow$ PRINT	output IEv forwarded to stdout

Examples:

```
event C.Ofoo(or) -> I.Foo(ir) WithAttributes {ir.x = or.x;
                                              ir.y = or.y};

event C.Ofoo(or) -> J[idx].Foo(ir) WithIndex {idx = ir.x}
                                   WithAttributes {ir.x = or.x;
                                                   ir.y = or.y};

I.Bar(n) OccurredWhen {n == 0} -> PRINT;
J[idx].Bar(n) OccurredWhen (idx != n) -> PRINT;
```

See Also: Recognizer Definitions (A.11), Event Definitions (A.10.1),

Instance Declarations (A.12.2)

A.16 Main Module Definitions

The main module declares all the instances in the monitoring stack, and contains a sequence of event forwarding definitions that implements the bottom-to-top execution of the stack.

M	::=	begin $(TyDec;) * (SigDec;) * (IDec;) * (F;) * F$ end

Examples:

```
begin
typedef {int x; int y} ty_rec;

recognizer Cap: {input event string Init;
                 output event ty_rec Ofoo;};

recognizer R: {input event int Init;
               input event ty_rec Foo;
               output event int Bar}
```

```
instance Cap C(init) WithAttributes {init = ``blah''};
instance R I(init) WithAttributes {init = 0};

event C.Ofoo(or) -> I.Foo(ir) WithAttributes {ir.x = or.x;
                                              ir.y = or.y};

event I.Bar(n) OccurredWhen {n == 0} -> PRINT;
end
```

See Also: Main Module Declarations (A.12), Event Forwarding Definitions (A.15)

A.17 Run-time Errors

NERL is strongly typed. At compile-time, the static checker finds all violations of the grammar rules, all out-of-scope variables, and most kinds of ill-typed expressions. However, some type rules can be checked only at run time, such as array bounds violations. At run-time, a NERL program may raise one of the following errors:

ArrBndErr	Array bounds violation
EmptyArr	Deleting cells from an empty array
WriteErr	Writing to a location twice in one round
RepeatedEvt	Generating two instances of an event in one round
BadLocation	Accessing a location that does not exist
Error	Unexpected Error

The first two errors assert that an array is being used incorrectly. The last two denote artificial restrictions on the run-time behavior of recognizers that, in our opinion, make the recognizer easier to read and understand. The third error asserts that a variable has been modified more than once in the same round. The fourth error asserts that the same event has been generated more than once at the same recognizer instance in the same round. The last two errors should never occur in a NERL program and signify an error in the compiler.

APPENDIX B

TRACE SEARCH ALGORITHMS WITH PROOFS OF CORRECNESS

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 4, we described how NERL can be used to program such a recognizer for the ping protocol. However, as we observed in Section 2.3.1, 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 chapter, 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 it is generated and received by the network

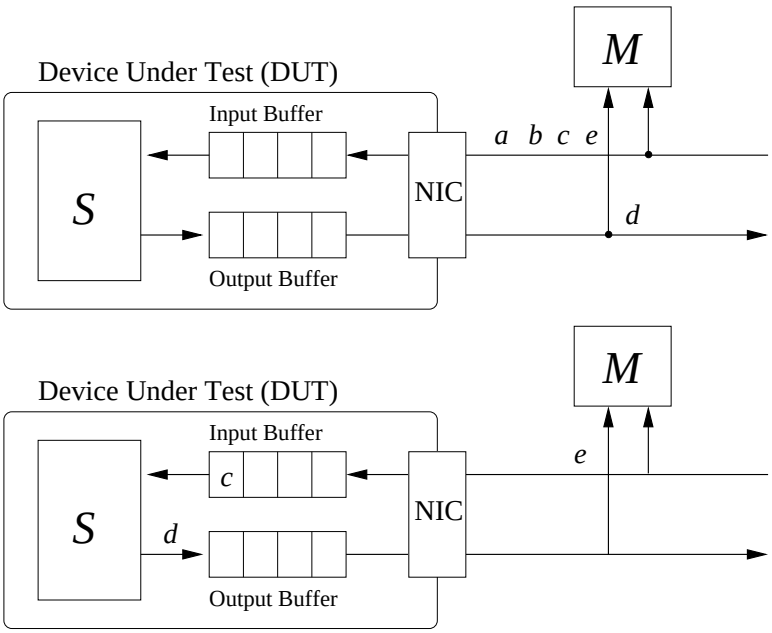

Figure B.1 Co-networked Monitoring

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. This configuration is depicted in Figure B.1.

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 (like *c* in Figure B.1). 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 protocol monitor 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 behaviors that are caused by the input buffer.

In the rest of this chapter, we develop an algorithm that transforms recognizers that assume perfect fidelity (co-located monitors) to recognizers that assume input buffering and input loss. In Section B.1, we formalize the effect of trace infidelity on the monitored packet traces in the co-networked monitoring model. Then, in Section B, we describe the trace search algorithm and sketch a proof of correctness. Finally, Section B.3 observes some limitations of the model and possible extensions. The work presented in this chapter was co-authored by Dr. Satish Chandra, Dr. Peter McCann, and Prof. Carl Gunter, and has been published elsewhere (BCMG01).

B.1 Co-networked Monitor Model

Assume that we want to monitor a protocol with one input channel and one output channel (like the ping signature in Chapter 4). At each step, 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 *i*nput symbol a in *q*ueue I; id_a corresponds to deletion of a from head of I and simultaneous *i*nput into *d*evice; od_a corresponds to *o*utput of a from *d*evice. 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 of 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 B.1 (Admissibility) *ω, 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 ω, the number of iq tokens minus the number of id tokens must not exceed m.*

According the the admissibility model, there are three sequences that a co-networked recognizer needs to be aware of

- τ: the *iq/od* sequence observed by the monitor,
- σ: the *id/od* sequence that actually occured at the device, and
- ω: the *iq/id/od* sequence that represents the execution of the device-monitor system and contains both τ and σ as projections.

Since the monitor can only see τ, it must reconstruct the system execution ω and extract the device execution sequence σ. Admissibility defines the set of possible system execution sequences (ω) 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 τ belongs to L_M, if there is some admissible sequence ω whose *iq/od* projection is τ, and whose *id/od* projection belongs to L_S. Formally:

Definition B.2 *The* co-networked monitor language *accepted by $M(S,m)$ is defined as* $L_M = \{\tau \mid \exists\omega : \omega_{iq,od} = \tau \ \wedge \ \omega_{id,od} \in L_S \ \wedge \ \omega \ admissible \ w.r.to \ M(S,m)\}$.

For each string s in L_S, we can construct several τ that must be in L_M. Let us first construct an admissible sequence whose *id/od* projection is s. We take each input token i_a in s and split it into two consecutive tokens iq_a and id_a. We 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 ω yields a string τ in the language recognized by M (L_M), simply by erasing out 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, etc., and instead label the various tokens by order of their appearance in the string, using the notation id^k for the k'th occurrence of *id*.

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

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_4.

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

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).

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

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

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

The monitor language L_M defines the extent to which a co-networked monitor must *relax* the monitored properties to allow for input buffering. If the device produces a correct sequence then the co-networked monitor will accept it, irrespective of the behavior of the input buffer. Formally:

Theorem B.3 *Suppose a device with an input buffer of size m produces an execution sequence σ, and the trace observed at the co-networked monitor is τ. If $\sigma \in L_S$, then $\tau \in L_M$.*

Proof. The proof is straight-forward to prove by expanding the definition of L_M. If σ was produced by the device and τ was observed at the monitor, then thre must be an execution sequence for the complete system (ω) that contains both σ and τ and is admissible with respect to the buffers. Instantiating this ω in the definition of L_M we get that $\tau \in L_M$. □

The above theorem ensures that if we observe a τ that is not in L_M then it will certainly represent an error in the device ($\sigma \notin L_S$). On the other hand, the inability of the monitor to see σ directly means that it cannot distinguish between two device execution sequences that are equivalent up to buffering:

Definition B.4 *Device execution sequences σ_1 and σ_2 are said to be buffer-equivalent with respect to an input buffer of size m and observed trace τ ($\sigma_1 \sim_{\tau,m} \sigma_2$) if $\{\omega \mid \omega_{id,od} = \sigma_1 \wedge \omega_{iq,od} = \tau \wedge \omega\ admissible\ w.r.to\ M(S,m)\} = \{\omega \mid \omega_{id,od} = \sigma_2 \wedge \omega_{iq,od} = \tau \wedge \omega\ admissible\ w.r.to\ M(S,m)\}$.*

For the above definitions and theorems to work, the monitor must know the size of the input buffer at the device. This is not as difficult as it sounds, because an over-estimation

is always safe - it only increses the set of ω examined. Moreover, if we under-estimate the buffer and find that $\tau \in L_M$ then even for a larger buffer this will be true (the reverse is not true). The safety of buffer over-estimation if formalized by the following lemma:

Lemma B.5 *If ω is admissible w.r.to $L(S,m)$, then it is admissible w.r.to $L(S,m')$ where $m' > m$.*

Proof. This lemma follows directly form 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 ω that are admissible with respect to $M(S,m)$, we are interested in the maximum size of this set.

Theorem B.6 *Suppose that the monitor observes a trace τ, and the input buffer has size m. Then the total number of ω that can be constructed from τ ($\omega_{iq,od} = \tau$) such that they are also admissible with respect to $M(S,m)$ is less than $C(N_{iq}) * (m+1)^{N_{od}}$, where N_{iq} is the number of inputs in τ, N_{od} is the number of outputs in τ, and $C(n)$ is the nth Catalan number:*

$$\frac{\binom{2n}{n}}{n+1}$$

Proof. Note that every ω that is admissible w.r.to $M(S,m)$ is constructed by inserting *id* tokens between the *iq* and *od* tokens. First, each ω 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 ω must also follow the causality clause - so the number of *id* tokens can never exceed the number of *iq* tokens in any prefix of ω.

So one can view the inputs in ω as a path in a two-dimensional grid from $(0,0)$ to (N_{iq}, N_{iq}), where the x-dimension refers to *iq*s and the y-dimension refers to *id*s. This path must never go above the major diagonal ($x = y$) because then the number of *id*s will exceed the number of *iq*s. The maximum number paths in an $n \times n$ grid from $(0,0)$ to (n,n) without corssing the major diagonal is known to be given by the nth 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 $x >= y$ and $x <= y + m$. 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 $2^{N_{iq}-1}$. So even with an input buffer of size 1, the number of ω the co-networked monitor needs to reconstruct may be as great as $(2^{N_{iq}-1} * 2^{N_{oq}})$ or $O(2^{|\tau|})$.

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

Example. Consider the following sequence:

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

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

$$i_b\ o_d$$

Whereas, the monitor observes the following tokens:

$$iq_a\ iq_b\ iq_c\ od_d$$

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 B.7 (Admissibility) *ω, 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 ω 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 ω, the number of iq tokens minus the number of id/il tokens must not exceed m, and*
- **Input Loss Limit:** *in any prefix of ω, the number of consecutive il's without an intervening id token must not exceed l.*

The definition of L_M also changes accordingly:

Definition B.8 *The* co-networked monitor language *accepted by $M(S, m, l)$ is defined as $L_M = \{\tau \mid \exists \omega : \omega_{iq,od} = \tau \ \wedge \ \omega_{id,od} \in L_S \ \wedge \ \omega \ admissible \ w.r.to \ M(S, m, l)\}$.*

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.

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

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

Theorem B.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 σ, and the trace observed at the co-networked monitor is τ. 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, approzimating the properties even further:

Definition B.10 *Device execution sequences σ_1 and σ_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 τ ($\sigma_1 \sim_{\tau,m,l} \sigma_2$) if $\{\omega \mid \omega_{id,od} = \sigma_1 \ \wedge \ \omega_{iq,od} = \tau \ \wedge \ \omega \ admissible \ w.r.to \ M(S,m,l)\} \ = \ \{\omega \mid \omega_{id,od} = \sigma_2 \ \wedge \ \omega_{iq,od} = \tau \ \wedge \ \omega \ admissible \ w.r.to \ M(S,m,l)\}$.*

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

Lemma B.11 *If ω is admissible w.r.to $L(S,m,l)$, then it is admissible w.r.to $L(S,m',l')$ 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 ω than before.

Theorem B.12 *Suppose that the monitor observes a trace τ, the input buffer has size m, and the loss module has a limit l. Then the total number of ω that can be constructed from τ ($\omega_{iq,od} = \tau$) such that they are also admissible with respect to $M(S,m,l)$ is less than $2^{N_{iq}} * C(N_{iq}) * (m+1)^{N_{od}}$, where N_{iq} is the number of inputs in τ, N_{od} is the number of outputs in τ, and $C(n)$ is the nth Catalan number:*

$$\frac{\binom{2n}{n}}{n+1}$$

Proof. Since now we have a loss module, only a subset of the *iq* tokens translate into *id* tokens, the rest become *il*s. 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 ω exactly as it there were only an input buffer but treating *id* and *il* 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 ω that the monitor needs to keep track may still grow to $O(2^{|\tau|})$. Since this is already too large, the precise number of possible ω sequences is not that relevant. Hereafter we shall only refer to an "exponential" sized admissible set.

B.2 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 τ of *iq* and *od* events The device is not incorrect as long as $\tau \in L_M$: we can exhibit a sequence ω which is derived from τ by the addition of *id* and *il* tokens and which is consistent with the following set of conditions:

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

Suppose that g is the colocated recognizer for S. So g checks S on a sequence α 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 tranformation 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 τ 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 τ, guess a sequence ω admissible with respect to $M(S, m, l)$, such that $[\omega]_{iq,oq} = \tau$. If $[\omega]_{id,od}$ satisfies g, τ is OK. Otherwise, report failure.

A deterministic version of the above algorithm is brute-force search: simply construct all possible ω's from τ, checking each admissible ω against g, until a match is found or all possibilities are exhausted. Additionally, we would like F to be computable *on-line*, meaning that it should make only one pass over the input τ. We give such a brute-force breadth-first-search algorithm in Table B.1. 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 ω and checks it against g. The sequences it allows are exactly those in the co-networked monitoring language:

Theorem B.13 *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:

Lemma B.14 *If g is a co-located monitor for S, then $BF(g, \tau, m, l)$ produces a set Ω containing* exactly all *the strings ω that satisfy the following:*

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

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

C3 *ω is admissible (with respect to $M(S, m, l)$).*

Table B.1 Trace Search Algorithm for Co-networked Monitoring

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

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

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

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

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

If the $n+1$'th token in τ is iq_a, then the algorithm first adds this token to every ω 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 Ω, 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 ω. 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 Ω. 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 effient to use the particular features of a co-located recognizer to manually transform it to run in a co-networked environment.

B.3 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 occurence 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-deteministic, 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 stratiegies for co-networked monitoring. several of the features of our co-networked recognizer.

B.4 Proof of Counting Property Optimization

In this section we prove the correctness of the optimized co-monitoring algorithm (Table 8.9.3) that was used to check that in each TCP session, at most two DATA packets were sent between each ACK. This property falls in a class that 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 $c_{\min}$ and $c_{\max}$ inputs. For TCP, $c_{\min} = 0$ and $c_{\max} = 2$. Suppose that input channels have a buffer size of B, and output channels are unbuffered, and assume that no packets are lost in the input buffer. Then, we can design a special recognizer for checking the counting property in the presence of buffering. The recognizer implements the algorithm shown in Table B.2.

This algorithm maintains two integers, $buf_{\min}$ and $buf_{\max}$, representing the minimum and maximum number of inputs that are currently buffered on the channel between the monitor and the device under test. If $buf_{\min}$ ever grows too large, it indicates that the particular iq, od 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

Table B.2 Trace Search Algorithm for Counting Properties

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

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

Event Handlers. On receiving

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

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

inputs seen so far. Similarly, if $buf_{\max}$ ever becomes too small, it indicates that the particular iq, od 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 (by raising 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 B.2 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 Table B.1 to the counting algorithm in Table B.2. 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 B.15 (iqtail) *Let* iqtail(ω) *equal the sequence of iq elements in* ω *that are prior to the last oq element but which were not consumed before the oq, that is, which do not have a corresponding id element before the oq, plus all iq's that appear after the last oq element,*

regardless of whether the corresponding id is in ω*. If* ω *does not contain any oq's, then* iqtail(ω) *is just the sequence of iq's in* ω*.*

Definition B.16 (idtail) *Let* idtail(ω) *equal the sequence of id elements in* ω *that appear after the last od element. If* ω *does not contain any od's, then* idtail(ω) *is just the sequence of id's in* ω*.*

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

$$\neg\, e \Longrightarrow (\forall i :: \mathit{buf}_{\min} \leq i \leq \mathit{buf}_{\max} \Longleftrightarrow$$
$$(\exists(\omega, b) \in \Omega :: |\text{iqtail}(\omega)| = i))$$
$$\wedge \quad e \Longleftrightarrow \Omega = \phi$$

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

Inductive Step. Consider the input actions in each algorithm corresponding to *iq* events. In Table B.1, this action deletes each member of Ω and adds a new member with the *iq* action added to both ω and b, as long as ω is admissible and its projection $[\omega]_{\mathit{id,od}}$ is in g. In Table B.2, an input event increments both $\mathit{buf}_{\min}$ and $\mathit{buf}_{\max}$, checks for an error condition, and then adjusts $\mathit{buf}_{\max}$ if necessary. We need to show that the state space mapping still holds. First, note that if any $(\omega, b) \in \Omega$ has an ω such that $|\text{iqtail}(\omega)| >= B + c_{\max}$, it will be deleted from Ω because the result of adding the *iq* to ω will be inadmissible according to M', which recall is equal to $M(S, m{+}c{*}n, 0)$. In this case $c = c_{\max}$ and $B = m{+}c{*}n$. ω will be inadmissible because at most $c_{\max}$ *id*'s can appear after the last *od*, *oq* by way of property P1. The remaining B *iq*'s fill up the buffer and adding one more causes it to overflow, removing the resulting string from consideration.

If $\mathit{buf}_{\min} >= B + c_{\max}$ before the input action, this implies (by the inductive hypothesis) that every $(\omega, b) \in \Omega$ has a iqtail(ω) whose size is at least $B + c_{\max}$. When the *iq* event is added, all will be deleted and Ω will be empty. This corresponds to setting e to *true* in Table B.2.

If $\mathit{buf}_{\min} < B + c_{\max}$ before the input action, then there is at least one $(\omega, b) \in \Omega$ such that $|\text{iqtail}(\omega)| = i$ for every i between $\mathit{buf}_{\min}$ and $\mathit{buf}_{\max}$. Each will be removed and replaced with an (ω, b) pair with exactly one *iq* event added to each of ω and b, except for those that become inadmissible with respect to M'. This corresponds exactly to $\mathit{buf}_{\min}$ and $\mathit{buf}_{\max}$ being incremented by one, and $\mathit{buf}_{\max}$ being capped at $B + c_{\max}$ in Table B.2. The iteration step 2 in Table B.1 has no effect on the mapping because it only adds *id* events to ω and does not modify iqtail(ω). Note, however, that this iteration guarantees that every pattern of idtail(ω) such that $0 \leq |\text{idtail}(\omega)| \leq \min(c_{\max}, \mathit{buf}_{\max})$ will be generated. This is because each element of the iqtail(ω) will be converted to an *id* and check-added to Ω. Those with more than $c_{\max}$ *id*'s will be discarded, since they are not in g, but all others will be accepted due to property P1. The longest iqtail has length $\mathit{buf}_{\max}$, leading to the upper bound.

Next, consider the output actions in each algorithm corresponding to the *oq* events. In Table B.1, this action first deletes every element of Ω, replacing each with one that has the *od*, *oq* event pair added. Then the same iteration discussed previously is repeated. Note that if any (ω, b) has an idtail$(\omega) < c_{\min}$, it will be deleted because the result of adding the od will not be in g, according to P1. In Table B.2, if $\mathit{buf}_{\max} < c_{\min}$, then all idtail$(\omega)$ must have length less than $c_{\min}$, because idtail$(\omega) <$ iqtail$(\omega) < \mathit{buf}_{\max}$. Ω thus becomes empty after the first step. This corresponds to setting e to true in Table B.2. Otherwise, Table B.2 decrements $\mathit{buf}_{\max}$ by $c_{\min}$, but caps $\mathit{buf}_{\max}$ at B because even though iqtail can grow larger than B, the new iqtail will be at most B because it consists of only the un-consumed *iq* events in ω, of which there can be at most B. The algorithm also decrements $\mathit{buf}_{\min}$ by $c_{\max}$, setting it to zero if it goes negative. This corresponds to the fact that pairs (ω, b) will be deleted from Ω if they violate the $c_{\min}$ and $c_{\max}$ limits given by g, which means that the new minimum and maximum values of iqtail(ω) will be given when the maximum number of inputs are consumed from the minimum previous *iqtail* and the maximum number of inputs are consumed from the maximum previous *iqtail*. These consumptions are justified because we have a guarantee that

every pattern idtail(ω), such that $0 \leq |\text{idtail}(\omega)| \leq \min(c_{\max}, \textit{buf}_{\max})$, will have been generated at the end of the previous *iq* or *oq* event.

We have established a state space mapping between the trace search algorithm and a specialized algorithm for counting properties. This mapping asserts that the specialized algorithm raises an error e if and only if the trace-search algorithm raises an error as well ($\Omega = \phi$). Therefore, the specialized algorithm is equivalent to the general algorithm for the purpose of monitoring counting properties. This completes our proof.

APPENDIX C

PING EVENT RECOGNITION

This appendix contains code listings for the Ping protocol as described in Chapter 4. Section C.1 lists a recognizer module for Ping. This module differs slightly from the one in Figure 4.7; it checks additional preconditions on the incoming events. In particular, it remembers the parameters of the monitored session and only accepts requests and replies if they belong to the session. The NERL compiler translates this program to a function in the C programming language. Section C.2 lists this translation.

Section C.3 lists a main module for Ping. It also differs slightly from the corresponding module in Figure 4.9; the PCap module is divided into an IP capture module and an ICMP parser module. Section C.4 lists the translation of this main module into a C executable program.

Finally, Section C.5 lists the Promela code for the ping configuration, and includes the properties that are checked for this configuration. It contains the Promela model for the sender, receiver, and the Promela translation of the ping recognizer.

C.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)
                                            WithAttributes
                                            {a = e.icmp_data};
  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;
```

C.2 Ping Recognizer translated to C: ping.c

```
#include "includes.h"

/*   Type Definitions   */

#define CELL(t) struct{t value;}
typedef CELL(bool) boolcell;
typedef CELL(int) intcell;
typedef CELL(double) doublecell;
typedef CELL(string) stringcell;

#define EVENT_TY(a) struct {bool flag; int timestamp; 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 {
  int now;

  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->now = cb->inputs->Init.timestamp;

    cb->state->ip_src = (intcell*) malloc(sizeof(intcell));
    cb->state->ip_src->value = 0;

    cb->state->ip_dst = (intcell*) malloc(sizeof(intcell));
    cb->state->ip_dst->value = 0;

    cb->state->icmp_id = (intcell*) malloc(sizeof(intcell));
    cb->state->icmp_id->value = 0;

    cb->state->status = (intcell*) malloc(sizeof(intcell));
    cb->state->status->value = 0;

    cb->state->icmp_seq = (intcell*) malloc(sizeof(intcell));
    cb->state->icmp_seq->value = 0;

    cb->state->icmp_data = (stringcell*) malloc(sizeof(stringcell));
    cb->state->icmp_data->value = "";
  }

  /* Round Begins */

  /*   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) {
  cb->state->now = cb->inputs->Init.timestamp;
  ty_ping* i;
  i= (cb->inputs->Init.attrib);
  {cb->state->ip_src->value = i->ip_src->value;
  cb->state->ip_dst->value = i->ip_dst->value;
  cb->state->icmp_id->value = i->icmp_id->value;
  cb->state->status->value = 0;
  }
}

if (cb->inputs->EchoRequest.flag == true) {
  cb->state->now = cb->inputs->EchoRequest.timestamp;
  ty_echo* p;
  p = (cb->inputs->EchoRequest.attrib);
  if ((p->ip_src->value == cb->state->ip_src->value) &&
```

```
      (p->ip_dst->value == cb->state->ip_dst->value) &&
      (p->icmp_id->value == cb->state->icmp_id->value) &&
      (p->icmp_code->value == 0)  &&
      (p->icmp_type->value == 8)) {
    cb->locals->MyEchoRequest.flag = true;
    cb->locals->MyEchoRequest.timestamp = cb->state->now;
  }
}

if (cb->inputs->EchoReply.flag == true) {
  cb->state->now = cb->inputs->EchoReply.timestamp;
  ty_echo* p;
  p= (cb->inputs->EchoReply.attrib);
  if ((p->ip_src->value == cb->state->ip_dst->value) &&
      (p->ip_dst->value == cb->state->ip_src->value) &&
      (p->icmp_id->value == cb->state->icmp_id->value) &&
      (p->icmp_code->value == 0)  &&
      (p->icmp_type->value == 0)) {
    cb->locals->MyEchoReply.flag = true;
    cb->locals->MyEchoReply.timestamp = cb->state->now;
  }
}

if (cb->inputs->EchoRequest.flag == true) {
  cb->state->now = cb->inputs->EchoRequest.timestamp;
  ty_echo* e;
  e= (cb->inputs->EchoRequest.attrib);
  if (cb->locals->MyEchoRequest.flag == true) {
    if (cb->state->status->value == 0) {
      cb->state->icmp_seq->value = e->icmp_seq->value;
      string_copy(&(cb->state->icmp_data->value),e->icmp_data->value);
      cb->state->status->value = 1;
    }
  }
}

if (cb->inputs->EchoRequest.flag == true) {
  cb->state->now = cb->inputs->EchoRequest.timestamp;
  ty_echo* e;
  e= (cb->inputs->EchoRequest.attrib);
  if (cb->locals->MyEchoRequest.flag == true) {
    if (cb->state->status->value != 0) {
      cb->outputs->BadEchoRequest.flag = true;
      cb->outputs->BadEchoRequest.timestamp = cb->state->now;
      {stringcell* b;
      b= cb->outputs->BadEchoRequest.attrib;
      {b->value = "Multiple Echo Requests";}}
    }
  }
}

if (cb->inputs->EchoReply.flag == true) {
  cb->state->now = cb->inputs->EchoReply.timestamp;
  ty_echo* e;
  e= (cb->inputs->EchoReply.attrib);
```

```
  if (cb->locals->MyEchoReply.flag == true) {
    if ((cb->state->status->value == 1)  &&
        (e->icmp_seq->value == cb->state->icmp_seq->value)  &&
        ((e->icmp_data->value.length == cb->state->icmp_data->value.length)?
           (strncmp(e->icmp_data->value.value,
                    cb->state->icmp_data->value.value,
                    e->icmp_data->value.length) == 0): false){
      cb->outputs->IsAlive.flag = true;
      cb->outputs->IsAlive.timestamp = cb->state->now;
      {stringcell* a;
      a= cb->outputs->IsAlive.attrib;
      {string_copy(&(a->value),e->icmp_data->value);}
    }
  }
}

if (cb->inputs->EchoReply.flag == true) {
  cb->state->now = cb->inputs->EchoReply.timestamp;
  ty_echo* e;
  e= (cb->inputs->EchoReply.attrib);
  if (cb->locals->MyEchoReply.flag == true) {
    if ((cb->state->status->value != 1) ||
        (e->icmp_seq->value != cb->state->icmp_seq->value) ||
       !((e->icmp_data->value.length == cb->state->icmp_data->value.length)?
           (strncmp(e->icmp_data->value.value,
                    cb->state->icmp_data->value.value,
                    e->icmp_data->value.length) == 0): false){
      cb->outputs->BadEchoReply.flag = true;
      cb->outputs->BadEchoReply.timestamp = cb->state->now;
      {stringcell* b;
      b= cb->outputs->BadEchoReply.attrib;
      {b->value = "Incorrect Echo Reply";}}
    }
  }
}

if (cb->outputs->IsAlive.flag == true) {
  cb->outputs->Done.flag = true;
  cb->outputs->Done.timestamp = cb->state->now;
  {boolcell* d;
  d= cb->outputs->Done.attrib;
  {d->value = true;}
  }
} else
  if (cb->outputs->BadEchoReply.flag == true) {
    cb->outputs->Done.flag = true;
    cb->outputs->Done.timestamp = cb->state->now;
    {boolcell* d;
    d= cb->outputs->Done.attrib;
    {d->value = true;}
    }
  } else
    if (cb->outputs->BadEchoRequest.flag == true) {
      cb->outputs->Done.flag = true;
      cb->outputs->Done.timestamp = cb->state->now;
```

```
        {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->value = 2;
    }
  }
}
```

C.3 Ping Main Module in NERL: pingmod.nerl

```
begin

 typedef {
   string filename;
 } file;

 recognizer PCap : {
   input event file Init;
   output event Ty_ippkt IP;
   output event bool Done;
 }

 recognizer ICMP_Parse : {
   input event bool Init;
   input event Ty_ippkt IP;
   output event Ty_echo ICMP_Echo;
 }

 recognizer Ping : {
  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;
 }


 instance PCap        P  WithAttributes
                         {init = "icmp.pcap"};
```

```
 instance ICMP_Parse  I   WithAttributes
                          {init = true};
 instance Ping        E[Ty_ping c]
                          WithAttributes
                          {init = c};

 #define REQUEST  8
 #define REPLY    0

 event P.IP(p) OccurredWhen (p.ip_p == 1) ->
       I.IP(q) WithAttributes {q = p};

 event I.ICMP_Echo(e) OccurredWhen (e.type == REQUEST) ->
       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 == REPLY) ->
       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
```

C.4 Ping Main Module Translated to C: pingmod.c

```
/* Index Type Definitions */

typedef struct {
  stringcell* filename;
} file;

/* Signatures */

typedef struct {
  EVENT_TY(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() {

  int roundnumber;

  /* 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 */

  roundnumber = 0;

  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) {
  roundnumber++;
  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_ippkt* 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 occured - ",
                E_cb->outputs->BadEchoReply.timestamp);
      printf("BadEchoReply:");
      print_string(E_cb->outputs->BadEchoReply.attrib->value);
      printf("\n");
    }

    if (E_cb->outputs->BadEchoRequest.flag == true) {
      printf("%f: Event BadEchoRequest has occured - ",
                E_cb->outputs->BadEchoRequest.timestamp);
      printf("BadEchoRequest:");
      print_string(E_cb->outputs->BadEchoRequest.attrib->value);
      printf("\n");
    }

    if (E_cb->outputs->IsAlive.flag == true) {
      printf("%f: Event IsAlive has occured - ",
                E_cb->outputs->IsAlive.timestamp);
      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->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->EchoReply.flag = true;
          {ty_echo* p;
          p = e;
          E_cb->inputs->EchoReply.attrib = p;
          Ping_recognize(E_cb);
          }

          if (E_cb->outputs->BadEchoReply.flag == true) {
            printf("%f: Event BadEchoReply has occured - ",
                      E_cb->outputs->BadEchoReply.timestamp);
            printf("BadEchoReply:");
            print_string(E_cb->outputs->BadEchoReply.attrib->value);
            printf("\n");
          }

          if (E_cb->outputs->BadEchoRequest.flag == true) {
            printf("%f: Event BadEchoRequest has occured - ",
                      E_cb->outputs->BadEchoRequest.timestamp);
            printf("BadEchoRequest:");
            print_string(E_cb->outputs->BadEchoRequest.attrib->value);
            printf("\n");
          }

          if (E_cb->outputs->IsAlive.flag == true) {
            printf("%f: Event IsAlive has occured - ",
                      E_cb->outputs->IsAlive.timestamp);
            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);
          }
        }
      }
    }
  }
}
```

C.5 Ping Promela Model: ping.spin

We list two files: `pingmain.spin`, containing models for the sender and receiver, and `ping.spin`, containing the Promela model generated from `ping.nerl`.

```
/*******************************************************/
/*              pingmain.spin                          */
/*******************************************************/

#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;
  boolcell d;
  intcell 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;
            fi;
            if
            :: cb_table[0].outputs.BadEchoReply?[a] -> recbadechoreply = true;
            :: else -> skip;
            fi;
            if
            :: cb_table[0].outputs.Done?[d] -> recdone = true;
            :: else -> skip
            fi
        }
    }
}

/* Properties checked:
   Correct meta event:
           <>(recisalive == true)
           [](recisalive == true) -> <>(isalive == true))

   No false positives:
           [](recbadechorequest == false)
           [](recbadechorequest == false)
*/

/*******************************************************/
/*              ping.spin                              */
/*******************************************************/

/*   Type Definitions   */

typedef intcell {
  int value;
};

typedef boolcell {
  bool value;
};

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;
};

/*   State Declarations   */

typedef Ping_state {
  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;
};

/*   Event Declarations   */

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 */

proctype Ping_recognize(int me) {
  ty_ping i;
  ty_echo p;
  ty_echo e;
  boolcell d;
  intcell 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 */

  /*   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.value = i.ip_src.value;
        cb_table[me].state.ip_dst.value = i.ip_dst.value;
        cb_table[me].state.icmp_id.value = i.icmp_id.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.value) &&
            (p.ip_dst.value == cb_table[me].state.ip_dst.value) &&
            (p.icmp_id.value == 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.value) &&
            (p.ip_dst.value == cb_table[me].state.ip_src.value) &&
            (p.icmp_id.value == 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.value == 0 ->
                cb_table[me].state.icmp_seq.value = e.icmp_seq.value;
                cb_table[me].state.icmp_data.value = e.icmp_data.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.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] ->
      if
      :: (cb_table[me].state.status.value == 1) &&
         (e.icmp_seq.value ==  cb_table[me].state.icmp_seq.value)  &&
         (e.icmp_data.value ==  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] ->
      if
      :: (cb_table[me].state.status.value != 1) ||
         (e.icmp_seq.value != cb_table[me].state.icmp_seq.value) ||
         (e.icmp_data.value != 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.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 D

AODV EVENT RECOGNIZER

The following is a code listing for the AODV recognizer, `aodv.nerl`. This recognizer was used for the simulation analysis in Chapter 6.

```
Recognizer AODV =

/* Type/Struct Definitions, typically for event attributes */

#define NODES 51
#define INFINITY 255

typedef {
  int eventty;
  int pktty;
  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 ttl;
  int life
} ty_pkt;

typedef {
} ty_basic;

typedef {
```

```
  int at;
  int dst;
  int best_seq;
  int best_hc;
  int best_next
} ty_cell;

typedef {
  int node;
  int next;
  int dest;
  int seq_at_node;
  int hc_at_node;
  int seq_at_next;
  int hc_at_next
} ty_loopinfo;

/* Input Events */

input event ty_pkt Pkt ;
input event ty_basic Init ;
input event ty_basic Finish ;

/* Output Alarms */

output event ty_cell NotSeqMono;
output event ty_cell DestForwards;
output event ty_cell BadFwd;
output event ty_cell BadDestRep;
output event ty_cell BadNodeRep;
output event ty_cell BadRReq;
output event ty_cell BadRerr;
output event ty_cell BadRerrFwd;
output event loopinfo LoopInvFails;

/* Private  Local Events */

event ty_cell Routeinfo;
event ty_cell Recvroute;
event ty_cell Recvbetter;
event ty_cell Sendroute;
event ty_cell Send;
event ty_cell Sendnext;
event ty_cell Sendreq;
event ty_cell Incseq;
event ty_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 Pkt(p)
      OccurredWhen
         (p.pktty > 0) -> Routeinfo(r)
                          WithAttributes {
                            r.at = p.atnode;
                            r.dst = p.src;
                            r.best_seq = best_seq[p.atnode][p.src];
                            r.best_hc = best_hc[p.atnode][p.src];
                            r.best_next = best_next[p.atnode][p.src]};

event (Pkt(p) & Routeinfo(r))
       OccurredWhen
          (p.eventty == 0) -> Recvroute (rcv)
                              WithAttributes {rcv = r};

event (Pkt(p) & Recvroute(rcv))
       OccurredWhen
          (p.src_seq > best_seq[rcv.at][rcv.dst]) ||
          ((p.src_seq == best_seq[rcv.at][rcv.dst]) &&
           (p.src_hc < best_hc[rcv.at][rcv.dst])) ->
                              Recvbetter(rcvb)
                                WithAttributes {rcvb = rcv};

event (Pkt(p) & Routeinfo(r))
       OccurredWhen
          (p.eventty == 1) &&
          (p.src_hc < INFINITY) -> Sendroute(snd)
                                   WithAttributes
                                   {snd = r};

event (Pkt(p) & Routeinfo(r))
       OccurredWhen
          ((p.eventty == 1) &&
           (p.pktty == 2) &&
           (p.src_hc >= INFINITY)) -> Senderr(serr)
                                      WithAttributes
                                      {serr = r};

event (Pkt(p) & Sendroute(snd))
        OccurredWhen
           ((snd.dst == snd.at) &&
            (p.src_seq > best_seq[snd.at][snd.at])) ->
                                      Incseq(is)
                                      WithAttributes
                                      {is = snd};

event Pkt
       OccurredWhen
        (p.eventty == 1) -> Send(s)
                            WithAttributes {
                              s.at = p.atnode;
                              s.dst = p.dest;
                              s.best_seq = best_seq[p.atnode][p.dest];
```

```
                                s.best_hc = best_hc[p.atnode][p.dest];
                                s.best_next = best_next[p.atnode][p.dest]};

event (Pkt(p) & Send(s))
       OccurredWhen
        ((p.pktty == 0) ||
         (p.pktty == 2))   -> Sendnext (snext)
                                WithAttributes {snext = s};

event (Pkt(p) & Send(s))
       OccurredWhen
       (p.pktty == 1) -> Sendreq(sreq)
                         WithAttributes {sreq = s};

/* Output (error) Event Definitions: Conformance Check */

event (Pkt(p) & Sendroute(snd))
       OccurredWhen
        (snd.dst == snd.at) &&
        (p.src_seq < best_seq[snd.at][snd.at]) -> NotSeqMono (nsm)
                                                  WithAttributes
                                                  {nsm = snd};

event Sendnext(snext)
       OccurredWhen
         (snext.at == snext.dst) -> DestForwards (d)
                                    WithAttributes {d = snext};

event (Pkt(p) & Sendnext(snext))
       OccurredWhen
         (p.next_hop != best_next[snext.at][snext.dst])
                                 -> BadFwd (bf)
                                    WithAttributes {bf = snext};

event (Pkt(p) & Sendroute(snd))
       OccurredWhen
         ((snd.dst == snd.at) &&
          (p.src_hc != 0))          -> BadDestRep (bdr)
                                       WithAttributes {bdr = snd};

event (Pkt(p) & Sendroute(snd))
       OccurredWhen
         ((snd.dst != snd.at) &&
          ((p.src_seq != best_seq[snd.at][snd.dst]) ||
           (p.src_hc != best_hc[snd.at][snd.dst]))) -> BadNodeRep (bnr)
                                                       WithAttributes
                                                       {bnr = snd};

event (Pkt(p) & Sendreq(sreq))
       OccurredWhen
         ((Sendreq.dst == Sendreq.at) ||
         ((p.dest_seq != 0) &&
          (p.dest_seq < best_seq[Sendreq.at][Sendreq.dst])))
                                 -> BadRReq (breq)
                                    WithAttributes
```

```
                                        {breq  = sreq};

event (Pkt(p) & Senderr(serr))
        OccurredWhen
         (best_hc[serr.at][serr.dst] < INFINITY) &&
         (p.src_seq < best_seq[serr.at][serr.dst] + 1)
                                    -> BadRerr (brerr)
                                        WithAttributes
                                        {brerr = serr};

event (Pkt(p) & serr(serr))
        OccurredWhen
         (best_hc[serr.at][serr.dst] == INFINITY) &&
         (p.src_seq != best_seq[serr.at][serr.dst])
                                    -> BadRerrFwd (berrf)
                                        WithAttributes
                                        {berrf = serr};

/* 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(p) & Recvbetter(rcvb) -> {
  best_seq[rcvb.at][rcvb.dst] = p.src_seq;
  if (p.src_hc == INFINITY) then {
      best_hc[rcvb.at][rcvb.dst] = INFINITY
  } else {
      best_hc[rcvb.at][rcvb.dst] = p.src_hc + 1
  };
  best_next[rcvb.at][rcvb.dst] = p.prev
};

transition seq:
Pkt(p) & Incseq(iseq) -> {
```

```
  best_seq[iseq.at][iseq.at]  = p.src_seq
};

transition err:
Pkt(p) & Senderr(serr) -> {
  best_seq[serr.at][serr.dst]  = p.src_seq;
  best_hc[serr.at][serr.dst] = INFINITY;
  best_next[serr.at][serr.dst] = 0
};

/* Requirements Checking: Loop Invariant */

event (Pkt(p) & Recvbetter)
       OccurredWhen
         ((best_seq[rcvb.at][rcvb.dst] >
           best_seq[best_next[rcvb.at][rcvb.dst]][rcvb.dst])
       || ((best_seq[rcvb.at][rcvb.dst] ==
            best_seq[best_next[rcvb.at][rcvb.dst]][rcvb.dst])
        && (best_hc[rcvb.at][rcvb.dst] <=
            best_hc[best_next[rcvb.at][rcvb.dst]][rcvb.dst])))

                            -> LoopInvFails (linv)
                               WithAttributes {
                                 linv.node = rcvb.at;
                                 linv.next = best_next[rcvb.at][rcvb.dst];
                                 linv.dest = rcvb.dst;
                                 linv.seq_at_node = best_seq[rcvb.at][rcvb.dst];
                                 linv.hc_at_node = best_hc[rcvb.at][rcvb.dst];
                                 linv.seq_at_next = best_seq[
                                                      best_next[rcvb.at]
                                                               [rcvb.dst]][
                                                      rcvb.dst];
                                 linv.hc_at_next =
                                 best_hc[best_next[rcvb.at][rcvb.dst]][
                                         rcvb.dst]
                               };
EndRecognizer;
```

APPENDIX E

SMTP EVENT RECOGNITION

E.1 IP Fragment Reassembly

To 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 tha 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.

```
transition Frag(p) -> {
  push(ipdatabuf,1);
  ipdatabuf[ipdatabuf#length - 1].first = p.ip_off;
  ipdatabuf[ipdatabuf#length - 1].len = p.ip_plen;
  ipdatabuf[ipdatabuf#length - 1].data = p.payload;

  int i = p.ip_off in {
    while (i*8 < p.ip_plen) {
      ipdatarecd[i] = true;
      ipdataptr[i] = ipdatabuf#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 (ipdatarecd[i] == false) then pdone = false;
              i = i + 1;
            }
          }
          d.isdone = pdone;
          if (pdone == true) then {
            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 = "";
        }
      }
```

E.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.

```
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;
          }
        }
      }
```

E.3 SMTP Event Recognizer

The following is a code listing for the SMTP recognizer, `smtp.nerl`, that was used to carry out the analysis of mail server implementations in Chapter 7.

```
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 DATAEND   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  {
    ty_mbox sender;
    ty_mbox receivers[];
    string data;
  } ty_msg;

  typedef {
    string error;
    int status;
  } ty_smtperror;

  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 ty_smtperror Command_Error;
output event ty_smtperror 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.error = "Junk Command";
                                          b.status = status;
                                          };


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.error = "Unexpected Hello";
                                                  b.status = status;
                                                  };

transition m: MailFrom(m)
              OccurredWhen
                ((status == HELLO) &&
                 (respstatus == RESPDONE)) -> {laststatus = status;
                                               status = MAIL;
                                               respstatus = RESPNONE;
                                               senderseen.user = m.user;
                                               senderseen.domain = m.domain;
                                              };

event MailFrom(h)
      OccurredWhen !((status == HELLO) &&
                     (respstatus == RESPDONE))
                        -> Command_Error(b)
                           WithAttributes
                           {b.error = "Unexpected MailFrom";
```

```
                          b.status = status;
                         };

transition r: RcptTo(r)
              OccurredWhen
                (((status == MAIL) ||
                  (status == RCPT)) &&
                  (respstatus == RESPDONE)) -> {
                    laststatus = status;
                    status = RCPT;
                    respstatus = RESPNONE;
                    receiverseen.user = r.user;
                    receiverseen.domain = r.domain;
                    push(receiversseen,1);
                    receiversseen[receiversseen#length-1].user =
                    r.user;
                    receiversseen[receiversseen#length-1].domain =
                    r.domain;
                  };

event RcptTo(h)
      OccurredWhen !(((status == MAIL) ||
                      (status == RCPT)) &&
                      (respstatus == RESPDONE)) ->
                                    Command_Error(b)
                                     WithAttributes
                                     {b.error = "Unexpected Rcpt";
                                      b.status = status;};

event RcptTo(r)
      OccurredWhen (((status == MAIL) ||
                     (status == RCPT)) &&
                     (respstatus == RESPDONE)) -> 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)) ->
                             {laststatus = status;
                              status = DATA;
                              respstatus = RESPNONE;};

event Data(h)
      OccurredWhen !((status == DATA) &&
                     (respstatus == RESPDONE)) ->
                              Command_Error(b)
                              WithAttributes
                               {b.error = "Unexpected Data";
                                b.status = status;};

transition dl: Dataline(d)
```

```
                OccurredWhen ((status == DATA) &&
                              (respstatus == RESPDONE)) ->
                                 {concat(databuf, d);};

transition de: Dataend(d)
                OccurredWhen ((status == DATA) &&
                              (respstatus == RESPDONE)) -> {
                                 laststatus = status;
                                 status = DATAEND;
                                 respstatus = RESPNONE;
                                 concat(databuf, d);};

event Dataend(d)
      OccurredWhen ((status == DATAREAD) &&
                    (respstatus == RESPDONE)) ->
                                 Mail_Sent(m)
                                 WithAttributes {
                                    int i;
                                    m.sender.user = senderseen.user;
                                    m.sender.domain = senderseen.domain;
                                    push(m.receivers,receiversseen#length);
                                    i = 0;
                                    while (i < receiversseen#length) do {
                                      m.receivers[i].user =
                                         receiversseen[i].user;
                                      m.receivers[i].domain =
                                         receiversseen[i].domain;
                                      };

                                    m.data = databuf;
                                    };

transition rs: Reset(r)
                OccurredWhen (respstatus == RESPDONE) -> {
                  laststatus = status;
                  status = RESET;
                  respstatus = RESPNONE;
                  pop(receivers,receivers#length);
                  pop(receiversseen,receiversseen#length);
                  databuf = "";
                  };

event Reset(h)
      OccurredWhen
       (respstatus != RESPDONE) -> Command_Error(b)
                                    WithAttributes
                                    {b.error = "Unexpected Reset";
                                     b.status = status;};


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.error = "Unexpected Quit";
                                                b.status = status;
                                                };

transition n: Noop(n)
              OccurredWhen (respstatus == RESPDONE) -> {laststatus = status;
                                                        status = NOOP;
                                                        respstatus = RESPNONE;
                                                        };

event Noop(h)
      OccurredWhen (respstatus != RESPDONE) -> Command_Error(b)
                                              WithAttributes
                                               {b.error = "Unexpected Noop";
                                                b.status = status;
                                                };

transition hl: Help(n)
               OccurredWhen (respstatus == RESPDONE) ->
                                   {laststatus = status;
                                    status = HELP;
                                    respstatus = RESPNONE;};

event Help(h)
      OccurredWhen (respstatus != RESPDONE) ->  Command_Error(b)
                                               WithAttributes
                                               {b.error = "Unexpected Help";
                                                b.status = status;
                                                };

transition v: Verify(n)
              OccurredWhen
               (respstatus == RESPDONE) ->     {laststatus = status;
                                                status = VERIFY;
                                                respstatus = RESPNONE;};

event Verify(h)
      OccurredWhen
       (respstatus != RESPDONE) ->  Command_Error(b)
                                     WithAttributes
                                     {b.error = "Unexpected Verify";
                                      b.status = status;};

transition e: Expand(n)
              OccurredWhen (respstatus == RESPDONE) -> {laststatus = status;
                                                        status = EXPAND;
                                                        respstatus = RESPNONE;
```

```
                                                        };

event Expand(h)
      OccurredWhen
       (respstatus != RESPDONE) -> Command_Error(b)
                                     WithAttributes
                                      {b.error = "Unexpected Expand";
                                       b.status = status;};

transition junk: Command_Error(b) -> {laststatus = status;
                                      status = JUNK;
                                      respstatus = RESPNONE;
                                      };


event Response(r)
      OccurredWhen
       ((respstatus == RESPDONE) ||
        ((respstatus == RESPOK) &&
         (r.code != respcode))) -> Response_Error(b)
                                     WithAttributes
                                     {b.error = "Unexpected Response";
                                      b.status = status;};

transition rsp: Response(r)
                OccurredWhen ((respstatus == RESPNONE) ||
                              ((respstatus == RESPOK) &&
                               (r.code == respcode))) -> {
                                 if (r.contd == true) then {
                                   respstatus = RESPOK;
                                   respcode = r.code;
                                   } else {respstatus = RESPDONE};
                                 };

event Response(r)
      OccurredWhen (r.contd == false) -> MetaResponse(m)
                                          WithAttributes {
                                            m.client_ip = r.client_ip;
                                            m.server_ip = r.server_ip;
                                            m.client_port = r.client_port;
                                            m.code = r.code;
                                            m.ok = r.ok;
                                            m.data = r.data;
                                            m.contd = r.contd;
                                          };

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 != 451) &&
  (m.code != 452)))) ||
((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 != 421)))) ->
      Response_Error(b) WithAttributes
                         {b.error = "Incorrect Response Code";
                          b.status = status;
                          };

transition mr: MetaResponse(m)
               OccurredWhen ((status == HELLO) &&
                             (m.ok == false)) -> {status = laststatus;};

transition mr: MetaResponse(m)
               OccurredWhen ((status == HELLO) &&
                             (m.ok == true)) -> {
                               pop(receivers,receivers#length);
                               pop(receiversseen,receiversseen#length);
                               databuf = "";
                               };

transition mr1: MetaResponse(m)
                OccurredWhen ((status == MAIL) &&
                              (m.ok == false)) -> {status = laststatus;};

transition mr2: MetaResponse(m)
                OccurredWhen ((status == MAIL) &&
                              (m.ok == true)) -> {
                                sender.user = senderseen.user;
                                sender.domain = senderseen.domain;
                                };

transition mr3: MetaResponse(m)
                OccurredWhen ((status == RCPT) &&
                              (m.ok == false)) -> {status = laststatus;};

transition mr4: MetaResponse(m)
                OccurredWhen ((status == RCPT) &&
                              (m.ok == true)) -> {
                                push(receivers,1);
                                receivers[receivers#length-1].user =
                                receiverseen.user;
                                receivers[receivers#length-1].domain =
                                receiverseen.domain;
                                };

event MetaResponse(m)
      OccurredWhen ((status == RCPT) &&
                    (m.ok == true)) -> 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)) -> {
                                  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.error = "Junk Resp is OK.";
                                            b.status = status;
                                            };

  event MetaResponse(m)
        OccurredWhen (m.ok == false) ->    Response_Error(b)
                                            WithAttributes
                                            {b.error = "Neg Resp.";
                                             b.status = status;
                                             };

  event MetaResponse(m)
        OccurredWhen (m.code == 421) -> Done(d) WithAttributes {d = true};

EndRecognizer;
```

APPENDIX F

TCP EVENT RECOGNIZER

The following is a code listing for the TCP recognizer, `tcp.nerl`, that was used to carry out the initial analysis in Chapter 8. This recognizer is unchanged for the co-networked analysis, only its signature is modified to include channel annotations.

```
Recognizer TCP =

  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;
    int tcp_sport;
    int tcp_dport;
    bool to;
    bool from;
    ty_seg segs[];
  } ty_tcpdata;


#define CLOSED 0
#define SYNSENT 1
#define SYNRECD 2
#define ESTABLISHED 3
#define FINWAIT1 4
#define FINWAIT2 5
#define CLOSING 7
#define CLOSEWAIT 8
#define LASTACK 9
#define C_SYNRECD 10
#define S_SYNRECD 11
#define S_ESTAB 12
#define E_CLOSEWAIT 13
#define F_CLOSING 14
#define F_CLOSED 15


  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_sport;
  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 ToData;
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(s) WithAttributes {s = 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};


  /* ****************************************** */
  /*        SYN   To    */
  /* ****************************************** */

  event (TCP(p) & Syn(s) & To(t))
        OccurredWhen (sender.status != CLOSED) ->
        TCP_Error(hs) WithAttributes
                    {hs = "1:Syn sent in non-closed state"};

  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;
  };

/* ****************************************** */
/*        SYN   From    */
/* ****************************************** */

  event (TCP(p) & Syn(s) & From(t))
        OccurredWhen (receiver.status != CLOSED) ->
        TCP_Error(hs) WithAttributes
                    {hs = "2:Syn sent in non-closed state"};

  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;
        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;
  };

  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 = F_CLOSING;
        sender.fin = p.tcp_seqno + p.tcp_dlen;
  };

  transition (TCP(p) & Fin(s) & From(t))
        OccurredWhen (sender.status == FINWAIT2) -> {
        sender.status = F_CLOSED;
        sender.fin = p.tcp_seqno + p.tcp_dlen;
  };


/* ****************************************** */
/*        ACK of FIN To     */
/* ****************************************** */


  transition (TCP(p) & Ack(s) & To(t))
        OccurredWhen (sender.status == E_CLOSEWAIT) -> {
        sender.status = CLOSEWAIT;
  };

  transition (TCP(p) & Ack(s) & To(t))
        OccurredWhen (sender.status == F_CLOSING) -> {
        sender.status = CLOSING;
  };

  transition (TCP(p) & Ack(s) & To(t))
        OccurredWhen (sender.status == F_CLOSED) -> {
        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 {}
  };

 event (TCP(p) & Data(d) & To(t))
       OccurredWhen (receiver.unacked > 1) ->
         TCP_Error(e)
          WithAttributes
          {e = "17:Too many unacked packets";};

 event (TCP(p) & Data(d) & To(t))
       OccurredWhen ((p.tcp_seqno < lastack) ||
                     (p.tcp_seqno >= receiver.lastack + receiver.wsize) ||
                     ((receiver.fin != 0) &&
                      (p.tcp_seqno > receiver.fin))) ->
         TCP_Error(e)
          WithAttributes
          {e = "18:Data sent outside window";};


/* ****************************************** */
/*        DATA From     */
/* ****************************************** */


  transition TCP(p) & Data(d) & From(t) -> {
    bool tmpcopy;
      int i;
      tmpcopy = false;
```

```
      i = 0;

      sender.unacked = sender.unacked + 1;

      while (i < sender.segments#length) do {
          if ((sender.segments[i].beg == p.tcp_seqno) &&
              (sender.segments[i].dlen == p.tcp_dlen))
          then {tmpcopy = true}
          else {tmpcopy = false}
        };
      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    */
/* ***************************************** */


  event TCP(p) & Ack(a) & To(t) ->
        FromData(d) WithAttributes {
          int i;
          i = 0;
          d.wbase = p.tcp_ackno;
            while ((i < sender.segments#length) &&
                   (sender.segments[i].beg < p.tcp_ackno)) do {
              push(d.segments,1);
              d.segments[d.segments#length-1].data = sender.segments[i].data;
```

```
            };
        };

  transition TCP(p) & Ack(a) & To(t) -> {
          sender.lastack = p.tcp_ackno;
          sender.wsize = p.tcp_win;
          sender.unacked = 0;
        };

/* ****************************************** */
/*        Ack of DATA From     */
/* ****************************************** */

  event TCP(p) & Ack(a) & From(t) ->
        ToData(d) WithAttributes {
          int i;
          i = 0;
          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;
        };


/* ****************************************** */
/*        TCP_Data Output     */
/* ****************************************** */

  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 {
        receiver.segments[i].beg = receiver.segments[i+d.segments#length].beg;
        receiver.segments[i].dlen = receiver.segments[i+d.segments#length].dlen;
        receiver.segments[i].data = receiver.segments[i+d.segments#length].data;
      };
        pop(receiver.segments,receiver.segments#length - i);
  };

  transition FromData(d) OccurredWhen (d.segments#length > 0) -> {
    int i; i = 0;
    sender.wbase = d.wbase;
      while (i < sender.segments#length-d.segments#length) do {
        sender.segments[i].beg = sender.segments[i+d.segments#length].beg;
        sender.segments[i].dlen = sender.segments[i+d.segments#length].dlen;
        sender.segments[i].data = sender.segments[i+d.segments#length].data;
      };
        pop(sender.segments,sender.segments#length - i);

  };

/* ****************************************** */
/*         Done     */
/* ****************************************** */

  event TCP(p)
        OccurredWhen
         ((receiver.status' == CLOSED) &&
          (sender.status' == CLOSED)) -> Done(d) WithAttributes {d = true};

EndRecognizer;
```