Testing a Network by Inferring Representative State Machines from Network Traces

Nancy Griffeth, Yuri Cantor, and Constantinos Djouvas
The City University of New York
345 Fifth Avenue
New York, NY

***Abstract*— This paper describes an innovative approach to network testing based on automatically generating and analyzing state machine models of network behavior. The models are generated by the network test tool AGATE (Automatic Generator of Automata for TEsting), which is also described in this paper. The proposed test approach mimics experimental method, requiring repeated cycles of observing the network, modeling the network, making predictions about network behavior, and evaluating predictions.**

This paper focusses on the modeling step, in which the test tool AGATE automatically generates representative state machines from observed network traces. The generated state machines closely approximate the behavior of components of the network under test. Faults in the system may be immediately apparent from the state machines, but more importantly the state machines can be used for formal analysis. We propose this as a cost-effective alternative to manually defining a state machine before beginning tests.

I. Introduction

Network testing can be quite difficult due to non-deterministic behaviors, unpredictable interactions of large numbers of components, and the variety of equipment and software involved in a network. A failure that occurs at a vulnerable point in the execution of a protocol can have a catastrophic effect, while the same failure at any other point is benign. It is therefore crucial to support network testing with formal models that allow analysis of the network and of the effectiveness of the testing. However, experience has shown that formal models are rarely available before a networked system is tested [6], [11].

Because of this, we propose an experimental approach to network testing, illustrated in Figure 1 and described in more detail in [10]. The target modeling language is I/O automata [25].

The approach can be summarized as follows:

1) *Prepare.* Determine the requirements for the network. In our experiments, we state the requirements as trace properties of I/O automata, thereby avoiding the need for detailed modeling.
2) *Iterate.* Repeat until the tester develops a high level of confidence that the model is representative of the actual network behavior.

This work supported in part by NSF Award Number 0435130 and the Cisco University Research Program.

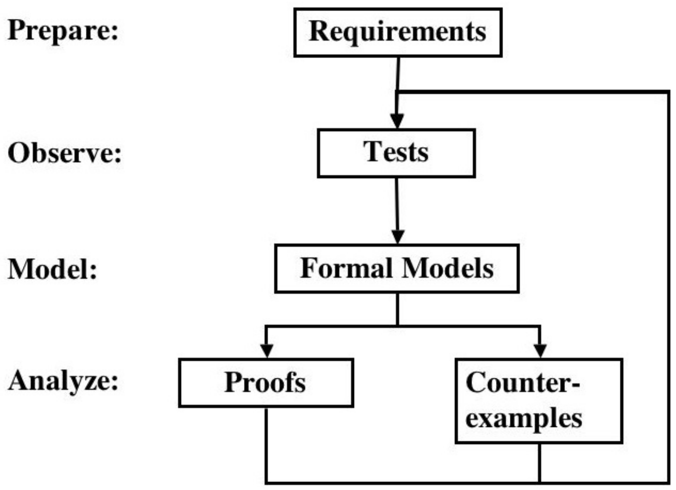

Fig. 1. The test process.

a) *Observe.* Design and run tests. The output of the tests is a collection of network traces, as created by tcpdump or ethereal, in libpcap format.
b) *Model.* Generate timed I/O automata models of the processes running in the network.
c) *Analyze.* Prove the required properties for the automata or find counter-examples. In our experiments, we use PVS[27] for theorem-proving and GMC2[12], [14] for model-checking.

II. Background

We use timed I/O automata (TIOA) to model network behavior [20], [25]. TIOA provide an integrated model of time, a natural composition operator for automata, and tools for converting TIOA to other forms. For example, tools have been developed to convert TIOA to the automated theorem-proving system PVS[27] and are being developed to convert them to the randomized, Monte-Carlo model checker GMC2[13].

This work is motivated by experience with protocol conformance testing and interoperability testing. Most importantly, the seminal work of David Lee and others[1], [15], [23], [24] illustrates the power of formal models in testing. Many authors have subsequently developed models for test case generation, verification, or model-checking of network protocols[3], [5], [7], [17], [29], [19].

The biggest problem that testers encounter with applying formal methods to testing is that formal models are seldom available until years after the software has been implemented in the network. Also, much software is proprietary, so that testers do not have the option of generating a model from the software, but must develop their own model.

Generating state machines from a network trace is closely related to synthesis of state machines from message sequence charts (MSCs). MSCs for important scenarios of a network protocol are commonly seen in requirements documents, but state machines are much less common. Thus synthesizing state machines from MSCs is a useful step toward using a formal model to generate test cases and perform other analyses of the requirements.

A number of researchers have described algorithms for synthesizing state machines from MSCs. Kruger et al[21] define semantics for MSCs and describe an algorithm for defining Statecharts[16] from MSCs. Alur et al[2] give answers to some questions raised by this use of MSCs: Is a collection of MSCs realizable by state machines for the components? Can a state machine be synthesized from a collection of MSCs? Are additional, possibly undesirable MSCs implied by the collection, and can they be identified?

Generating state machines from network traces is motivated differently and raises different issues. We generate the state machines in order to analyze what the system actually does, rather than what it should do, and then ask if what it does is correct. Important issues include how to obtain network traces that cover enough behaviors to generate a representative state machine; how to use such a state machine to find errors and anomalies in the system under test; and how to create a state machine that can be interpreted by a tester.

Our approach reverses an important approach to protocol conformance and interoperability testing, which requires using a state machine to generate an optimal set of test cases [1], [8]. For that kind of testing, the state machine must be available before the implementation can been tested.

Developing a model of the system to be tested can be extremely time-consuming for present-day protocols. For example, TCP was first standardized in RFC 793[28], in 1981. In most cases TCP models have been only partial[32], [9], [31]. A complete model of TCP[6] was developed 24 years after the TCP standard, and reportedly took a full 9 staff-years to develop.

Another, more efficient approach is to model-check the code itself. This was done quite successfully in developing a switching software for a voice over IP gateway [18]. Another interesting use of this method was testing TCP [26]. However, this is possible only if code is available; often, when testing a network, the code is proprietary.

Our approach is also related to *machine identification*[23], [4], which describes algorithms for identifying a finite state machine by observing its input/output behaviors. Some of the assumptions cannot be satisfied in the testing environment, however. In particular, there is no way to know how many states the state machine has.

Instead of trying to determine exactly what state machine is executing, we seek to use network traces and tester knowledge of the intended workings of the protocol to approximate the state machine more and more closely on successive iterations.

III. The AGATE Algorithm

The goal of the algorithm is to construct I/O automata corresponding to the components involved in a networked system, such that every observed network trace could be generated by executing the constructed automata. The automata may also generate additional traces, which can reasonably be inferred from observed network traces.

In fact, we may find it convenient to generate and then compose different automata for a single network component, each managing a different activity in the component. For example, a DHCP server incorporates two important activities: One is communication with its clients and the other is determination of which IP address to offer a client. We generate one automaton to manage communication between client and server and a second automaton to manage IP address allocation. These automata are simple to generate and their composition emulates the behavior of a single network component doing both.

A. Roles of fields in messages

Generating an automaton from a set of traces requires making some assumptions about the way the components use the fields in the messages. We assume first that there are related sequences of messages, which we call *sessions*, and that each session is uniquely identified by the values of one or more fields (the *session identifiers*). We assume that the way a component processes a session is the same, regardless of the exact values of the session identifier fields.

An example of a session is a TCP session. A single TCP session transmits a single stream of data. The *session identifier* fields are the IP address and port for the source and the destination.

Another example is a DHCP communication session. The *session identifier* field is the transaction ID. A DHCP session includes all the communication between a DHCP client and a DHCP server, involving the client's requests to lease and repeatedly renew a lease on some IP address.

A second type of DHCP session manages allocation of the IP addresses, and involves all the messages between clients and servers involving a single IP address. In an IP session, the IP address may be allocated to one client after another. (Of course, correct behavior requires that it be allocated to at most one at a time.)

The fields identifying separate components of the networked system are *automaton identifiers*. For a TCP session, automata are identified by IP address and port. For a DHCP session, server automata are identified by the server IP address and client automata are identified by the client hardware address.

A message contains other identifying fields in addition to session and automaton identifiers. These fields identify objects of interest in the session, for example, an offered IP address in a DHCP session. The values of these fields are arbitrary, and different values will be treated similarly by the networked components. We call such fields *symbolic*, and we assume that any permutation of the values of the symbolic fields in a legal session results in another legal session for the automaton. The concept of symbolic field is closely related to the concept of *data independence* in model checking[22]. Roughly speaking, a protocol is data independent with respect to a data type if the only operation performed on the data type in the operation of the protocol is to test it for equality. It would appear that fields are symbolic for a protocol exactly when the protocol is data independent with respect to the type of the field.

For example, a requested IP address is an arbitrary identifier in DHCP. Clients and servers treat the IP address in the same way, regardless of its value. Similarly, the offered IP address is an arbitrary identifier. The server ID is an arbitrary identifier to the client; the client hardware address is an arbitrary identifier to the server.

The remaining fields are either ignored (*invisible* to AGATE) or *regular*, meaning that they are significant in the state changes in the generated state machine.

A tester must be able to identify the types of the fields in order to use AGATE. This is not a particularly high bar, since the tester must know the network and the protocol well enough to identify errors in the traces. This certainly requires that the tester knows and understands the use of the fields in the messages.

The state machine generation algorithm uses an xml file, like the one shown in Figure 2, to define the fields in a packet. The file is based on a Packet Details Markup Language (PDML) and NetPDL, which have been developed for a Windows-based network analyzer [30]. PDML is also one output format provided by the network analyzer Ethereal. Field attributes are specified by keywords such as name, size, and so on. Optional fields typically follow the fixed fields, and are usually encoded as $\langle type, length, value\rangle$ triples. The example shows how these are defined using a combination of a loop and a case structure.

Finally, the automaton field(s), session identifier field(s), regular fields, symbolic fields, and invisible fields are identified are the representation of the fields in the messages has been defined.

```
<proto name="DHCP">
<fields>
<fixed name="op" />
<fixed name="htype" />
<fixed name="hlen" />
<fixed name="hops" />
<fixed name="xid" size="word" />
<fixed name="secs" size="short" />
<fixed name="flags" size="short" />
<fixed name="ciaddr" size="word" />
<fixed name="yiaddr" size="word" />
<fixed name="siaddr" size="word" />
<fixed name="giaddr" size="word" />
<fixed name="chaddr" vector="16" />
<fixed name="sname" vector="64" />
<fixed name="file" vector="128" />
<fixed name="magic" size="word" />
<loop>
<looptype type="while" oper="1"/>
<fixed name="opcode"/>
<switch>
<case value="50" >
<includeblk name="IPRequest"/>
</case>
<case value="51" >
<includeblk name="leaseTime"/>
</case>
<case value="53" >
<includeblk name="msgType"/>
</case>
...
</switch>
</loop>
</fields>

<block name="IPRequest">
<fields>
<fixed name="len"/>
<fixed name="address" size="word"/>
</fields>
</block>
<block name="leaseTime">
<fields>
<fixed name="len"/>
<fixed name="time" size="word"/>
</fields>
</block>

<block name="msgType">
<fields>
<fixed name="len"/>
<fixed name="type"/>
</fields>
</block>
...
</proto>

automaton servID
session xid
symbolic yiaddr, chaddr, IPRequest
regular leaseTime, msgType
invisible op, htype, hlen, secs, flags, ciaddr,
          siaddr, giaddr, sname, file, hops
```

Fig. 2. Message definition file for DHCP Server automaton. This definition was used to generate the automaton in Figure 3. For readability of the examples, we select just a few key fields to work with: `servID` is the server id, `xid` is the transaction id, `yiaddr` is the IP address offered by a server, `chaddr` is the client hardware address, and `IPrequest` is the IP address requested by a client. The only regular fields are `leaseTime`, which is the number of minutes proposed for the lease, and `msgType`, which is the message type.

An efficient representation of messages permitting efficient determination of the equivalence of sessions is defined in Section III.C.1.

B. The Algorithm

In this section, we describe the automated construction of a session state graph. The session state graph is a directed graph representing a finite state machine. Equivalent sequences of messages follow the same paths through the graph. Equivalence of sessions is defined as follows.

Definition 1. Let $m_1, ..., m_k$ and $w_1, ..., w_k$ be two sequences of messages. Denote the value of field f of a message by $m(f)$. Then we say that $m_1, ..., m_k$ is *equivalent* to $w_1, ..., w_k$ if

1) For each regular field r and each i, $m_i(r) = w_i(r)$.
2) For each symbolic field p and each i, there is a permutation π_p of the values of p such that $\pi_p(m_i(p)) = w_i(p)$.

C. The Session State Graph

We outline the AGATE algorithm for constructing a session state graph here and expand on the details in subsections III.C.1, 2, and 3.

1) Partition the network trace at a component into *sessions* according to the values of the session fields and map the symbolic fields to the normal representation of their values (defined in the Section III.C.1).
2) Define a tree corresponding to the messages in the trace, using the above equivalence relation (section III.C.2).
3) Roll up equivalent branches of the tree from the leaves (section III.C.3). The motivating idea is that if the set of futures of two states are identical (i.e., the set of possible

traces starting in those states), then the states themselves are indistinguishable from outside the automaton and therefore we can combine the two states.

1) Representing values of symbolic fields.: In this section, we define a representation for the symbolic fields of messages that allows efficient determination of the equivalence of messages and hence of sessions, which are sequences of messages.

For each session, the algorithm converts the values of the symbolic fields to their *normal representation* to make them easier to process in subsequent steps. The normal representation is used to determine efficiently whether two sessions are equivalent. The normal representation seems to be a rather obvious way to represent such values, but since we have not yet seen it discussed in the literature we present it here.

Definition 2. Given any sequence of values $x_1, ..., x_k$, define the *normal representation* $n_1, ..., n_k$ of the sequence as follows:
(1) $n_i = n_j$ if there is a $j < i$ such that $x_i = x_j$
(2) $n_i = size(\{x_j | j < i \wedge x_j \neq x_i\})$ otherwise

In other words, the first time a value appears in a sequence, its value in the normal representation is equal to the number of distinct values that appear before it in the sequence. For example, the normal representation of the sequence $a, a, d, a, b, c, a, b, d$ is $0, 0, 1, 0, 2, 3, 0, 2, 1$. Note that the first value in the normal representation of any sequence is 0. We can use of the normal representation in place of the actual values, because if two sequences of values over a domain are related by a permutation of the values in the domain, then their normal representation is the same, and vice versa.

Define the *normal representation* for a sequence of messages $m_1, ..., m_k$ as a sequence of messages $m'_1, ..., m'_k$ with the property that each sequence of symbolic field values $m'_1(p_j), ..., m'_k(p_j)$ is equal to the normal representation for the sequence $m_1(p_j), ..., m_k(p_j)$. We can determine the equivalence of sequences of messages efficiently using the normal representations.

Two sequences of messages $m_1, ..., m_n$ and $w_1, ..., w_n$ are equivalent if and only if the normal representations of the messages are equal in the regular and symbolic fields.

2) Building the session state graph.: A session state graph is constructed so that there is a path $p_1, ..., p_k$ from the root to a leaf for any sequence of messages $m_1, ..., m_k$. Each edge p_i is labeled by the normal representation of the corresponding message m_i. The tree has the property that two sessions follow the same path for k steps if and only if the sequences of messages consisting of the first k messages in each session are equivalent.

1) The first step is to initialize the tree, with a single start state (the root of the tree).
2) Add the first session to the tree. Suppose the first session has messages $m_1, ..., m_n$. Then after adding the first session, the tree has a single path starting at the root. The edge below the root has label m_1; the next edge on the path has label m_2; and so on, to the edge above the leaf, which has label m_n and primary message m_n.
3) Repeat for each session: To process each subsequent session $w_1, ..., w_n$ in the trace, find the path with the largest k such that $w_1, ..., w_k$ is the same as $m_1, ..., m_k$.
   a) $\mathbf{w}_1$. Choosing the first edge requires finding the edge below the root such that the edge label m_1 has the same values as w_1 in the regular and symbolic fields.
      If there is no message m_1 on an edge below the root with corresponding regular and symbolic fields equal, then create a new edge below the root with label containing w_1 and having a new state at its child end. The remaining messages $w_2, ..., w_n$ will be the edge labels for a new path below w_1.
   b) $\mathbf{w}_{h+1}$. Assume that after processing the first h messages, the sequence of messages $w_1, ..., w_h$ follows a path leading to state s_h in the tree.
      If there is an edge whose label is m_{h+1} has the same values for regular and symbolic fields as w_{h+1}, then the sequence of messages $w_1, ..., w_{h+1}$ leads to state s_{h+1} at the child end of the edge.
      If there is no such edge, add a new edge below s_h and use w_{h+1} as its label. The node at the child end of the edge is a new state.

Theorem 1. Two sequences of messages $m_1, ..., m_k$ and $w_1, ..., w_k$ lead to the same level k node of the tree (i.e., to the same state) if and only if they are k-equivalent.

Proof: By induction on k. ■

3) Combining equivalent subtrees.: The motivation of this part of the algorithm is the idea that two states are the same if no subsequent step of the automaton distinguishes between them.

All the leaves are the same state, i.e., the terminating state. Two states that are parents of leaves are the same state if they have the same number of edges below them and if the primary messages on the edges are equivalent to each other, i.e., they have the same normal representations. Such states are combined by the automaton generation algorithm, by throwing away one of the states and connecting the edge above the discarded state to the remaining state.

Subsequently, two states at k levels above the leaves are the same state if they have the same children after rolling up all levels up to level $k - 1$ above the leaves and if the messages on the edge to each child are equivalent. As before, after the algorithm has determined that two states are the same, it removes one state and connects its parent edge to the remaining state.

A tester needs to be careful that the test run doesn't end until all sessions have reached some form of termination. Otherwise, there will be incomplete sessions in the network trace, so that messages that should only occur in the middle of a session will appear to be at the end. This will cause AGATE to combine some states that are not actually equivalent

IV. Examples.

We show three examples of session state graphs. The first (Figure 3) was taken from a small LAN using a LinkSys router,

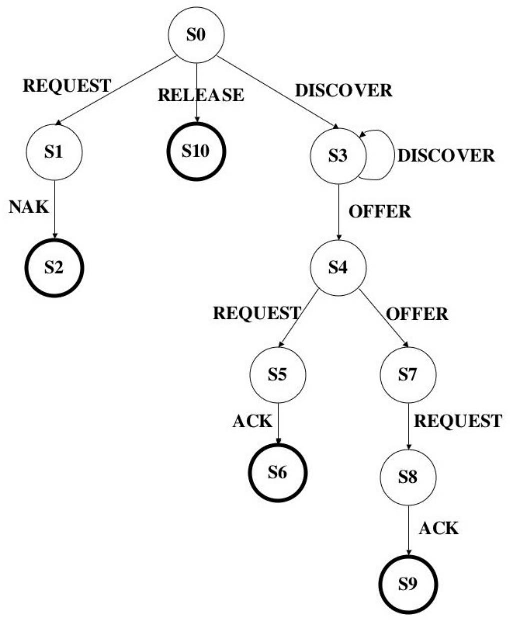

Fig. 3. This DHCP session state graph was created from a trace taken on a small LAN. It includes 14 DHCP transactions (sessions).

and illustrates the communication session state graph obtained from normal operation of a DHCP server.

This figure contains four kinds of sessions:

1) *States S1-S2.* Sessions requesting a previously allocated IP address, immediately after connection to the network. In all cases, the requests were denied.
2) *State S10.* Sessions releasing an IP address.
3) *States S3-S6.* Successful allocation of an IP address. The loop on state S3 indicates that multiple identical DISCOVERS were sent from the client.
4) *States S3-S4 and S7-S9.* Successful allocation of an IP address. This differs from the previous successful allocation in that two IP addresses were offered and these sessions accepted the second. If they had accepted the first, state S9 could have been combined with S6 and S8 with S5.

Figure 4 is included to illustrate an IP allocation session state graph that might be observed on a correctly operating DHCP server. This represents some sessions that a tester would expect to see after generating a state machine from actual network traces, assuming that the DHCP server is correct. The left branch shows an IP address that was reallocated to one client after initial connection to the network, followed by allocation to a second client after the first left. The right branch shows initial allocation of an IP address to one client and then to another.

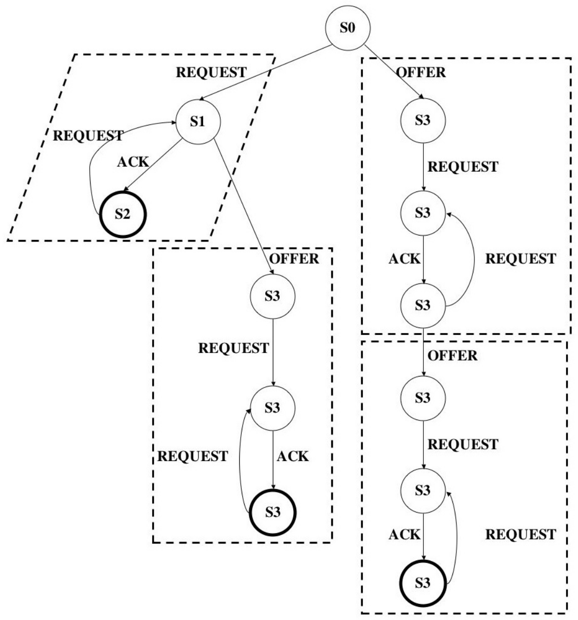

Fig. 4. This DHCP session state graph illustrates some sessions that might be seen with a reasonably correct server.

Figure 5 illustrates a session state graph for IP address allocation. This was obtained using a network trace from an ISC DHCP server with faults injected. Thus the state machine deviates from the expected state machine in several ways. The session state graph records the attempted allocation of two IP addresses to four clients on a single LAN. Dotted boxes isolate separate transactions. Some of the transactions are clearly incorrect - for example, the REQUEST leading to state S1 is an INIT-REBOOT, which received a NAK, but the OFFER should be a response to a DISCOVER in a new transaction, not a response to a request in the same transaction.

V. Conclusions and Future Work

The next step is to apply formal tools to the generated I/O automata. One such tool is PVS. The theory group at MIT has built a TIOA to PVS converter, so that we can convert automata directly to a form that PVS can process. The two most important properties of DHCP are:

1) Never allocate overlapping leases on an IP address to two clients; and
2) Allocate an IP address when possible.

We will experiment with proofs of these properties. Also, as the examples indicate, some properties involving correct sequencing of messages should probably be proven as well. We will also run these experiments on the DHCP failover protocol.

Another important step is to use timing information in the construction of the automata. We are collecting the set of elapsed times between messages when we build the session state graph, and plan to use this to infer timeouts and timed triggers. As with the roles of fields, we think that some

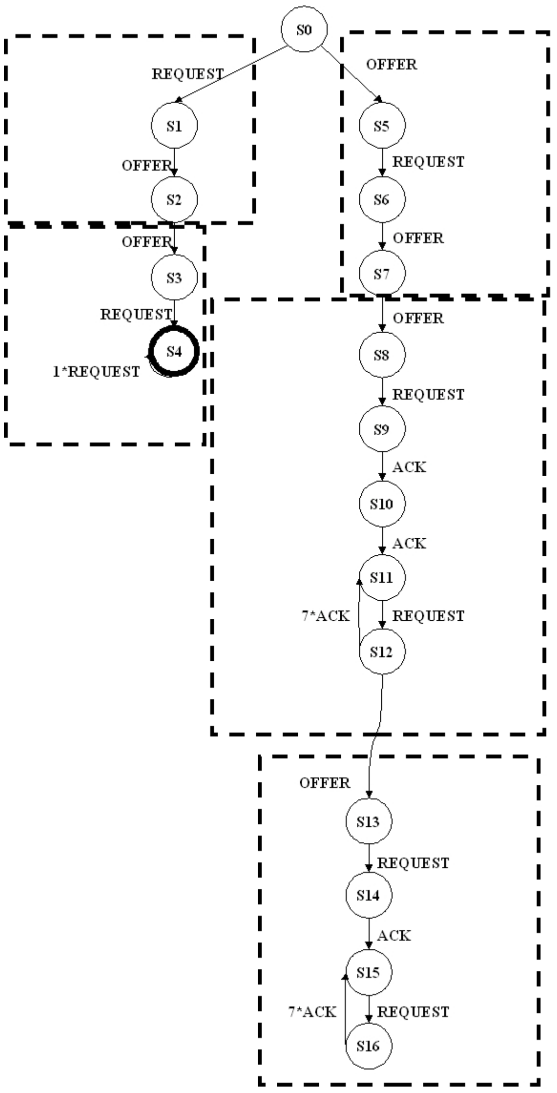

Fig. 5. This DHCP session state graph organizes sessions by the IP addresses. DHCP transactions are contained in the dotted boxes.

information from the programmer - such as the use of the lease time in DHCP - will be necessary to make this successful.

Clearly, the quality of the test results will ultimately depend on running the test cases that expose the faults in the code. We plan to examine ways of using the generated session state graphs to explore parts of the state space that may have been neglected.

Although it was not the intent of this work, this approach may support reverse engineering a system. It also appears that the session state graph may support better network monitoring by providing a useful summary of a network trace.

References

[1] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *IEEE Trans. Commun.*, 39:1604–1615, November 1991.

[2] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *International Conference on Software Engineering*, pages 304–313, 2000.

[3] Rajeev Alur and Bow-Yaw Wang. Verifying network protocol implementations by symbolic refinement checking. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 169–181, London, UK, 2001. Springer-Verlag.

[4] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[5] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.

[6] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *SIGCOMM Proceedings*, 2005.

[7] D. Cypher, D. Lee, M. Martin-Villalba, C. Prins, and D. Su. Formal specification, verification, and automatic test generation of atm routing protocol: Pnni. In *Proceedings of FORTE/PSTV'98, Case Studies in Protocols*, 1998.

[8] D. Cypher, D. Lee, M. Martin-Villalba, C. Prins, and D. Su. Formal specification, verification, and interoperability testing of atm routing protocol: Pnni. In *Proc. FORTE/PSTV*, 1998.

[9] Scott Dawson, Farnam Jahanian, and Todd Mitton. Experiments on six commercial TCP implementations using a software fault injection tool, 1997.

[10] Constantinos Djouvas and Nancy Griffeth. Experimental method for testing networks. In *Proceedings of SERP'05 - The 2005 International Conference on Software Engineering Research and Practice*, June 2005.

[11] Nancy Griffeth and Frederick Stevenson. An approach to best-in-class interoperability testing. *Journal of the International Test and Evaluation Association*, 23(3):68–82, October 2002.

[12] R. Grosu, X. Huang, S. Jain, and S. A. Smolka. Open-source model checking. In *Proc. of SoftMC'05, the 3rd Workshop on Software Model Checking*, July 2005.

[13] R. Grosu and S.A. Smolka. Monte Carlo model checking. In *Lecture Notes in Computer Science*. Springer-Verlag, April 2005.

[14] Radu Grosu and Scott A. Smolka. Monte carlo model checking. In *TACAS*, New York, October 2005.

[15] Ruibing Hao, David Lee, Rakesh K. Sinha, and Nancy Griffeth. Integrated system interoperability testing with applications to voip. *IEEE/ACM Transactions on Networking*, 12(5):823–836, 2004.

[16] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[17] Ahmed Helmy, Deborah Estrin, and Sandeep K. S. Gupta. Fault-oriented test generation for multicast routing protocol design. In *FORTE XI / PSTV XVIII '98*, pages 93–109. Kluwer, B.V., 1998.

[18] G. J. Holzmann and Margaret H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, - 2000.

[19] Y. Lakhnech K. Stahl, K. Baukus and M. Steffen. Divide, abstract and modelcheck. In *Proc. 5th and 6th International SPIN Workshops*, London, UK, 1999. Springer-Verlag.

[20] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed I/O Automata: A mathematical framework for modeling and analyzing real-time systems. In *24th IEEE International Real-Time Systems Symposium (RTSS'03)*, 2003.

[21] Ingolf Krger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In *Distributed and Parallel Embedded Systems*. Kluwer, 1999.

[22] Ranko Lazić and David Nowak. A Unifying Approach to Data-independence. Technical Report PRG-TR-4-00, Oxford University Computing Laboratory, June 2000.

[23] D. Lee and M. Yannakakis. Testing finite state machines: state identification and verification. *IEEE Trans. on Computers*, 43(3):306–320, March 1994.

[24] 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, August 1996.

[25] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., March 1996.

[26] M. Musuvathi and D. Engler. Model checking large network protocol implementations, 2004.

[27] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[28] Jon Postel. RFC 793: Transmission control protocol, September 1981.

[29] B. Premore. An experimental analysis of bgp convergence time. In *ICNP '01: Proceedings of the Ninth International Conference on Network*

Protocols (ICNP'01), page 53, Washington, DC, USA, 2001. IEEE Computer Society.

[30] F. Risso. Analyzer 3ȯ. http://analyzer.polito.it/.

[31] David Sinclair and James Power. Specifying and verifying communications protocols using mixed intuitionistic linear logic. *Electronic Notes in Theoretical Computer Science*, 133:255–273, May 2005.

[32] Mark A. Smith and K. K. Ramakrishnan. Formal specification and verification of safety and performance of tcp selective acknowledgment. *IEEE/ACM TRANSACTIONS ON NETWORKING*, 10(2), 2002.