

Experiences in automating the testing of SS7 Signalling Transfer Points

Tim Moors[†], Malathi Veeraraghavan, Zhifeng Tao, Xuan Zheng, Ramesh Badri[‡]
[†] Uni. of New South Wales
Sydney, NSW 2052, Australia
+61 2 9385 4000
moors@ieee.org

Polytechnic University
5 Metrotech Center
Brooklyn NY 11201 USA
+1 718 260 3050
mv@poly.edu
jefftao@photon.poly.edu
zhxfifa@photon.poly.edu

[‡] Sprint PCS
600 Cummings Park, Suite 2850
Woburn, MA 01801 USA
+1 781 638 1547
rbadri01@sprintspectrum.com

ABSTRACT

Signalling System 7 (SS7) is a widely used signalling protocol in telephone networks. Service providers frequently need to test their Signalling Transfer Points (STPs), which switch SS7 messages, for both protocol conformance and interoperability. This paper describes the Automated SS7 Test Result Analyzer (ASTRA), which automatically analyzes the data collected during STP tests. ASTRA consists of files that describe how the STPs are expected to behave during the test, and Perl code that translates this Expected Behavior into a program that can search the data collected during the test for the expected events, and report on whether the system passed the test. ASTRA readily processed over 30,000 events for each test run, and identified abnormal behavior that could interfere with interoperability and protocol conformance.

Categories and Subject Descriptors

B.4.5 [Input/Output and Data Communications]: Reliability, Testing, and Fault-Tolerance

C.2.2 [Computer-Communication Networks]: Network Protocols – *protocol verification*

General Terms

Design, Reliability, Experimentation, Verification.

Keywords

Automation, Signalling System 7, Signaling System 7, SS7, STP

1. INTRODUCTION

Signalling System 7 (SS7) [6] is used around the world to provide the signalling required for telephone communications. A core component of a SS7 network is the Signalling Transfer Point (STP). Service Switching Points (SSPs) generate SS7 messages, and act as their ultimate destinations, while STPs switch these messages on their path from origin to destination. For reasons of reliability, these STPs are deployed in mated pairs, so that they can maintain the switching service even when one STP fails. One mated pair of STPs may connect to a set of SSPs, to Service Control Points (SCPs) that provide database functions, or to another mated pair of STPs, forming a “quad” configuration, as shown in Figure 1.

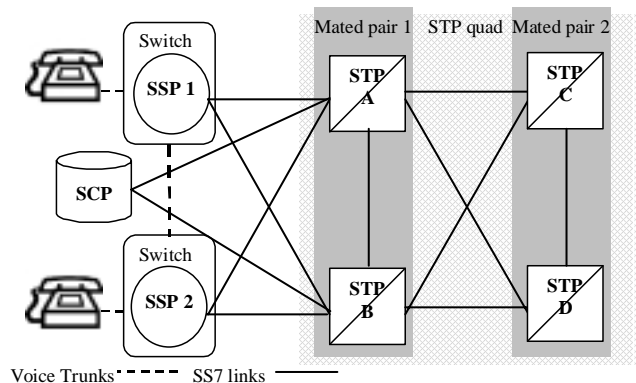


Figure 1: Main elements of an SS7 network.

Pairs of STPs that are interconnected to form a quad may belong to an SS7 network operated by a single organization, or may form the interface between SS7 networks from different organizations. Interfaces between SS7 networks allow a Local Exchange Carrier to send its signalling traffic over another carrier’s network. This is becoming increasingly important in the United States as government regulatory bodies are encouraging competition between Local Exchange Carriers by requiring incumbent carriers to open their network to the signalling traffic of competitors, in exchange for being granted rights to provide long-distance services. However, different carriers may source their STPs from different vendors or configure their STPs differently, and this leads to a need to test the interoperability of STPs from different carriers. A single carrier may even source STPs from different vendors, for reasons such as risk management or encouraging competition between vendors for the supply of equipment. Thus, carriers have a strong need for interoperability [1] and conformance [5] testing of STPs from different vendors.

For a carrier to ensure that the signalling network continues to operate correctly, it needs to regularly test STP quad interoperability: A carrier will often deal with STPs from three or more different vendors, and each vendor regularly issues new releases of the software and hardware for their STPs, e.g. multiple times per year, and the carrier may interface with other carriers who reconfigure their STPs. Not only must the testing be performed regularly, but it is complicated (due to the nature of the SS7 protocol), time consuming, and (for the most part)

mundane. For example, by one estimate, the analysis of each test takes 933 hours of labor, and over five years the test analyses would cost approximately \$1.5M. These characteristics make STP quad interoperability testing a good candidate for automation.

Verizon Communications, who operate one of the largest SS7 networks in the world in support of their US operations, recently decided to automate their STP quad interoperability testing. This automation project divided the testing process into four stages: setting up the test environment, executing the tests, retrieving data collected during the test, and analyzing the test data. Verizon sponsored the Center for Advanced Technology in Telecommunications at Polytechnic University to create and implement a system for the analysis stage of the testing process. This system is called the Automated SS7 Test Result Analyzer (ASTRA). This paper focuses on ASTRA, and the analysis stage of the testing process. While ASTRA was designed for use in conjunction with other stages of testing that are automated, it has also been successfully used to analyze the results of tests that were conducted manually.

Protocol testing [7] requires a Test Plan (e.g. [4]) that describes a series of tests, each specifying the stimuli that will be applied to the System Under Test during the test, with the intention of exercising the functionality of the System Under Test. From this Test Plan, and knowledge of the requirements of STPs [2] and the operation of the SS7 protocol [3], we developed a description of the Expected Behavior (EB) of the STPs for each test in the Test Plan. The EBs are expressed in an abstract machine-readable form, which will be described in Section 2. The key stages of the analysis process are: First, ASTRA takes as input the data collected during the test, a description of the network configuration used during the test, and a record of when each test step was performed, and formats these into an Actual Event Record (AER) for the test. ASTRA then compares this concrete AER with the abstract EB, and classifies events described in the AER and EB as being either matched (in EB and AER), missing (in EB but not in AER), hidden (in EB but not in AER and not expected due to monitoring limitations), or unexpected (in AER but not in EB). ASTRA then statistically analyzes the events observed in the AER in order to infer STP parameters, and presents these and information about the test events in a report that describes whether (and how) the STPs passed (or failed) the test. Figure 2 illustrates this architecture.

The structure of this paper follows that of ASTRA: Section 2 describes the format used to represent the EB of the SUT, and Section 3 describes the Translator that translates the EB into an Analyzer, a Perl program that can perform the analysis. Section 4 describes how data that was collected during a specific test run is formatted, and Section 5 describes the process of matching events in the EB with those in the AER. Section 6 discusses ASTRA's statistical analysis and report generation, Section 7 describes some bugs that ASTRA detected when analyzing the data produced during a test run, and Section 8 concludes the paper.

2. THE EXPECTED BEHAVIOR (EB)

This section describes the Expected Behavior Expression Language (EBEL) that we created to describe how we expected STPs to behave during a test.

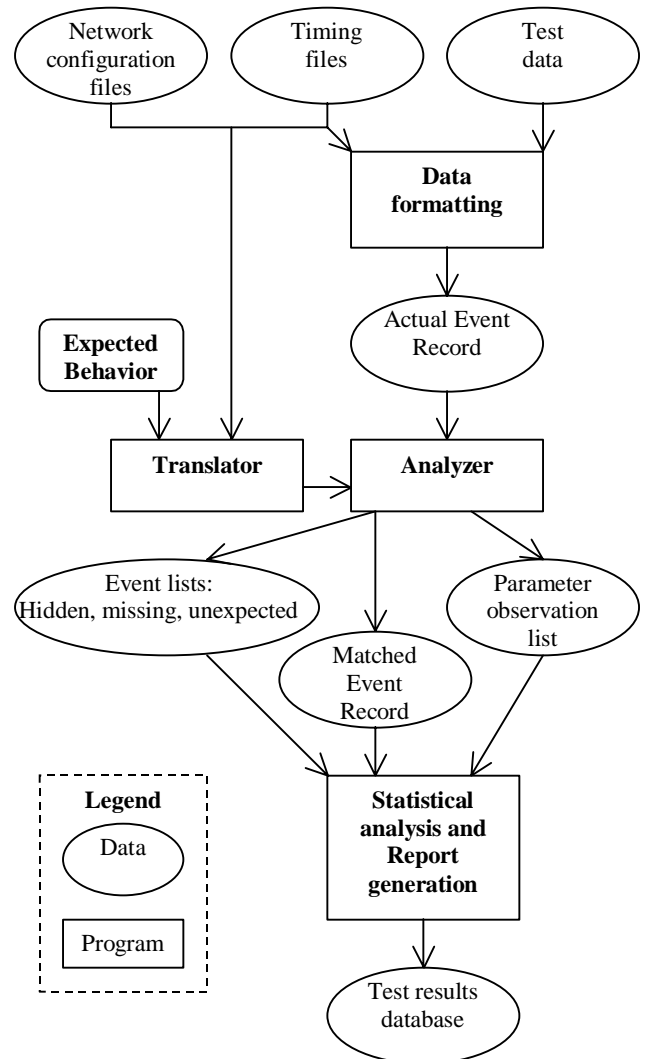


Figure 2: Architecture of the analysis system.

A key feature of the Expected Behavior (EB) is that one event can cause multiple succeeding events, and the relative order of these succeeding events is not always predictable. For example, referring to Figure 1, the failure of the link connecting SSP 1 to STP A should cause STP A to send Transfer Restricted (TFR) messages to neighboring SSP 2 and the SCP, indicating that it no longer provides a direct path to the SSP whose link failed. The protocol does not specify the order in which STP A must send these two TFR messages. For our tests, each expected event is triggered by a single predecessor; there is no situation in which an event depends on multiple independent predecessors. Thus, the EB can be expressed as a tree, as shown in Figure 3.

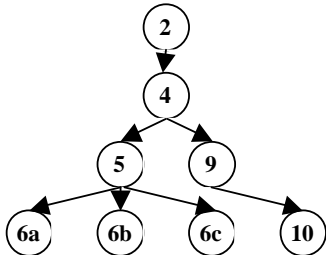


Figure 3: Abstract expected events form a tree of causation.

The heart of the EB for a test is a numbered list of events that are expected during the test, with each event indicating the numbers of any other events that it causes. Figure 3 provides an abstract example of the tree structure of EBs, showing which numbered events cause which other events¹. To describe the EB, we created text files that described one event on each line, with lines describing events that caused other events containing the phrase “causes” followed by a list of the line number(s) of the event(s) that the event on this line causes. To manage the numbering and references, we created the EB using Microsoft Word, using its line-numbering, cross-references and update features, and saved these as text files for automated processing.

There are two types of events in the EB: stimuli to the SUT, and the responses of the SUT.

The chain of events that stimulate the SUT during the tests define the Test Plan, and the Generic Commands and control structures used to represent these events can be used in both the EB, and also in a Test Action List that drives automated test execution. The Generic Commands and corresponding stimuli are:

`fail_link()`, `restore_link()`: fail or restore a specific link in the network configuration,
`set_load()`: set the load on a particular link to a particular level, and
`pause()`: wait a certain period (e.g. to allow network traffic to stabilize) before injecting the next stimulus.

In our testing, we were able to express the Test Action List in an open-loop manner so that the test actions, and their timing, did not depend on the behavior of the SUT. In the EB, each line containing a test action also contains a symbolic label (e.g. `$ta_fail{$s}`), indicating that the value of the variable `$s` should be appended to the string `$ta_fail`, to create a label for the test action on this line) that other parts of the EB (e.g. `while` loops) can use to refer to the time of the test action, and are used by the matching process to define the scope of the search for an event. The EB uses symbolic names for these times, since it is produced before the tests are performed. After the test, during analysis, these symbols are instantiated with the actual times when each test action was performed.

¹ The numbers of this abstract example correspond to the line numbers of a particular EB shown in Figure 4, which is why events 6a 6b 6c have letter designations, indicating different iterations of a loop.

The messages that an STP transmits on the links are the most common measure of the response of an STP, and the EB may be concerned with the type and parameters of a specific message, or with the aggregate volume of messages flowing over a particular path. The EB includes `findmsg()` commands to find a message with particular properties (e.g. type and originating and destination point codes) passing over a certain link or an arbitrary link, and `assert_load()` commands to check whether the load on a link is within a specified tolerance of a certain level. The STP’s log of events that it observes is another important response, since accurate logs help alert network operators of imminent problems, and help problem diagnosis.

The parameters to `findmsg()` can be specified as wildcards (*), indicating that the matching process should ignore that parameter when trying to find a matching message. Wildcards are convenient in the EB to determine if an STP sends a particular message to *any* other node (wildcards for link name and destination), or whether an STP sends *any* type of message to a particular node.

The EB also includes control structures that allow iteration of the stimuli and responses. For example, a `foreach` loop can be used in an EB, in conjunction with a `fail_link()` command, to fail each link connecting an STP to adjoining SSPs. A `while` loop can be used in an EB, in conjunction with a `findmsg()` command, to indicate that an STP is expected to continue sending a certain type of message until a certain time.

A `waitfor()` command allows the EB to describe situations in which the SUT is expected to wait for a certain period before making some response. The `waitfor()` command has two arguments: The first provides the symbolic name of the parameter that specifies the period that the STP should wait for, and the second indicates which STP’s parameter to use. Part of the matching stage (Section 5) involves recording the observed values of these parameters, so that the report generation stage (Section 6) can infer the value of this parameter that the STP is using and compare it to the value that the STP claims to be using.

The `waitfor()` command is used in two different ways: Within `while` loops, it regulates the speed at which the `while` loop repeats, and directly causes only one other event. For example, a `waitfor()` command may be used in a `while` loop to describe the expectation that an STP will send a probe or status update message regularly while a link is unavailable. Outside of `while` loops, it is used to indicate a delay between one event and one or more consequent event(s). In this case, the `waitfor()` command may cause multiple events.

Figure 4 provides a sample extract of an EB, showing how the STPs are expected to respond to the sequential failure of links connecting SSPs to STP A. In this example, `@SSPS_E` is a set of SSP identifiers, and the `foreach` loop is repeated to fail the link connecting STP_A to each SSP (`$s`) in this set. The set `@set_1` describes the nodes (SSPs and SCPs) that remain connected to STP_A after the link has been failed; the extract of the code does not show where `@set_1` is updated. In this example, a period T11 seconds after STP A detects the link failure, it is expected to send TFR messages to STP D and the

```

1. foreach $s (@SSPS_E) {
2.     $sta_fail{$s}: fail_link(A(STP_A)($s)) causes 4,...
3. ...
4.     waitfor(T11, STP_A) causes 5, 9, ...
5.     foreach $t (@set_1) {
6.         findmsg(A(STP_A)($t), $t, STP_A, $t, TFR, $s)
7.     }
8. ...
9.     findmsg(D(STP_A)(STP_D), STP_D, STP_A, STP_D, TFR, $s) causes 10
10.    while($time<$ta_restore{$s}) {
11.        waitfor(T10, STP_D) causes 12
12.        findmsg(D(STP_A)(STP_D), STP_A, STP_D, STP_A, RSR, $s)
13.    }
14. ...
15. foreach $s (@SSPS_E) {
16.     $ta_restore{$s}: restore_link(A(STP_A)($s)) causes ...

```

Figure 4: Extract of the Expected Behavior of an STP (with ellipsis added and lines renumbered).

remaining neighboring SSPs. Once STP D receives the TFR message, it should repeatedly (with period T10 seconds) send a Signalling Route-Set-Test-Restricted (RSR) message to STP A enquiring if the path has been restored. \$time is a global variable that is updated during the matching process of analysis to indicate the time of the latest actual event that the matching process has attempted to match with an expected event.

Creating the EB for each test in the Test Plan, and validating our expectations against observed behavior, proved to be a time-consuming task. For the twelve tests in our Test Plan, the EB totalled almost 6,000 lines and described over 30,000 events that occur in a test run.

3. THE TRANSLATOR

Apart from the “causes” construct and line numbering, the language described thus far for the EB bears a striking resemblance to Perl [8], with the addition of calls to functions such as `findmsg()` and `waitfor()`. Originally, we intended to use Perl for pattern matching between the EB and AER, both of which are text files. We intended to write a Perl script that took these two files as input, and interpreted the EB to determine what events were expected next and then search the AER for those events. But then we realized that we could simply translate the EB file itself into a Perl script: If one event causes a group of other events, then this can be expressed in Perl as the first event being the condition of an `if` statement, and the consequent events being placed in the body of the `if` statement. When an event causes a group of events surrounded by a control structure (e.g. event 4 causes a `foreach` loop), then that control structure is included in the body of the `if` statement that depends on that event. Figure 5 shows the nesting of `if` statements to represent the abstract event tree of Figure 3. Each number of Figure 5 can be replaced with the line from Figure 4 that has the same number, creating the hierarchy of `if` statements corresponding to Figure 3.

The depth of the tree of expected events can lead to many levels of hierarchy in the `if` statements, producing code that is difficult for humans to read. That is acceptable because the translated EB is only used by the automated system, and is precisely the reason

```

if( 2 ) {
    if( 4 ) {
        /* 5 */ foreach $t (@set_1) {
            6$t
        }
    }
    if( 9 ) {
        10
    }
}

```

Figure 5: Representing the abstract event tree of Figure 3 as a series of nested if statements.

why the human analysts did not express the EB as a hierarchy of `if` statements in the first place.

The hierarchy of `if` statements leads to a program that performs a depth-first search of the tree of expected events. The search for an event covers all actual events that occur between the times of the antecedent event (stored in a variable `$time`) and the time of the next test action (stored in another variable `$next_action_time`). Events that are enclosed in a `while` loop form an exception to this general rule; their search continues until the time specified as a parameter to the `while` loop, not until the time of the next test action.

ASTRA includes a Translator program, written in Perl, that translates the EB into an Analyzer. This translation process is similar to the compilation of a program written in a high-level language such as C: It takes as input a program that is in human-readable form, and produces a program that can be executed. In the case of ASTRA, the Analyzer that results from translation is a Perl program. The translation process also links the Analyzer to libraries of functions that are used during the matching process, and for formatting the results of the matching process.

The Translator is run after a test is run so that the Analyzer can include information that is specific to the test run. This information includes the network configuration (which affects the definitions of sets of nodes used in the EB), the list of links that

were monitored during the test (which determines which events will be classified as being “hidden”), and the times of test actions (which affect the searching operation).

A secondary role of the Translator is to help a human analyst create the EB files. The Translator can be used to check the syntax of an EB file, helping the human analyst (programmer) debug the EB. The Translator also converts identifiers in the EB into the form required by Perl: enclosing names of links and nodes in quotes (e.g. STP_B → “STP_B”) and prefacing function calls with ampersands and suffixing them with semicolons (e.g. findmsg(...) → &findmsg(...);).

To summarize, the Translator takes an EB file and information about the network configuration used for a test, and the times of test actions for that test, and produces an Analyzer – a Perl script that can be executed to analyze the data produced during the test.

4. DATA FORMATTING

The data formatting is fairly mundane, but nonetheless necessary. The formatting takes several files as input, and produces a single Actual Event Record (AER) that is used for event matching. This section describes the five stages of data formatting: filtering, integration, consistency, abstraction, and preparation for processing.

Filtering: The complexity of SS7 equipment means that it is often time consuming to configure a system from scratch, even when this is done under automated control. Consequently, the STPs that we tested were often configured with information (point codes) about nodes that were not involved in the test, and produced messages about these nodes. The filtering stage of the data formatting removed this irrelevant information from the AER.

Consistency: Multiple monitoring devices collect data during a test. Unfortunately, monitoring devices from different manufacturers present the observed data in different formats. For example, monitors from different vendors used different ways of representing the time of monitored events. The formatting stage of analysis is responsible for ensuring that the data from monitors from different vendors is represented in a consistent manner. Even monitors from the same vendor may use the same format for the time, but may not be properly synchronized before the test. In our experience, it was simpler to record the time on each monitor at an epoch point before the test, and to synchronize the events during data formatting, than to synchronize the monitors themselves prior to the test.

Abstraction: In the EB, nodes and links are identified in a human-friendly symbolic form (e.g. “STP_A”), whereas the monitors identify nodes and links in a numerical form that depends on the configuration of nodes used in the test (e.g. STP A may be identified by its point code 246-022-001). The data formatting stage converts information recorded by the monitors from its numerical form to the corresponding symbolic form, so as to facilitate matching of the expected and actual events.

Integration: During a test, data is collected from multiple monitoring devices; both monitors of the same type monitoring different links, and monitors of different types (e.g. STPs, link monitors, and SS7 emulators). The formatting stage of

processing is responsible for integrating data from each of these sources into a single file called the AER.

Pre-processing: The final stage of data formatting is designed to facilitate the processing of the data during event matching. First, ASTRA sorts the events in the AER into chronological order, so as to expedite searching during event matching. ASTRA also adds to each event a single-character “matched” field that it sets to 0. This is done because the filesystems that we used with ASTRA (Unix and Microsoft Windows) allowed extension of files, and re-writing of existing characters of files, but not insertion of characters in the midst of files. By adding this field when ASTRA sequentially traverses (and copies) the AER during data formatting means that ASTRA does not have to insert this field at random locations when it matches events during the event matching stage.

It is worth pointing out two practical realities of our experience with monitoring equipment: First, it was not always possible to comprehensively monitor all points in the test network due to the cost and availability of monitoring equipment. Consequently, one input to ASTRA is a list of monitoring points for which data was not collected. When an event occurs in the EB, but the corresponding point is not monitored, then ASTRA classifies the event as being “hidden”. The second point is that deviations between the EB and observed behavior may be due to failure of the monitoring equipment, not of the SUT. Amongst the vast volume of data that we analyzed, we noticed a couple of instances in which the monitoring equipment seemed to provide incorrect reports (e.g. missing a digit in a timestamp) or was just as suspicious as the STPs under test (e.g. the interval between periodic messages would occasionally be twice the normal value, suggesting that an interstitial message was dropped by either the STP or monitoring equipment). The next section will discuss ASTRA’s search depth, i.e. how far it traces down the tree of expected events once an event is deemed to be missing, and how ASTRA’s search depth and the potential for monitors to miss events can affect ASTRA’s output.

5. EVENT MATCHING

The goal of event matching is to classify events in the EB and AER as being either hidden, missing, matched or unexpected. Figure 6 shows these event classifications, and how they relate to which events were expected, which actually occurred, and which were observable given the monitors used during the test. The classification of an event is recorded by writing it to one of these files: either the hidden, missing, matched or unexpected event list.

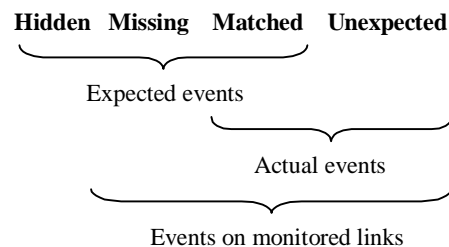


Figure 6: Relationship between the four event classifications and sets of expected and actual events, and monitor coverage.

The first stage of event matching is driven by the EB: For each event in the EB, the analyzer checks if the event was expected on a link that was monitored during the test. If not, then the event is classified as hidden. If the link was monitored, then the analyzer searches the AER for the event. If the analyzer cannot find the event in the AER, then it is classified as missing, otherwise the event is classified as being matched. If the matched event causes other events, then ASTRA also searches for those events from the time of the matched event. Before elaborating on the details of the first stage, we will briefly describe the second stage:

The second stage of event matching is driven by the AER, or more precisely the Matched Event Record (MER), which indicates which events in the AER were matched to the EB, and which were not. This second stage takes each event in the MER that was *not* matched to the EB, and classifies it as unexpected, writing it to the Unexpected Event List.

There are three important details about the operation of the first stage of event matching. These relate to the search depth, the recording of parameter observations, and the matching of events that use wildcards, and are discussed in the following paragraphs.

Search depth: Whenever ASTRA fails to match an event, it prunes the tree of expected events at that event; i.e. it does not search for consequent events that the missing event was expected to cause. If the consequent events were also missing, then they will *not* appear in the Missing Event List, since the omission of their antecedent implies that they will be missing. The motivation for this behavior is to reduce the list of missing events so that a human analyst can concentrate on the root events that were missing, and need not be inundated with a mass of consequent events that may be missing. If the consequent events were not missing, e.g. if monitoring equipment simply failed to detect the antecedent event, then they will appear in the AER, and ASTRA will mark these as being unexpected. Thus, a single missing event can cause multiple consequent events to be marked as unexpected.

Parameter observations: When ASTRA matches an event that is caused by a `waitfor()` statement, it records an observation of the delay between the event that caused the `waitfor()` statement, and the event that the `waitfor()` statement caused. This observation measures an instance of the parameter of the `waitfor()` statement that indicates how long the STP should wait before causing the next event. The report generation stage (Section 6) provides a statistical analysis of these parameter observations, allowing a human analyst to infer the parameter values that the STP is using. The details of parameter observations depend on whether or not the `waitfor()` statement was included within a `while` loop.

Observations that indicate missing events: The rate at which a `while` loop repeats is generally limited by a `waitfor()` statement within the `while` loop. For example, an STP may be expected to send a message of a certain type, and then wait for a time specified by an STP parameter, before repeating the process. ASTRA only marks an event that is caused by a `waitfor()` statement in a `while` loop as being missing if it was *never* found. Other missing events are detected by the

distribution of the parameter observations. For example, if an STP was expected to send a message every T_{10} seconds, but every second message that it tried to send was lost before collection by the monitor, then ASTRA would indicate that the message was found, but that the timer parameter was observed to be $2 \times T_{10}$ seconds.

Observations of multiple events with a single cause: `waitfor()` statements that are outside `while` loops can cause multiple consequent events. For example, lines 2 and 4 to 7 of Figure 4 state that T_{11} seconds after STP A detects a link failure, it should send TFR messages to each SSP that connects to it. In theory, these consequent events should occur at the same time, but in practice they may occur at different times. This can happen because of serialization on the links on which the messages are observed, or because of serialization within the processor that generates these events. The question arises as to how to use the different delays from the antecedent event to these different consequent events to infer the parameter value that the STP is using? Often, the intention of `waitfor()` statements is to postpone consequent events; i.e. the consequent events should occur with a delay *no less* than that specified by the parameter, rather than the consequent events should all occur before a specified delay. Thus, each time ASTRA encounters a `waitfor()` statement that causes multiple events, it distinguishes between the parameter measurements for the first event that was caused, and those for other events that were caused.

Wildcards: The analyzer distinguishes between messages that were matched with wildcards and those that matched messages that were explicitly specified in the EB. This distinction is marked in the MER by an asterisk (*) for events matched with wildcards, and a 1 for events matched to a specific event in the EB. This is done to enable more meaningful reporting of the test results: It is useful to be able to indicate what fraction of messages were missing or unexpected relative to those that were matched, since high ratios may indicate significant deviations from the EB. However, wildcards are generally used to match messages that the analyst doesn't care about², merely to ensure that they don't end up in the unexpected event list. Including events matched with wildcards in the denominator of the ratio leads to ratios that are misleadingly low. Thus, ASTRA distinguishes between events matched with and without wildcards, and only uses the events matched with wildcards when indicating the proportion of missing or unexpected events.

6. REPORT GENERATION

Classifying events as matched, missing, hidden or unexpected is fine, but the people conducting a test are really interested in whether the system under test passed or failed the test, and if it failed, then how did it fail? The report generation stage of ASTRA takes the classified events from the Analyzer and produces a report that provides a top-down summary of the test.

² The analyst may also choose to use wildcards to cover behavior that is too complicated to describe fully using the current form of the language EBEL. We will return to this topic when discussing future work in Section 8.

ASTRA presents the reports in HTML format, since HTML can be presented on varied viewing devices, and this also allowed the reports to be readily combined with a web server that provides a user interface for invoking ASTRA. The results are color-coded, with green for positive results (e.g. expected events that were matched), red for negative results (e.g. missing events), and gray for inconclusive results (e.g. hidden events).

The report starts with a synopsis of the result of the test: pass, fail or inconclusive. The SUT passes the test when there are no missing, unexpected, or hidden events. The SUT fails the test when there are missing or unexpected events. A test is considered to be inconclusive if there are no missing or unexpected events, but there were hidden events. If the SUT failed the test, then ASTRA provides a numerical estimate of the “severity” of the failure. The severity is calculated as the ratio of the number of unexpected or missing events to the number of events matched without wildcards. The severity measure gives missing events twice the weighting of unexpected events, to reflect the presumed precedence of missing events: a STP should still be functionally useful if it does everything it is supposed to (no missing events), even if it does some additional things (some unexpected events).

After the synopsis of the test result, ASTRA continues with an overview of the messages and parameter observations made during the test. The message overview lists the different message types, and how often messages of that type were missing, hidden, unexpected or matched with, or matched without, wildcards. This overview helps a human analyst to identify what protocol functionality failed during a test.

The report then presents an overview of the observations of timer parameter values. For each STP, the report lists the parameters of that STP that were used in the EB, and lists the number of observations of that parameter, and the minimum, average and maximum values of the observations. There are separate tables that provide these statistics about the *first* event observed that suggested that the timer had expired, and another table with statistics about *all* events observed that were caused by the timer expiring.

After this overview, ASTRA presents a detailed listing of each event related to the test. The listing follows the order of events as described in the EB, and either indicates the time of the event that matched an event in the EB, or indicates that the event was missing. Unexpected events are listed immediately after the last test action that preceded them, and may have caused them. Unexpected messages are listed with the time when they occurred so that the analyst can consult the AER and other monitor data to find more information about these messages.

7. TEST RESULTS

When we ran ASTRA on the data produced during a series of interoperability tests between STPs from two manufacturers, we detected three bugs with the STPs from one manufacturer that could interfere with interoperability. These bugs appeared in a link failure test. According to the SS7 protocol, an STP should respond to link failure or restoral by sending a message to neighboring STPs indicating that traffic flow through it is either *restricted* (because it has poor connectivity to the destination), *prohibited* (because it has no connectivity to the destination), or

allowed (because a link has become available, enabling connectivity to the destination). In our tests, the STPs from one vendor sometimes responded to link failure by indicating that traffic was *allowed* to flow, rather than it was indicating that it was restricted or prohibited. These STPs also sometimes sent restriction messages, when they were expected to send prohibition messages, allowing a routing loop to occur. Finally, these STPs were sometimes unnecessarily conservative in that they indicated to their mated pair that traffic was prohibited when it should only have been restricted. ASTRA readily allowed the human analyst to pinpoint this unexpected behavior amidst the 30,000 other events that occurred during the testing.

8. CONCLUSION

The Automated SS7 Test Result Analyzer (ASTRA) is able to automate the analysis of data produced when testing communication protocols. We applied it to analyzing the data collected during interoperability testing of STPs that are used in an SS7 network, and found several bugs in the behavior of STPs from one manufacturer that could interfere with interoperability with other STPs. ASTRA consists of a total of 3,800 lines of Perl code, and the description of the EB for the system under test in our testing amounted to another 6000 lines of code. While ASTRA was designed to solve the specific problem of analyzing the results of SS7 tests, its underlying mechanism (e.g. language for expressing the Expected Behavior, process of translating an EB into a Perl script, and the event matching process) could also be applied more generally to analysis of tests of other protocols.

Future work on automated test result analysis could extend ASTRA to application in tests that have closed-loop control: When conducting testing the response of STPs to congestion, we found that different STPs became congested at different workloads, e.g. because of the performance of their implementation. To formulate these tests in an open-loop manner required manually testing each type of STP to determine the workload at which it would congest, and then incorporating the measured workload in the list of test actions and EB. A closed-loop approach would progressively increase the workload until congestion was detected, i.e. to a level that is not known before the test, and only then check for the expected response to congestion.

Another item for future work might be to extend the Expected Behavior Expression Language so that the EB can depend on events observed during the test. For example, it is important to test the response of STPs to link failures, but for many network configurations, multiple links may fail at approximately the same time and in an order that is unknown before the test. Since the response of the STPs will depend on the order in which the links failed, it should be possible for an Analyzer to determine the order of link failure, and use that to control the Expected Behavior.

9. ACKNOWLEDGMENTS

We would like to thank Verizon Communications for sponsoring this project, and for their staff who collaborated on this project. In particular: Nabeel Cocker, Rimma Iontel, Mei-Jong Lin, John Murphy, Gaston Ormazabal, Wayne Peer, James Sylvester, and Joseph Vecchioli. This project was also sponsored by the New

York State office of Science, Technology and Academic Research (NYSTAR) through the Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University. We would also like to thank Wichean Preedalumpabut. All authors (including Moors and Badri) were at Polytechnic University when this work was done.

10. REFERENCES

- [1] J. Alilovic-Curgus and S. T. Vyong: "A framework for interoperability testing of network protocols", *Proc. International Conference on Network Protocols*, pp. 376-83, 1993
- [2] Bellcore: "Signaling Transfer Point (STP) Generic Requirements", Generic Requirement GR-82-CORE; Issue 2, Dec. 1996
- [3] Bellcore: "Specification of Signalling System Number 7", Generic Requirement GR-246-CORE; Issue 3, Dec. 1998
- [4] ITU-T: "Signalling System No. 7 Protocol Tests", ITU, Recommendation Q.755, Mar. 1993
- [5] S. Kang: "Relating interoperability testing with conformance testing", *Proc. Globecom*, pp. 3768-73, 1998
- [6] A. Modarressi and R. Skoog: "An overview of Signaling System No. 7", *Proc. IEEE*, 80(4):590-606, Apr. 1992
- [7] D. P. Sidhu and T.-K. Leung: "Formal methods for protocol testing: A detailed study", *IEEE Transactions on Software Engineering*, 15(4):413-26, 1989
- [8] L. Wall and others: *Programming Perl*, 3rd ed., O'Reilly & Associates, 2000