Property-Based Testing of Network Implementations

By

Janine Taylor
B.S. (University of California, Davis) 1995

THESIS

Submitted in partial satisfaction of the requirements for the degree of
MASTER OF SCIENCE
in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Committee in Charge

1997

Contents

Chapter 1

Introduction

This paper examines how the use of specifications can facilitate the testing process of a network protocol implementation. Testing programs for flaws and conformity to requirements is a time-consuming and often complex process. The testing process attempts to verify that the implementation of the software is what the customer wanted and that the implementation does not have flaws. The information describing what the program is supposed to do can be outlined in a set of written requirements. In any program, a deviation from the written requirements could possibly allow a user access to information that is supposed to be protected by an authorization scheme. One example would be allowing a user to alter a password file. An oversight in implementation could also allow actions such as giving superuser access to a regular user who might

intentionally or accidentally remove or change files that he is not supposed to have access to. A2program with network capabilities has the possibilities for bugs of a regular program and the possibility of these unauthorized users gaining access over network rather that having to be physically at the machine the program runs on to take advantage of the flaws. Rather than fixing the mistake after it has been discovered and exploited by an unauthorized user, the ability to detect flaws in the implementation of a network protocol during development would prevent the flaws from possibly allowing damage to occur to the system. Testing the implementation against the requirements specifications can be used to detect flaws in the implementation of a protocol. By formally writing the requirements specifications for a program and using these specifications to test the program, the programmer has a greater chance of discovering flaws in the implementation during development. Specification languages provide the capability to formally describe requirements for a system. There are many types of specification languages, each having different purposes. This paper focuses on using the TASpec specification language which is state- based and event driven. TASpec specifications can be used by a tool, the Testers' Assistant, which processes the specifications and uses them to test the implementation for conformation to the requirements. This paper describes how these tools can be used effectively to test an implementation of the alternating bit protocol and file transfer protocol (FTP). It also gives a brief description of using TASpec to write specifications for the TCP/IP protocol.

1.1 The Testing Problem

Currently many testing procedures base their testing of a piece of software on the knowledge of the source code. Information about the implementation is helpful in developing possible test cases to determine the accuracy of the implementation. The tester can use his knowledge to know what to test in the implementation. However, this also leaves the prospect of missing a possible hole in the implementation. Situations where the software is so complicated it is easy to miss testing for possible openings for flaws because there are so many possibilities in the code for paths to take or where the tester is the one who wrote the source code and knows it so well he misses possible holes in the flow of control in the code are particularly susceptible to missing flaws when testing only through information about the implementation. In addition, testing through the source code means that some knowledge of the structure of the implementation is needed in order to test the software. This does not mean that testing should not be done through knowledge about the source code, but it does mean that it should only be a part of the whole testing process.

1.2 Specifications

In developing software, somewhere in the process a document is usually written which states the requirements for that piece of software. Having an accurate requirements document for the software is important because of its possibilities for later use in the development and testing of the software. Requirements specifications can be used for different purposes. They tell the implementer what the code he is writing is supposed to do. He can also use the requirements specification to develop a design specifications document for the implementation. The tester can use a requirements specification to drive the testing process by using it to create test cases for the whole program or to test specific portions of the code. If the tester can also use the requirements document to determine if the implementation conforms to the requirements, this document has the ability to enhance the testing procedure beyond just suggesting test cases. Writing this document in a defined specification language makes it clear to the programmer what is needed and allows the requirements to be checked for correctness. This means that the requirements document can be used to enhance the testing of the software with greater effectiveness whether it is used to suggest test cases or used to test the implementation.

1.3 Security Flaws

Because of the complexity of implementing large pieces of software, it is important that the software be tested while it is being developed as well as when it is finished. Because implementers are human, mistakes will be made and all the possibilities for doing the unexpected can not be caught. In a program without network capabilities, these flaws can cause security problems on the machine the code runs on. These security problems are magnified in programs with network capabilities. This is because not only the machine that runs the flawed code can be affected, but also any machines that are connected to it can be affected by the security holes. If a program is allowed to get into an unpredicted and unmonitored state, it is possible for undesired actions to be performed. With a networked program it could mean that an unknown and/or unprivileged person could get onto the machine and change protections and security or remove and change files on the machine that are not theirs to alter. This could allow them more access that was expected by using the networked program.

1.4 Testers' Assistant

The Testers' Assistant is a tool which tests the C language implementation of a piece of software against TASpec language specifications written for that software. The TASpec language, used by the tool Testers' Assistant, gives the tester a way to formally write specifications for a network protocol. The specifications written could be a software requirements specification, a design requirements specifications, or even specifications which test for certain general types of flaws in code. TASpec is a state-base, event driven language which lends itself to, but is not limited to, writing specifications for event driven and state model based programs. It allows the specifications to be written independently of the source code until the point where direct links need to be made to reference variables or functions in the code. This means that a knowledge of the structure of the source code is not needed to write the specifications or to develop test cases. The specifications can be used to help develop test cases. Once the specifications are written the Testers' Assistant takes them and compiles them with the source code so the test cases can be used to indicate whether the implementation stayed within the requirements for the software. It expects that the system the software is running on does not have flaws.

1.5 FTP

FTP is a tool to allow transfer of files to and from machines connected through a network. Since this service allows access across the network to a machine, there is the possibility that a flaw in the implementation of this protocol could create a security hole on one or both of the connected machines. The flaw could be on either side of the ftp connection, causing such problems as denial of service or unauthorized access, for example. With ftp this could mean that an unauthorized person at either end of the networked machines could be able to overwrite a password file, giving himself superuser access to the machine. Having a requirements specification for checking the implementation reduces the risk of this occurring. This paper looks at a way of writing specifications for FTP using the specification language TASpec and then using these specifications to test for flaws in the implementation.

1.6 Contribution of Thesis

This document describes specifications for the alternating bit protocol and shows how the Testers' Assistant catches flaws in the implementation by means of the specification. This paper does the same for an ftp client. It also describes how to use the Testers' Assistant from writing the specifications to testing the implementation through the specification. Possible specifications for an implementation of TCP/IP are included in an appendix. Also included is the procedure for using the Testers' Assistant in its current state.

1.7 Organization

Chapter 2 contains a brief overview of general types of security flaws that can exist in network protocol implementations.

In chapter 3, we present a quick background on specification languages in general and indicate briefly why TASpec was chosen for writing specifications.

Chapter 4 gives some information about property-based testing and an overview of how the Testers' Assistant works.

Chapter 5 explains how to use TASpec for writing specifications and describes the language itself. Also presented is a simple example of specifications written in TASpec.

In order to give an example of how TASpec and the Testers' Assistant can be used the specifications for an implementation of the alternating bit protocol is in chapter 6. It also contains an explanation of how the Testers' Assistant uses the specification to test the implementation of the protocol.

Chapter 7 contains information about writing specifications for FTP and using the Testers' Assistant with those specifications. It also describes some possible security holes in FTP.

Chapter 8 gives ideas for future work with the Testers' Assistant and TASpec, states the results of writing specifications for alternating bit and FTP, and concludes the thesis with some comments.

The appendices include a completed alternating bit protocol specification and implementation, specifications for an FTP client, partially completed specifications for TCP/IP server, and specific instructions about how to use the Testers' Assistant.

Chapter 2

Security Flaws

There are a variety of flaws that can cause security problems in network protocol implementations. Flaws that can cause security problems can be loosely categorized in a variety of ways. Some flaws are general flaws which can be found in any implementation, for example, not checking that a function returns the expected. Many security holes can be caused by a program with privileged access being used inappropriately or a flaw in that program being exploited. Other flaws can be specific to the protocol and its implementation. For example, the implementation of a program with a state-based flow which missed checking for a possible unaccounted for state transition can allow the program to be forced to stay in a state indefinitely. This can open the door for a denial of service attack. These are not the only possible categories of flaws.

2.1 General Flaws

General flaws can be related to specific events in an or flaws that occur in many different protocols. For example, not checking the return value of system calls and other language-defined functions in a program can cause a problem in implementations. Joel Dodson (Dodson 1996) described a way of testing for these flaws by means of writing specifications for each function detailing how it should be used. He called this a flaw library. In programs which have a sequence of steps in their execution it is possible to write the implementation such that it will allows a process to skip steps in the sequence. This mistake can mean a program can be halted because it can not get out of its current state. Skipping states could also allow a connection to be allowed that normally would not be allowed because a step in the connection setup process was skipped. Another general flaw could be neglecting or improperly checking user authentication. Also, not checking that a program which has privileged permissions is doing only what it should be doing can cause a security hole like the ability to overwrite a password file for the machine it runs on. Some general flaws are so well-known that they have been given names. They can be caused by different events in implementations, but they can all lead to the implementation creating security holes. One such problem is wedging. This is when a process is blocked from going to the next step for an unexpected reason. For example, wedging could be caused by a process skipping a step. When this happens the process is wedged because it can not go to the next step because it is expecting information it is never going to get. Wedging could also be caused by too many requests being sent to the same process which can not handle that many at the same time. In some cases it may be possible to detect the possibility of wedging occurring. As described in chapter 5, wedging can create security holes in the process it occurs in. Joel Dodson (Dodson 1996), and others, have described a variety of ways to classify general security flaws that can occur. We would like to use the Testers' Assistant to catch these flaws in programs.

2.2 Protocol Specific

Other flaws are more protocol specific and are related to the processes a program performs and how they are done. For example, a protocol that sends and receives messages might require the implementation to maintain a list of messages that are expected but have not been received. If the implementation does not maintain this list properly, a flaw could exist which would allow an unexpected message to be accepted as a valid message. In a protocol which has a defined set of requirements for establishing a connection between two end processes, such as TCP with its three-way handshake, if the implementation does not verify that the connection setup process was completed properly before allowing the connection to be completed, a flaw exists in the implementation of the protocol which might allow a connection to be made by an unauthorized process. These types of holes could be caused by a general flaw. For example, a flaw which would allow an unverified connection to be created could be caused by a general flaw like skipping states or not checking the password.

Biswaroop Guha discovered the implementation was allowing states to be skipped in the a version of the TCP/IP protocol. These flaws, in many cases, could be caught by writing specifications for the protocol. The specifications would be written with respect to the requirements for the protocol rather than for catching flaws related to specific functions. Written specifications could also test for a flaw such as allowing the process to skip states and other flaws like it. These are general flaws which could occur in any program, but catching them requires knowledge of the states in the program to test for it. This way of testing the implementation should catch more than just known flaws which relate to the protocol's implementation, but also flaws which might be unrecognized.

Chapter 3

Specification Languages

Writing a requirements specification of some sort has become a generally accepted and necessary part of developing a piece of software. A requirements specification usually contains information about how the program is supposed to work, but not how it should be implemented. A design specification usually goes into detail about implementation, containing such details as the functions and structures of the variables used in the program. The requirements specifications describes what the program is supposed to do. It describes the user interaction and the flow of information or states in the code. It can be text based or written more formally in some specification language. Since the more formal version lends itself toward clearer specifications, a variety of specification languages have been created which can be used to write the requirements for a system which is being developed. Specification languages have a variety of styles, properties and abilities which makes them all useful in some fashion, but for the purposes of this paper, a useful specification language facilitates testing implementations. This chapter classifies specification languages loosely into general specification languages, code generators, and implementation testers.

3.1 General Specification Languages

These languages are ones which can be used to write detailed requirements for the software, but are not linked to the implementation. The programmer has to manually check that the implementation conforms to the specification. The programmer can use the specifications to manually develop test suites for the software and can visually check that the specification and the implementation match. In all cases of writing specifications, knowing that the specification is correct helps give the programmer some assurance that the implementation is correct once it conforms to the specification. As a further aid to their usefulness any of these languages have tools which can verify the specifications for correctness.

The Z ( ) specification language, while it does not have a tool which can verify the specification, is diverse in its ability to develop specifications for a set of requirements. It allows the specifications to be written in the form of a relational, functional, state, or abstract data type model or a combination of these. The SDL specification language ( ) is based on an extended state machine model and has constructs for abstract data type specifications. Checking for the correctness of specifications in both of these languages requires a separate verification process.

VDM (Jones 1990) is a mathematical logic based specification language which lends itself easily to manually using mathematical proofs to verify specifications written in this language.

PAISLey (Zave 1982), a specification language designed for real-time systems, allows for specification through defining processes for the system which include the set of all possible states and their successor states for that process. While it can not verify specifications, it will simulate the behavior of the system, which can ease the testing process by allowing comparisons between the simulation and the actual implementation.

AFFIRM (Guttag 1993) uses state transition machines to describe the specifications for a system. It verifies the specifications through an interactive theorem prover. It can also simulate the operation of the system through the specification.

The ACSR (Ben-Abdalla 1997) specification language, based on process algebra, uses the concept of building components and combining them to produce a full specification. ACSR allows for verification of implementation design specifications against the requirements specification through equivalence checking and state space checking, but it does not have an automatic means of verifying the requirements specifications.

These languages, although useful for writing specifications of software, in general do not help in testing implementations for our purposes. They can not be compared to the implementation beyond manually looking at them to see if the implementation conforms to the specification. Admittedly, the ones which simulate the behavior of the system could help in testing an implementation, but this would still involve manually checking the implementation against the simulated specification.

3.2 Code Generators

The languages ESTELLE and LOTOS, rather than simulating the specifications, will actually generate code for the implementation. Depending on how detailed the specifications are, the code can get fairly specific. These specification languages are good for developing a quick prototype of a piece of software.

ESTELLE (Guttag 1993)is designed for writing specifications for a distributed or concurrent system. The specification constructs are based on exchanging messages between non-deterministic state machines. The tool which processes ESTELLE takes a description of the software and generates C code from it.

LOTOS (Guttag 1993) allows for modeling of sequential behavior, choice, concurrency and non-determinism. Its constructs combine process algebra and abstract data types to produce the specification of a system. After the specifications are written, code is generated which implements the specifications. Depending on the tool used, the code can be generated in a variety of languages. LOTOS allows for actual code to be embedded into the specifications to make the generated code more likely to be executable. Unfortunately, this will also make the specifications more closely linked to the implementation than is usually desirable. The specifications can rapidly begin to look like the implementation and so become more of a design specification than a requirements specification.

These languages, because they actually generate code can help to confirm that the implementation conforms to the specification because the code generated from them will conform to the specification and this code can be tested. Unfortunately, for the generated code to be as complete as the implementation, writing the requirements specifications becomes almost like writing design specifications or even writing the implementation itself. This also could mean that the same flaw could be missed in testing because the generated code is similar to the implemented code.

3.3 Implementation Testers

The specification languages ANNA, GYPSY, and TASpec (described in more detail in the next chapter) actually test the implementation of a piece of software against the written specifications through specific references to variables and functions in the implementation. But, unlike ESTELLE and LOTOS, the requirements do not need to have a similar structure to the implementation in order to test the implementation against the specification.

The specification language ANNA (Luckham 1991) is an extension to the ADA programming language. A program written in ADA can be tested against specifications written in ANNA. Execution of the software will produce a detailed indicator of where the test of the execution failed if it did so. The specifications can be written for any program that is written in ADA. ANNA has the ability to describe high level requirements for the program and can also describe the low level details of how the program is implemented. ANNA seems, especially at the low level, to require some knowledge of the implementation in order to write specification for a program, possibly bringing the specification and the implementation into a closer relationship than should be needed to write a requirements specification for a system. The tool that tests the implementation against the specifications in ADA assumes the specification to be correct. Also, testing against the specifications must be done at run time. ANNA does not have the ability to test the implementation by means of the specifications statically.

GYPSY (Guttag 1993) has the ability to test the implementation against the specifications both before runtime through formal proofs and at runtime. This makes the process of verifying a little simpler at times because some requirements are easier to check by means of running the program than trying to specify them without the ability to run the program, while others do not need to be checked at run time because they can be checked statically. This ability reduces the amount of additional code that would have to be added to the implementation to check specifications at run time, thus increasing the efficiency of the enhanced execution. This tool requires that both the implementation and the specification for the software are written in the language GYPSY. GYPSY, too, assumes that the specifications are correct.

TASpec (Fink 199-), a state-based, event driven specification language, allows for property-based testing of a piece of software through testing the implementation of the software against written specifications for that piece of software. Once the specifications are written for the software, it uses location specifiers to connect the actual specifications to C source code for a program. Location specifiers reference actual variables and functions used and in the source code when the program is executed. Like ANNA, it assumes the specifications to be correct and can only test the program at run time. The specifications are based on assertions which indicate the current state of the process. As assertions are created and retracted, the result of executing the code is checked to make sure all assertions which needed to be made to get to the next set of assertions were true.

These languages all help test that the implementation of a piece of software conforms to the specifications. While the other two languages are both valid, this paper focuses on TASpec because of its ability to integrate itself with C language programs and its state based constructs. TASpec's ability to interface with the C language implementations increases its desirability because many network protocols are written in C. TASpec's syntax is easy to read and use. Writing specifications in TASpec requires minimal knowledge of the source code. Its state-based structuring of the specifications makes it useful in network programs because many of them are state-based.

Chapter 4

Property-based testing and the Testers' Assistant

The Testers' Assistant is based on the idea of property-based testing. Property-based testing involves testing an implementation for a property, such as conformity to requirements or making sure the password is checked correctly. The Testers' Assistant, using TASpec for the specifications, helps the tester to determine if an implementation has the property described by the specification.

4.1 Property-based testing

Property-based testing has the goal of testing a program for a defined property. For example, a property-based test could be used to determine if a program authenticates users correctly or it performs correctly the actions it was designed for. A human tester could use this technique to test a program for specific flaw(s). It could also be used to test the program for conformity to a requirements document.

Since there are innumerable possible flaws that could occur in an implementation, testing only for specific flaws would most likely miss a flaw. Because it focuses on what the program is supposed to allow, testing the implementation through the requirements specification seems to be more complete, even though this method could still miss flaws. So, a testing procedure should include testing a piece of software through specifications properties and flaw properties. Although writing the specification seems to focus on determining if the program performs its task properly, the specifications can also be used to determine what actions the program should not perform or allow to occur. This chapter focuses on using property-based requirements specifications tools for testing a program.

4.2 The Tester's Assistant

George Fink ¡Fink 199-¿ developed a tool, the Testers' Assistant, which is able to take a software requirements specification written in the specification language TASpec and test the implementation of a program against the specification. Using the TASpec language, the tester has the capability to describe the property desired in the implementation so that the Testers' Assistant can use the specifications to determine if the implementation has the desired property. This specification can be used not only to check that the implementation does what it is supposed to do, but also that it does not allow unauthorized or unexpected activities to occur through unexpected input to the system or other unexpected combinations of events. This means the Testers' Assistant can use the specifications for the whole program to test the implementation for flaws by giving the program test data that should not be allowed since it is not allowed by the specification.

The Testers' Assistant works by running the implementation of the program with given test data and then checking that the test data does not cause the program implementation to violate the written specification. The Testers' Assistant first statically analyzes the source code to produce a data-flow graph. Then it has the ability to slice the program based on a desired property, function name, or even a variable name through the data-flow graph. Slicing consists of looking at the data-flow graph to remove all portions of the code which could not possibly effect the value of the object being sliced on. (This paper does not look at slicing.) Then the Testers' Assistant generates another version of the program which will be the one executed during testing. This version has the same functionality as the original program but also has data gathering modules. These modules are added pieces of code which record when pieces of code which are referenced by the specification are executed. Then this new version of the code is executed with various test data. The records produced by these executions are then evaluated to determine if the properties specified have been violated by the implementation.

4.3 Testing the Implementation

The purpose of comparing the implementation of a system against its specification is to aid in testing that the implementation conforms to the specified requirements for that system. This comparison is not meant to be the only check of the implementation, but to be a useful addition to the testing procedure for the software, able to catch errors that other tests might miss or have a hard time finding. (Currently the Testers' Assistant has plans to develop a tool which creates a test suite from the specifications. This could also be used to help develop and expand test suites.) Since testing implementation against specification requires that the code be executed, a test suite or testing protocol of some kind needs to exist for this test to be useful.

TASpec separates writing specifications from knowing the design of the implementation. The specification for a system can be written with no knowledge of the implementation until time to test the implementation against the specification. Then, it is necessary to have some knowledge of the actual implementation. The Testers' Assistant "links" the specification to the implementation by means of location specifiers which contain specific information about the source code. The location specifiers reference user- specified function calls, variable assignments, variable declarations, and variable references in the C code. The specifier needs to determine how the function calls, assignments, and variable references are related to the state elements used in the specifications. State elements describe a portion of the state of the system. For example a specification might be:

```
clouds() and rain() and have_umbrella() {
        retract have_umbrella();
        assert open_umbrella();
};
```

The specifier would determine which of the state elements clouds(), rain(), and have_umbrella() occur because of an event in the implementation and write a location specifier for that assertion. So, if the state element have_umbrella() needed to be asserted whenever the actual variable bought_umbrella in the implementation was assigned the value 1, the location specifier would be:

```
location assign bought_umbrella returns 1 {
```

```
        assert have_umbrella();
};
```

This creates the necessary hook into the code for the Testers' Assistant to check the above specification with respect to the state element have_umbrella.

Using the specifications to test the implementation of a protocol means that writing the specification could be an iterative process as the program develops. However, since it is also possible to write a general specification for a protocol regardless of the implementation, there is a need for a methodology for adding the implementation specific information to the general requirements specification. There are two general approaches to adding this information to the specifications. The general specification could constrain the programmer to certain variable names. For example, state information would have to be contained in the variable named "state" in the code. This approach limits the programmer's choices of variable names and could force the programmer to add more pieces of code to the implementation than are needed in order to determine the value of the state variable, possibly lowering the efficiency of the implementation. Another possibility is to allow the programmer to use any variable name, giving great flexibility for determining state transitions and their conditions. This approach limits the ability to develop a general template for writing specifications, but only with respect to location specifiers. The latter approach seems to be the most usable because the location specifiers in the TASpec language make adding implementation specific information to general specifications simple. This places the fewest restrictions on the programmer with regard to the design of the implementation. And, it does not restrict the specifier's freedom to write the specifications because the location specifiers do not affect writing the specifications themselves beyond creating or deleting elements based on events in the execution.

Chapter 5

Specifications in TASpec

This chapter explores approaches to writing specifications with the TASpec specification language. It also explains the constructs of the language and gives examples for using them. The last section uses an example to attempt to clarify even further the description of constructs in the language and how they are used.

5.1 General Ideas for using TASpec

There are a number of approaches to writing specifications for systems. This section explores several of them with respect to using them with the TASpec language.

One approach treats the program and its specifications as a set of objects or abstract data types. In this approach, the specifications indicate the data each object contains and what actions can be performed on each object. This would work well in a program that could be viewed as a set of objects, but network programs usually work by moving from one state to another. Another problem with using this approach with TASpec is that it is not designed specifically for monitoring objects. To write the specifications from the viewpoint of a set of objects, an additional tool would probably be needed to facilitate writing the specifications using TASpec in terms of objects.

A second approach takes the functions in the program and specifies what the functions can and can not do. This approach, if used alone, forces the specifier to view the program only as a set of functions, giving a limited view of the abilities of the program. The specifier still has to write specifications for the code inside the functions in the program. Writing the specifications in terms of functions could easily be done using TASpec, but combining this approach with another approach would probably be a better use of writing specifications for functions. This method also pulls the specification and implementation together. Using this method on its own with TASpec would possibly be more useful in the design requirements phase of developing software.

Another approach is to treat the program as a series of steps or state transitions. The specifications tell how to get into each state by specifying what the previous state(s) can be and what the values of variables should be to get to the state and what they should be after the state has been reached. This allows the specifications writer to focus on the desired flow of information and control in the program. It also leaves the flexibility to focus on the functionality of portions of the code.

Since network protocols usually involve making a connection and using it through a series of communications, it seems to be most useful to focus on writing the specifications as if they were describing state transitions for programs while adding a little functional description to the specifications. For example, in TCP, for a connection to be established, the two processes need to exchange a set of messages. As a message is received, the program checks to determine if the message contains the information expected. If it does, the program moves into the next state. The program can only change to a state if the proper conditions are fulfilled. This means that the proper input was received and the state transition is a valid one for the program.

In writing the specification as a set of state transitions, the specifier needs to look at a variety of details. He needs to determine the details of all state transitions which are allowed. Then he needs to determine the input values which allow the protocol to move from one state to the next. The specifier also needs to determine what other values are important in that particular state. Also important is determining which variables need to be reset before the state transition can be made. If this information is not all correct and detailed enough, the Testers' Assistant might not detect a state transition that should not be allowed in the protocol.

5.2 TASpec – the language itself

TASpec constructs consist of state elements, location specifiers, compounding elements, and event processing. The Testers' Assistant uses these constructs to determine the state of a program as described through specifications which are outlined by these constructs.

State elements indicate some portion of the state of the variables and function calls in the program. Multiple state elements can exist at the same time with the same identifiers and different arguments. Examples of state elements are : open(), closed(), ran(ftp), ran(tcp). The state elements open and ran are identifiers and ran is an argument. State elements are asserted (created), retracted, and used to assert and retract other state elements through location specifiers and preconditions. State elements are the basis of the specification language.

Location specifiers reference specific variables and functions in the code and are used to get information about the state of the variables in the implementation. Function call return values and parameters are also accessed through location specifiers. Location specifiers also reference variable value, usage, and declaration. (The Testers' Assistant does not yet support variable declaration location specifiers.) Location specifiers can have qualifiers on them to prevent the match to the location if the qualifier does not succeed. At run time the location specifier will try to match to all occurrences of the location indicated whether it is a variable reference or assignment or a function call and generate a location indicator for the Testers' Assistant to run with the specifications. Currently there is no way to indicate a specific call of a variable or function. An example of a location specifier for the system function call fopen would be:

```
location funcall fopen($path, $mode) returns $res{
assert opened($res, $path, $mode);
                retract closed($r, $p, $m);
}
```

In this example the arguments starting with $ are metavariables which, at run time, are given the values the actual function call arguments have. The metavariable $res is given this function call's return value. This location specifier states that when the function fopen is called, TASpec needs to instantiate the state element opened with the arguments $res, $path, and $mode which have the values from the actual function call.

Metavariables can even reference items within structures, as shown below:

Given the structure: struct { int new_var; } x, y;
And the location specifier:

```
location assign $a->new_var returns $val {
        assert got_val();
};
```

The location specifier will match to all assignments to a variable with the name new_var within a structure. So, every time new_var is assigned a value within variables x or y, this location specifier will match and the state element got_val() will be asserted.

Compounding operators consist of arithmetic, relational, logical, and temporal operators. Arithmetic operators consist of actions such as addition, subtraction, and multiplication or data values. Relational operators consist of comparison operators like less than, greater than and equal to. And, Or, and Not are examples of logical operators. Temporal operators introduce a history or time remembering aspect to the specification. They allow the specifier to state, for example, that state element "a" must exist until state element "b" is asserted. Other temporal operators are eventually, before, and logical implication. Arithmetic and relational operators are used to express constraints on data values. Logical and temporal operators are used to derive new state elements from combinations of existing state elements. All are used in defining preconditions and invariants.

The Testers' Assistant is event driven. The events are invariants, preconditions, and assertion and retraction of state elements. When a location specifier matches the code, the event(s) indicated in that specifier occur. Invariants limit how the state can be changed by other events. For example, an invariant might say that two state elements with the identifier car() can not exist at the same time even if they have different arguments, as shown below.

```
car($x) and car($y) implies $x = $y;
```

Pre-conditions are logical expressions defined through the compounding operators. Preconditions activate other events if the logical expression they define is satisfied.

5.3 Simple Example

As an example, examine this C language program that uses the atoi() function to take two characters and add them together. (Note: atoi() takes a character and converts it to an integer).

The program is:

```
main(){
        char x;
        int a, b;
        x = '6';
        a = atoi(x); /* a is assigned the integer value 6 */
        b = a + a;
}
```

A set of specifications for this program could be :

```
/*These location specifiers will hook into the code for the program and get the values indicated. */
location assign x returns $val {
        assert x($val); };
location assign a returns $val {
        assert a($val); };
location assign b returns $val {
        assert b($val); };
location funcall atoi( $ch1 ) returns $val {
        assert convert($val, $ch1); };

/* Preconditions to check various results in the code */
/* If a got the results of the atoi function call, then a_okay() */
a($w) and convert($w, $y) {
        assert a_okay(); };
/* If b got twice the value of a and a was okay, then b_okay() */
a($l) and a_okay() and b($n) {
        if $n = 2 * $l {
        assert b_okay();
        retract a($l);
        retract b($n); }; };

/* Invariant to state variable b must be found okay by the time the program quits */
eventually b_okay();
```

The last precondition has a precondition within it which states the \$n metavariable must be twice the value of the \$l metavariable. The first two preconditions do not have additional preconditions because when the Testers' Assistant tries to satisfy these preconditions, it will only succeed if it finds state elements whose arguments match because the metavariables for these state elements are the same. Because of the invariant, if the assignment operator assigned the wrong value to the variable b, causing the b_okay() assertion never to be made, the Testers' Assistant would report that the program did not conform to the given specifications informing the tester that there was something wrong with the implementation. This is because of the temporal operator "eventually" which states that at some point in time, the event after it must have been asserted for the program to have executed in conformance with the specification.

The specifications for the above program are a little simplistic, but they are useful for showing the difference between location specifiers, preconditions, and invariants. This was not a good example of writing requirements specifications . These specifications were written for the purpose of providing an example of types of elements and structures that can be used in TASpec. The next chapter provides a more complete example of writing specifications for a program and using them to test the implementation.

Chapter 6

The Alternating Bit Protocol

The well-known alternating bit protocol provides a thorough but uncomplicated introduction to using the Testers' Assistant to test a C language implementation of a specification. This chapter gives a brief description of the alternating bit protocol, describes specifications for parts of its implementation, and describes how the Testers' Assistant caught flaws planted in the implementation. This chapter also describes how to use the Testers' Assistant to test an implementation against its requirements specification.

6.1 Description

The alternating bit protocol is a one-way message passing protocol. The sending client process sends messages to a receiving server process. The sender and receiver verify the expected message was passed correctly by means of sending a checkbit along with the message that flips every time a new message is sent.

The receiver knows that it received the message it was supposed to if the check bit with that message is the same as the checkbit it was expecting. If the message is the one the receiver was expecting, it updates the next check bit it is expecting to the reverse of the previous one (for example, if the expected bit was 1, it is now 0 and vice versa.). When a message is received, the receiver sends an acknowledgement of that message whether it is a valid message or not by sending the received checkbit back to the sender.

The sender knows that the message was received if the acknowledgement bit it gets from the receiver is the same as the one on the message it just sent. If the check bits are not the same, the sender resends the message it just sent with the same check bit it just sent.

6.2 Specification and Implementation

6.2.1 State Diagram

To write the specifications for the alternating bit protocol, we first draw a state diagram for each side of the protocol. We assumed the sender and receiver would be separate executables with their own source code. We will only look at the diagram, specification, and implementation of the sender side. The receiver sides are included in Appendix 1.

We define the states READY, MSENT, and ACKRCVD to indicate that the server is ready to send a message, has sent a message, and has received acknowledgement of the message, respectively. The sender starts in the ready state.

So, a possible state diagram for the sender's side of the communication process is:

```
        READY<--
          |     \
          v      |
  ----->MSENT    |
 /        |      |
|         v      |
|      ACKRCVD   |
|        / \     |
--------/   \-----
```

Next, we decide how the sender gets from one state to the next.

To get from the READY state to the MSENT state, the sender has to get a message from the user, and the message has to be sent to the receiver successfully. This does not mean that the sender has to have received a reply, it just means that the process to send the message has to appear from the sender's end to have succeeded.

To move from the MSENT state to the ACKRCVD state, the sender server needs to have received the acknowledgement sent by the receiver.

¿From here the ACKRCVD state goes back to the MSENT state if the acknowledgement value is not the one expected and it has to resend the message. The ACKRCVD state goes to the READY state if the acknowledgement bit was correct and the check bit is flipped appropriately.

6.2.2 General Specifications

The diagram and the description of how to change state provide the information needed to write some general specifications for the sender side of this protocol. We will need a little more information to hook the specifications to the implementation through location specifiers.

To prevent jumping past required states, we need some invariants which describe the state progression. The invariants shown below do that.

```
(state_assign(READY) implies sec_tuple(BEGIN, 0, 0)) or
    (state_assign(READY) implies sec_tuple(ACKRCVD, rcv_ack, ack_rcvd));

state_assign(DONE) implies sec_tuple(READY, flip_bit, corr_mnum);

(state_assign(MSENT) implies sec_tuple(READY, flip_bit, corr_mnum)) or
    (state_assign(MSENT) implies sec_tuple(ACKRCVD, rcv_ack, ack_rcvd));

state_assign(ACKRCVD) implies sec_tuple(MSENT, send_msg, msg_sent);


sec_tuple(READY, flip_bit, corr_mnum) until
    (sec_tuple(DONE, send_msg, msg_sent) or
    sec_tuple(MSENT, send_msg, msg_sent)
    );

sec_tuple(MSENT, send_msg, msg_sent) until sec_tuple(ACKRCVD, rcv_ack,
ack_rcvd);

sec_tuple(ACKRCVD, rcv_ack, ack_rcvd) until
    (sec_tuple(READY, flip_bit, corr_mnum) or
     sec_tuple(MSENT, send_msg, msg_sent)
    );
```

The first four invariants say that if the process is trying to get into the state defined by the state element state_assign(), the process has to be currently in the state defined by the state element sec_tuple(). So, if the process is trying to get into the MSENT state, it must currently be in the READY or ACKRCVD state. The last three declare that the state defined by the state element sec_tuple() before the until has to exist until one of the sec_tuple() states after the until. So, the process has to be in the READY state until it goes to the MSENT or the DONE state. In other words, the process can not go from the READY state to the ACKRCVD state. If this happens in the implementation, the Testers' Assistant will notice and inform the tester that the invariant was violated.

The user-defined state element sec_tuple() describes the current state and how that state was achieved. So, the state element sec_tuple(MSENT, send_msg, msg_sent) says that the process is currently in state MSENT and it got there through sending the message and having the send succeed from the sender's side. The state element state_assign tells the state the process is trying to get to from the current state.

Notice, in the above invariants, we created two more states for the sender side of the protocol. These were for the purposes of writing the specification. The BEGIN state is not really needed but it allows the process to cleanly begin checking the assertions and events. The DONE state is needed. If the process does not reach this state by the time it quits, the tester knows that there was an error in the implementation somewhere. To state that the process must eventually end up in the DONE state, we use the below invariant:

```
eventually sec_tuple(DONE, send_msg, msg_sent);
```

We also do not want the process to be able to try to get into two states at the same time so we create an invariant that states that this should not occur.

```
state_assign($x) and state_assign($y) and $x = $y;
```

Now, we are ready to write the preconditions which process events and move the process from state to state. We will not put all of the necessary preconditions for the sender's specifications here, the rest can be found in appendix 1. This first precondition moves the process from the ACKRCVD state to the READY state:

```
state_assign(READY) and sec_tuple(ACKRCVD, rcv_ack, ack_rcvd)
```

```
  and flip_bit() and corr_mnum(1)
{
retract state_assign(READY);
retract flip_bit();
retract corr_mnum(1);
retract sec_tuple(ACKRCVD, rcv_ack, ack_rcvd );
assert sec_tuple(READY, flip_bit, corr_mnum );
};
```

This precondition states that the process must be in the ACKRCVD state (sec_tuple(ACKRCVD, ...) when trying to get into the READY state (state_assign(READY)), the check bit must have been flipped (flip_bit()), and the sender must have received the expected check bit from the receiver(corr_mnum(1)). When all these state elements exist, then the precondition can execute the events inside it. The events inside clean up and move to the next state. The retracts prevent the state elements that got the process to this state from hanging around and getting the process to this state again without the events actually having occurred. Naturally, we also need to get rid of the current state if we are moving to a new state, so we do that. Lastly, we assert the current state's information.

The rest of the preconditions are similar to this one. They take the conditions we decided on earlier for getting from one state to the next and use them to declare the preconditions that allow the process to get from one state to the next.

6.2.3 Location Specifiers and Implementation

The next step is to link these specifications to the actual implementation of the sender's side of the alternating bit protocol. For this we need location specifiers which reference actual function calls and variables in the implementation source code. We will just include a few of the location specifiers here. Refer to Appendix 1 to see them all.

In the source code, we find there exists a variable named "state" which actually stores the information for the implementation about what state the implementation is in, so to get the state_assign state element asserted properly, we write a location specifier which catches when this variable is assigned:

```
location assign state returns 5  {
    assert state_assign(MSENT);
```

```
};
```

To make the specifications more readable, we have declared one location specific for each possible value state could be assigned rather than asserting the state element with the value that was assigned to state. This is so that we can take the value and translate it into a word.

We also have to deal with sending the message. The implementation sends the actual message by using the socket function call sendto(...). So, the specifications get this information by describing a location specifier for this function call, shown below.

```
location funcall sendto( ... ) returns $val if $val > -1
    {
    assert send_msg(); /* event */
    assert msg_sent(); /* condition */
    };
```

This location specifier has a quantifier attached to it that states that if the sendto() function did not return an error ($val ¡= -1) then the process can assert that it did try to send the message (send_msg) by calling the function sendto() and the local send (msg_sent) of the message did succeed (sendto returned ¿ -1 value). The "..." within the parentheses says that the specification does not need the value of the arguments of the function call - it only needs to know that the function was called.

The specifications need location specifiers for all other actions in the implementation that affect the state elements described in the specifications. So, to assert the flip_bit state element, we need a location specifier that references the actions(s) that caused the bit flip to occur. The specifications also need location specifiers to indicate a received message, regardless of whether the message was the one expected or not. Location specifiers are needed for all other events that cause state changes in the specifications which are effected by the actual implementation. Many state elements can be asserted by a combination of other state elements in existence.

Once all the location specifiers are written we can use the Testers' Assistant to test the implementation against the specification. The next section tells how to do that.

6.3 Using the Testers' Assistant

In order to use the Testers' Assistant to test the implementation against the specification, there are a couple of programs to run and a few modifications to make to the code before the actual testing can be done.

First, run the specification file through the TASpec executable. This produces two files for the Testers' Assistant to use. The first is a .loc file which tells the processor what to do with the location specifiers and how to put them into the implementation. The second is a .pl file. This file is used in the actual testing process. The Testers' Assistant is written in prolog, so the specifications need to be turned into prolog for the Testers' Assistant to deal with them. The .pl file needs some modification once it is created. This is a result of the Testers' Assistant's current state of completion. (PUT IN APPENDIX —- All the ta_and prolog declarations need to be checked to see that they have only two arguments and the appropriate parentheses added if not.)

The Testers' Assistant preproccesses the source code to add the modules which record when the location specifiers match. Since the Testers' Assistant preprocessor currently does not recognize C language defines and includes, the tester will need to add function prototypes to replace the include statements and an enum statement for the defines statement to allow the preprocessor to add the modules to the code. Once the preprocessor has modified the code, the tester needs to add his includes and defines back into the original source code.

Now, the modified source code can be compiled as if it had not been changed to produce an executable for the tester to test. As the executable runs it writes to a file which tells which location specifiers matched in the execution and the values they matched with.

To test the implementation against the result of running the code with the specification information in it, we take the original .pl file and a supplementary file (called engine.pl) and put them into prolog. Then we put the information from the file written by the executable into prolog and see which invariants were violated by the specification.

The next section describes the output from flawed programs and what the Testers' Assistant said about them.

6.4 Flaws Caught

Because the alternating bit protocol is uncomplicated, it was necessary to seed flaws into the implementations in order to test the Testers' Assistant by means of the written specifications. Appendix 1 has the source code for the implementations with the flaws in them and the Testers' Assistant's results of testing these programs showing that the flaws were caught.

In one case the sender program was changed so that it did not wait for the acknowledgement form the receiver, but continued to send messages to the receiver without the acknowledgement. The Testers' Assistant caught this skipped state and told which state was skipped.

In another example, the sender program was altered so that its sent message numbers would be out of synchronization from the receiver's expected message numbers. Since the program was forced to be not able to get synchronized with the receiver again, this caused the program to endlessly loop, forcing the tester to kill the process manually from the shell. The Testers' Assistant caught this flaw too, but it was not able to indicate where the flaw was. It stated that the specifications never reached the DONE state. This indicates that the specifications need to be written carefully so they indicate what state the specifications need to end in. Testing the implementation against the specifications can show that the program did not complete properly. This also shows how the Testers' Assistant can catch possible flaws in the protocol. This specification only caught the flaw because it expected the program to end in the DONE state. This indicates that the code was doing exactly what the specifications told it to do - keep trying to send the message until the correct reply bit is received. The protocol should probably have set a limit on the number of times to try to resend the message, so this problem could not occur. (The Testers' Assistant only works as well as the specifications it is given.) The Testers' Assistant was able to catch the flaws seeded into the program source code.

Chapter 7

File Transfer Protocol

Since the two-way communications File Transfer Protocol (FTP) client and server are separate executables usually residing on different machines, specifications for the client can be written separately from those for the server. We first look at the client implementation because the client protocol implementation is less complex than the server implementation. This chapter describes FTP specifications for the client side, some possible flaws in an FTP client, and how they were caught by the specifications. Also included are some ideas for possible flaws in the server side of FTP that specifications for the server could catch.

7.1 FTP Specifications

Since FTP is a communications protocol which creates a connection and passes messages by means of that connection, using a state-based specification seems logical. First, the "states" for the ftp client are determined. Next, determine the events and conditions for FTP to be in each "state". For clarity, we choose to describe this information in the form of a state diagram where the path is the means of getting from one state to the next. Then, this information can be used to write a requirements specifications for the client side.

Since there is a connection to a remote server, we choose to treat the FTP client as having a connected and an unconnected state. Each command takes the program out of the connected or unconnected state to an "executing the command" state. A different set of events and conditions is required to allow the state change to occur depending on which state the command is trying to execute from. So, all commands which access the remote server require that the client be in the connected state before any of them can even try to execute. This means, for example, for an ftp put command which needs to get data from the connected machine, the specifications would not allow the executing the command for this state to be asserted unless the process were in the connected state.

Another way to determine states for this program is to treat it as having only one main state, a waiting state, which waits for commands to be executed. This would mean that information about the state of the connection would have to be stored as a condition variable. It would be one of the conditions checked when commands are executed.

Neither of these two options for determining states for the ftp client has much relation to the implementation. We chose the first one because there are several paths for getting to the connected state which seem to be described more clearly by moving form an unconnected to a connected state.

In order to achieve the connected state in the FTP client correctly and to be able to indicate when there were flaws, it is necessary to break the movement from UNCONNECTED to CONNECTED into two paths and sometimes 3 steps.

Below is a partial state diagram for the FTP client breaking down the states for getting into the connected state and shows the states for a couple of the ftp commands:

```
   ---------------------------UNCONNECTED<-----------------
   |                         ^    / \    ^                 \
   v                        /    /   \    \                 \
CMD_QUIT                   /    v     v    \                 \
   |   ^            CMD_LINE_OPEN     CMD_OPEN                \
   v    \|              / \             / \                    \
  QUIT   |             /   \           /   \                    \
         |            v     \         /     v              CMD_DISCONNECT
         | CMD_LINE_LOGIN    \       /     CMD_AUTOLOGIN        ^
         |              \     \     /      /                    |
         |               \     \   /      /                     |
         |                \-->  v v    <-/                      |
         ---------------------CONNECTED--------------------------
                            ^  / .... ^ \
                           /  /        \  \
                          /  v          \  v
                       CMD_PUT         CMD_GET
```

For all the other commands that need to be the process to be in the connected state to work, the diagram for them would be the same as for that of CMD_PUT and CMD_GET. Commands that do not need the connected state to execute would be similar to CMD_QUIT, only they would return to the CONNECTED or UNCONNECTED state depending on which state they were in before. When coming from the UNCONNECTED state commands such as get and put do nothing, so there is really no need to put them in the diagram. However, for completeness sake, they would look the same as if they were coming from the CONNECTED state, except they would come from and return to the UNCONNECTED state or even have no path out of the UNCONNECTED state.

In following a state diagram for the requirements of a system, a process can only move to a state if the conditions indicated on the state diagram are met. The FTP specifications use an assertion called a sec_tpl which stands for state, event, condition tuple to describe the transition from one state to the next. This tuple indicates the current state and the event(s) and condition(s) which brought the process to this state from the previous state. This is a similar state element to the sec_tuple state element described in the alternating bit protocol.

As an event or condition occurs in the implementation an assertion is made through location specifiers to indicate this. When all the events and condition assertions exist which allow the system to go from one state to the next, the next state elements are asserted.

Once the FTP client is connected it sits waiting for commands from the user to send to the server. When it receives a command to send, the client changes state briefly to execute the command only to return to the connected state after command completion. In the specifications, we deal with this by creating a command execution waiting state (this is the state CMD_PUT, CMD_GET, etc.) for each command. When a command is called, the process goes into a waiting state for that command. The events and conditions to get out of this state require that the command execute "properly". Unlike the alternating bit protocol, there is no state_assign state element for these specifications because it is not needed. It can be asserted by combining in a precondition all necessary to reach that state if it is needed. This is because FTP does not assign a state variable.

For example, for the get file command, to return to the waiting for command state: the proper command has to be sent to the server, the server must send a reply back, and the file data sent by the server must be received and put into the proper file on the client. The conditions for other commands to finish "properly" are similar.

7.2 Security Flaws

In the FTP client two possible security flaws stand out. Getting a file from a server and sending a file to a server could cause a security problem on the client side machine.

In getting a file, the implementation needs to put the received data only in the file indicated by the get command. Overwriting the wrong file could mean that the get command could accidentally erase some information needed for security on the local machine or the server could send commands as data which, if put into the wrong file by the client, could give unauthorized access to the machine.

Sending a file to a remote machine correctly means that only the data in the file indicated should be sent to the remote machine. A mistake in this could mean that something like the password file on the local machine could be sent to the server machine, creating a possible security problem on the local machine.

7.3 Catching those Flaws

In order to catch these security flaws in the FTP client, the specifications for moving out of the command waiting state need to indicate the events and conditions for the command to execute properly. So, in general, for a put command to execute properly a command must be sent to the server, a reply from that command must be received from the server, and the specifications must check that the file whose data was sent was from the same file as the one indicated on the command line. So, the specification for moving out of the put command waiting state is:

```
sec_tpl(FTP_CMD_PUT, $ev, $co) and reply($x)
   and command('S','T','O','R') and not sununique()
   and file($f) and put_local($l)
   {
   ( $f = $l )
    {
    retract sec_tpl(FTP_CMD_PUT, $ev, $co);
```

```
    retract reply($x);
    retract command('S','T','O','R');
    retract file($f);
    retract put_local($l);
    assert sec_tpl(FTP_CONNECTED, reply, command_PUT);
    };
   };
```

This precondition says that the process must be in the CMD_PUT state, a reply must have been received (reply($x)), the proper FTP server command must have been sent (command('S','T'...)), a file must have been opened(file($f)) and the put command line argument must have been received (put_local($l)) for the inner precondition to even be looked at. The inner precondition states that the file opened ($f) must have the same name as the file given to the put command ($l). If all this succeeds, the specifications clean up the processes' state space by retracting all those state elements and asserting the new state.

To attach this to the implementation a number of location specifiers are needed which catch these actions as they occur during execution. Some of them are:

```
location funcall getreply( ... ) returns $val
    {
    reply($y) { retract reply($y); };
    assert reply($val);
    };
location funcall fopen($local, $mode)
    {
    file($x) { retract file($x); };
    assert file($local);
    };
location funcall sendrequest($cmd, $local, $remote, $p)
    {
    put_local($x) { retract put_local($x); };
    command($w, $x, $y, $z) { retract command($w, $x, $y, $z); };
    assert put_local($local);
```

```
assert command($cmd[0], $cmd[1], $cmd[2], $cmd[3]);
};
```

The first two location specifiers are simple. They state that when that particular function is called during execution, assert the corresponding state element. The first line in those two and the first two in the last location specifier tell the Testers' Assistant to remove all other instances of the state element to be asserted. This is to keep only one instance of each of these in existence at a time. The last location specifier gets two pieces of information out of the user defined function – the put command's argument, which was a file name, and the first four characters of the actual command sent to the server. These characters should be unique for all the different client commands that contact the server.

When the fopen call, which opens the file to be sent, is changed in the implementation to the specific file name of "ptemp" and the put command is tested by doing a put on the file name "ptemp1", the Testers' Assistant caught the improperly executed put command which sent the wrong file, "ptemp" instead of "ptemp1", to the server.

The get command in ftp works in a similar fashion to the put command. It has a similar specification that needs to compare file names to allow the get command to move back to the FTP_CONNECTED state. When the fopen function for the get command was changed to reference a specific file name, and an ftp get command was executed on a different file name, the Testers' Assistant caught the flaw. This flaw overwrote the wrong file on the client.

7.4 Server Side Possibilities

Because the server has more responsibility in permitting connections, there are more possibilities for flaws to occur in the implementation. In addition to sending and receiving data, like the client, the server also deals with accepting connections and allowing the directories on the server side to be accessed. In addition, many ftp servers allow the client to run custom commands on their machines. To allow the client to connect to the server, the server must check that the client's username and password fulfill the conditions for connection. The specifications could determine if the connection was made properly by checking that the implementation verified the user name and password against the stated conditions and used the proper files for dealing with the user name and password.

The server also allows the client to move around in the server host's directory structure, listing the files in the directories it sees. The server code must make sure that the client is not able to move out of or see the contents of directories the maintainers of the server machine want to be hidden from ftp clients. The specifications should check that with every command that accesses the server's directories, the implementation checked to see if the command is allowed to access the directories and files they are trying to see. It also needs to make sure the implementation is dealing with only those directories and files.

If the server allows custom commands to be run on the server from the client, the server must check the commands carefully. The server must check that the command given is one it allows to be run. It should also check that only the command and the necessary arguments are sent, and nothing more. An addition to the command could be another command which could delete the password file, or change file and directory access permissions. The specifications can check that the implementation checked these conditions. In addition, there is the possibility of a poorly written custom command creating a security hole. From the implementation side, it is less likely that a specification would catch this.

With all the command which access the server machine and do actions on the server machine, there is the possibility of a flaw in the implementation of these commands causing a security problem on the outside machine allowed to access it.

Chapter 8

Future Work, Results, and Conclusion

8.1 Future Work

This chapter describes future work that can be done with testing programs using specifications, TASpec and the Testers' Assistant, and other ideas. It then summarizes the results of this work and presents conclusions about using specifications to test programs.

8.1.1 TCP

In Appendix 4 are the beginning of specifications for the Transmission Control Protocol (TCP) for making connections between two machines. TCP is a good candidate for a program whose implementation flaws could cause security problems on the machine it runs on.

Biswaroop Guha (Guha 1995) describes security holes he discovered in a specific TCP implementation ¡ which one¿. The flaws he discovered all occurred in the connection set up process of the protocol. These flaws are the ability to do IP Spoofing, improper TCP state transitions, and timer problems. Each of these flaws could be caught by means of specifications for the protocol.

IP spoofing exploits applications which determine user authentication based on IP addresses. Part of the process of IP spoofing requires preventing the host whose address is being spoofed (mimicked), victim A, from responding to packets from the victim of the spoofing, victim B. The other part consists of the intruder being able to guess the packet sequence numbering system of victim B's protocol. The intruder first blocks victim A from responding to any packets from victim B. Then the intruder sends a number of requests to victim B to see if he can determine victim B's packet sequence numbering system. Lastly, the intruder sends a request to victim B as if it were from victim A and responds to victim B's packet with a sequence number the intruder has guessed will be next. If the intruder is correct victim B will make the connection, giving the intruder the same privileges as victim A because victim B thinks the intruder is victim A.

The ability to block victim A from responding to victim B's packets is called "wedging". In this case, the ability to wedge a victim results from the ability to force improper state transitions in the TCP implementation. Improper state transitions occur when the intruder can force the host to go from one state to another without the expected intermediate state transitions. This will cause the host to not know how to get out of that state and so it will just sit there waiting for information it is not going to get.

The timer problem Guha found occurs when the timer is turned off too soon. When the process of setting up a connection is in progress, a connection establishment timer is turned on. If the connection is not established in time, TCP quits trying to make the connection. However, when two processes are trying to connect to each other at the same time, a simultaneous open, the timer is turned off before the connection is established. This allows the intruder to not respond to the host, causing a denial of service for that host.

Specifications could be used to catch the flaws in the implementation which allow the wedging and timer problems to occur. Since wedging results from the implementation allowing the process to skip states, a specification which checked that the process did not skip states would show the flaw. The timer problem could also be caught by checking in the specifications that the timer gets set and stays on for its specified length of time. The IP spoofing problem consists of wedging and sequence number guessing. As described above, wedging could be caught by specifications. The ability to guess sequence numbers on the part of the attacker depends partly on the quality of the sequence numbering algorithm and the ability of the intruder to predict sequence numbers. Checking the quality of the sequence numbering algorithm could possibly be done through specifications, but a mathematical proof might do the job better and even possibly take into account the skills of the intruder.

The beginnings of specifications for TCP included in appendix 4 should catch the improper state transition in TCP that allow wedging to occur. The specifications need to have the location specifiers added which check directly a specific implementation. These specifications partly follow the state diagram for TCP given in the RFC requirements specification. They only deal with the connection establishment phase for TCP.

8.1.2 TASpec and the Testers' Assistant

The TASpec language and the Testers' Assistant, while effective for the ftp and alternating bit protocol specifications, could be enhanced by additions to the whole package. These enhancements deal with making tools to expand the possibilities for using TASpec and making the Testers' Assistant easier to use.

Its state-base approach places some limitations on TASpec's flexibility for testing a large variety of programs. Even with its state-based orientation, specifications for a program with an abstract data type or object oriented approach can be written using TASpec. However, a tool to allow the tester to write the specifications with an abstract data type or object oriented perspective would make the Testers' Assistant a more versatile tool. The ability to write specifications in an abstract data type manner would allow the specifications writer more choices for the approach to take in writing the specifications. The Testers' Assistant could be expanded by a tool which translates specifications written in an abstract data type or object oriented manner into TASpec language specifications for use by the Testers' Assistant.

Unlike some specification languages, TASpec does not have the ability to verify that the specifications written with it are correct. Another enhancement could be a tool which would translate specifications written in a verifiable language. This would increase the completeness of the specifications checked by the Testers' Assistant and consequently decrease the possibility of human error affecting the correctness of the specifications.

If the specifier takes the approach of writing the specifications in a state oriented fashion, a tool which can take all the information contained in the state model and turn it into specifications would be useful. It would help the specifier in writing the specifications and make it more likely that the specifications were complete with regard to the allowable state transitions.

Like most testing procedures, using specifications to test is subject to human error. This approach is dependent on the tester writing the specifications correctly. The specifications need to be detailed and thorough. The tester needs to specify every condition for changing to a state, including what variables should not be set and what variables should be cleared. A tool which would help reduce the possibility of human error would be a series of questions which would ease the specification writing process.

The Testers' Assistant could be expanded to include a library of flaws as described by Joel Dodson (Dodson 1995). This would expand the testing procedure for the software. Having flaws loosely categorized and knowing what kinds of flaws different varieties of network protocols are possibly susceptible to would further enhance the testing process. This would be especially useful with the flaws which can not be found by examining the specifications for the protocol, such as sequence number guessing. This would also allow the specifications to focus on the protocol's requirements rather than general flaws which could occur in any program.

Another problem with using specifications to test the implementation is that deciding what test data to use is partially dependent on knowing the flaw or knowing the specifications. It is possible for a flaw to exist and not be found because the proper test data was not used. Further development of a means of generating test data from specifications and the implementation would enhance the testing process.

Currently, the Testers' Assistant needs some manipulation to make it work completely. Fixing these missing pieces and problems within the source code would immeasurably enhance the usability of the tool. Because the declaration location specifier is not yet implemented, the specifications need some additional assertions which initialize the state elements which relate to the declaration of variables. Also, after the prolog version of the specifications has been created by the Testers' Assistant, it has to be modified to work properly because it is missing some parentheses in its output.

8.2 Results and Conclusion

The Testers' Assistant was able to catch flaws in implementations by testing them against the specifications. When a flaw causing skipped states was introduced into the implementation, the Testers' Assistant identified this flaw and informed the tester. The Testers' Assistant also caught what can be thought of as a flaw within the protocol itself when it caught that another flaw caused an endless loop in the implementation of the alternating bit protocol. After testing the alternating bit protocol, the client side of the ftp protocol was examined. The specifications for it also helped to catch flaws seeded into the implementation. Once the specifications are written, testing against the implementation is an uncomplicated process requiring only a set of tests, like any other testing process. Testing the implementation against the specification was an effective means of enhancing the testing procedure for a program. It cleanly defines what the program is supposed to do and checks that those boundaries can not be crossed.

This method of testing an implementation for flaws will not catch all the flaws in an implementation. A way to combine this method with other methods of testing implementation would be enhance the complete testing process.

There are some problems to this approach to testing a program for flaws. Some flaws can not be discovered even by testing the implementation against the specification. For example, allowing sequence number guessing to succeed can not be detected by testing against the specification because the specification does not usually have any information about what an attacker's abilities might be. Also this approach most likely will not detect flaws that occur when system functions are misused unless the tester remembers to check for this in the specifications. Joel Dodson (Dodson 1996) has begun to develop a library of flaws with specifications for them written in TASpec. This library could be used to detect general flaws that would not be discovered through the specification.

The purpose of this thesis is to show how specifications written in TASpec can be used to test the implementation of a piece of software to catch security flaws in it. The Testers' Assistant, along with TASpec, effectively enhances the testing process for a piece of software. Writing the requirements specifications for the program in TASpec allows the Testers' Assistant to use the specifications to check the implementation for its conformity to requirements. While it is effective in catching flaws which result from deviations from the requirements, this method should not be expected to catch every flaw in the code. This methodology of using specifications to test the implementation should be used as a vital part of the regular testing procedure rather that as a substitution for it.

AppendixThe Alternating Bit Protocol

The alternating bit protocol is a one-way message sending protocol which validates that messages are received by means of a checkbit. The sender and receiver processes start with their check bit set to 0. When the sender process sends a message, it sends its current check bit with the message. When the message is received, the receiver verifies that the check bit attached to the message is the same as the one it is expecting. If it is the same, the receiver sends an acknowledgement of the message back to the sender with its current check bit and then flips its current check bit from 0 to 1 (or 1 to 0). Otherwise, the receiver just sends an acknowledgement with its current check bit and ignores the message. The sender, on receiving the acknowledgement, verifies the received check bit against its current check bit. If they are the same the sender flips his check bit and is now ready to send another message. Otherwise the sender resends the last message. It assumes that the receiver never got the message it just sent, and goes back to waiting for an acknowledgement of the message. Below is an example of the communications which might go on between the two processes:

```
Sender(s.checkbit=0)         Receiver(r.checkbit=0)


        Message + 0
    ------------------------------?
r.checkbit=1 since
r.checkbit==0
                      Acknowledgement + 0
                ------------------------------
           s.checkbit = 1 since
     s.checkbit==0
```

The next sections contain an implementation written in C for the alternating bit protocol, specifications in TASpec for the protocol, and implementations with seeded flaws with the results of running them through the Testers' Assistant.

.1 The Code

/* This is the code for the receiver side of the alternating bit protocol */

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <netdb.h>

#define SERVER_CTL      5138
#define CLIENT_CTL      5139

#define         TRUE    1
#define         FALSE   0
#define         WAIT    2
#define         MRCVD   3
#define         READY   4
#define         MSNT    5
#define         ACKRCVD 6
#define         OKAY    7
#define         NEXT    8

#define         ACK             500
#define         DONE            501

int setup_sock(int);

void main ()
{
   int sockFD=0;          /*socket number*/
   int r_seq_num = 0;              /*number for msgs, alternates
   between 0 and 1 */
   struct sockaddr_in addr, from;   /*from is info about where recvd msg
   comes from and addr is where sent
msg is going to */
int from_length;  /*length of the from structure */
   int state=WAIT;              /*indicates the state of the program */
   int correct_msgnum;
```

```
   static struct pkt {
int type;
char msg[80];
} ack, in_msg;          /*msg that acknowledges msg that came */
/* in_msg is the received msg */

strcpy(ack.msg,"---ACK---");     /*initialize the ack message */
   ack.type = r_seq_num;

sockFD = setup_sock(SERVER_CTL); /*set up the socket */

   while (TRUE) { /*loop and wait endlessly sending and
receiving msgs until done msg rcvd */
     switch(state) {

case WAIT:            /* allowed to receive information */
/* set the receiving information */
from_length = sizeof (struct sockaddr);
from.sin_family = AF_INET;
from.sin_port = CLIENT_CTL;

/*receive the message */
recvfrom (sockFD, &in_msg, sizeof (in_msg), 0, (struct sockaddr *)&from, &
from_length);
state = MRCVD;
break;

      case MRCVD:
if ( in_msg.type == DONE) {         /*if msg.type is DONE, then */
    printf("received done message\n");
close(sockFD);                  /*close the socket and exit */
    state = DONE;
```

```
    exit (0);
}

printf("msg type is : %d   ",in_msg.type );

correct_msgnum = FALSE;
if ( r_seq_num == in_msg.type ) {  /*if received correct msg number
print out the msg */
    correct_msgnum = TRUE;
    printf ("Received-> %s\n", in_msg.msg);
    r_seq_num = (r_seq_num + 1 ) % 2;
    state = OKAY;
}

/* Sending ACK to server that it got the message*/
ack.type = in_msg.type;
state = NEXT;

/* set the sending information */
memcpy (&addr, &from, sizeof (struct sockaddr_in));
addr.sin_family = AF_INET;
addr.sin_port = CLIENT_CTL;

sendto(sockFD, &ack, sizeof(ack),0,(struct sockaddr *)&addr, sizeof(struct
sockaddr));
state = WAIT;
break;

     }  /* end switch statement */
   }  /* end while loop */
} /* end of program */
```

```
int setup_sock(int port)
{       char                    hn[80];
        struct hostent          *hp;
        struct sockaddr_in      addr;
        int                     sock;

        gethostname(hn, sizeof(hn));
        if ((hp = gethostbyname(hn)) == NULL)
                {
                perror("Couldn't get hostname");
                exit(-1);
                }
/*this command clears the variables */
        memset((struct sockaddr_in *)&addr, 0,
                        sizeof(struct sockaddr_in));
        memcpy((struct sockaddr_in *)&addr.sin_addr,
                        hp->h_addr, hp->h_length);

        addr.sin_family = AF_INET;
        addr.sin_port = port;



        if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) <= 0)
                perror("Couldn't get a socket");

        if (bind(sock,(struct sockaddr *)&addr, sizeof(struct sockaddr_in))== -
1)
                {
                perror("Couldn't bind control socket");
                close(sock);
                exit(-1);
                }
```

```
        return(sock);
}
```

/* This is the code for the sender side of the alternating bit protocol */

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>

#define SERVER_CTL      5138
#define CLIENT_CTL      5139

#define        TRUE    1
#define        FALSE   0
#define        WAIT    2
#define        MRCVD   3
#define        READY   4
#define        MSNT    5
#define        ACKRCVD 6
#define        OKAY    7
#define        NEXT    8

#define        ACK             500
#define        DONE            501

int setup_sock(int);

void main ()
{
   int sockFD=0;                       /*socket number*/
   int s_seq_num=0;                    /*number for msgs, alternates
```

```
 between 0 and 1 */
char server_name[80];              /*user entered server name */
struct hostent *hp;                /*info about machine to talk to */
struct in_addr rmt_machine_addr;   /*stores info about machine to talk to */
   struct sockaddr_in from, addr;      /*from is info about where recvd msg
         comes from and addr is where sent
msg is going to */
   struct pkt {
        int type;
        char msg[80];
        } stuff, in_msg;           /*stuff is msg to be sent and in_msg
is received msg */
   int state = READY;        /*tells the state of the program */
   int correct_msgnum;

int from_length;                   /*length of the from structure */



printf("Chat with: ");             /*get machine name to chat with */
   gets (server_name);
sockFD = setup_sock(CLIENT_CTL);    /*set up the socket */

hp = gethostbyname (server_name); /*info about where to send msg */
   memcpy (&rmt_machine_addr, hp->h_addr, hp->h_length);

   while (TRUE) {                     /*loop and wait endlessly sending and
receiving msgs until Control G typed */
     switch(state) {

case READY:                     /* allowed to send information */
printf ("<-Sending:");
gets (stuff.msg);
```

```
/* tells the both sides of connection to quit */
if (stuff.msg[0] == 0x07) {    /*if the msg is cntrlG then send
  msg to server to exit and then
exit itself */
    printf("User sent quit message \n");
    stuff.type = DONE;
    sendto(sockFD, &stuff, sizeof(stuff), 0, (struct sockaddr *)&addr,
sizeof(struct sockaddr));
    state = DONE;
            exit(0);
}

stuff.type = s_seq_num;   /*set message number for receiver to check */

    /* set sending information */
    addr.sin_family = AF_INET;
    addr.sin_port = SERVER_CTL;
    addr.sin_addr = rmt_machine_addr;

   /*actually send the message*/
sendto (sockFD, &stuff, sizeof(stuff), 0, (struct sockaddr *)&addr,
sizeof(struct sockaddr));
state = MSNT;
break;

      case MSNT:
/* Waiting for ACK from server that it got msg */
/* make this recvfrom nonblocking ??? */
/* add a time out here ??? */

    /* set receiving information */
```

```
    from_length = sizeof (struct sockaddr);
    from.sin_family = AF_INET;
    from.sin_port = SERVER_CTL;

recvfrom (sockFD, &in_msg, sizeof(in_msg), 0, (struct sockaddr *)&from,
&from_length);
/* check that it is an ack here ?? */
state = ACKRCVD;
break;

      case ACKRCVD:
printf("ack msg is : %d    ",in_msg.type );
if(in_msg.type == s_seq_num ) {   /*if got proper ack then can now send */
    correct_msgnum = TRUE;
                         /*another message*/
    s_seq_num = (s_seq_num + 1 ) % 2;
    state = READY;
}
else {                        /*else resend the message*/
    correct_msgnum = FALSE;
    sendto (sockFD, &stuff, sizeof(stuff), 0, (struct sockaddr *)&addr,
sizeof(struct sockaddr));
    state = MSNT;
}
break;

     }   /* end switch stmt */
   }  /* end while loop */
}  /* end of program */


int setup_sock(int port)
```

```
{       char                    hn[80];
        struct hostent          *hp;
        struct sockaddr_in      addr;
        int                     sock;

        gethostname(hn, sizeof(hn));
        if ((hp = gethostbyname(hn)) == NULL)
                {
                perror("Couldn't get hostname");
                exit(-1);
                }
/*this command clears the variables */
        memset((struct sockaddr_in *)&addr, 0,
                        sizeof(struct sockaddr_in));
        memcpy((struct sockaddr_in *)&addr.sin_addr,
                        hp->h_addr, hp->h_length);

        addr.sin_family = AF_INET;
        addr.sin_port = port;

        if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) <= 0)
                perror("Couldn't get a socket");

        if (bind(sock,(struct sockaddr *)&addr, sizeof(struct sockaddr_in)) == -
1)
                {
                perror("Couldn't bind control socket");
                close(sock);
                exit(-1);
                }
        return(sock);
}
```

.2 The Specifications

Below is the TASpec specifications for the receiver side of the alternating bit protocol.

```
/* File : rcv.tas
   File of specifications for the receiver.c program which is
   half of the alternating bit protocol protocol implementation.
   sec_tuple is a tuple containing the state, event to reach that
   state and the condition to reach that state.

   Note: fake values are asserted in the first WAIT state after the
        BEGIN state.  This is to begin the loop around process and
        make the sec_tuple asserts and the state assignments fit
        together properly.
*/
/*  from the #defines in the programs
#define         TRUE    1
#define         FALSE   0
#define         WAIT    2
#define         MRCVD   3
#define         READY   4
#define         MSNT    5
#define         ACKRCVD 6
#define         OKAY    7
#define         NEXT    8
#define         ACK             500
#define         DONE            501
 */

/**********************************************************************
  This section contains invariants for the specifications
**********************************************************************/
```

```
/* invariant states that only one state_assign can exist at a time */

state_assign($x) and state_assign($y) and $x = $y;

/* invariants assure the proper progression of states */

(state_assign(WAIT) implies sec_tuple(BEGIN, 0, 0)) or
    (state_assign(WAIT) implies sec_tuple(NEXT, check_mnum, bad_mnum)) or
        (state_assign(NEXT) implies sec_tuple(NEXT, flip_bit, corr_mnum));

state_assign(DONE) implies sec_tuple(MRCVD, rcv_msg, msg_rcvd);

state_assign(MRCVD) implies sec_tuple(WAIT, send_ack, ack_sent);

state_assign(OKAY) implies sec_tuple(MRCVD, rcv_msg, msg_rcvd);

(state_assign(NEXT) implies sec_tuple(MRCVD, rcv_msg, msg_rcvd)) or
(state_assign(NEXT) implies sec_tuple(OKAY, flip_bit, corr_mnum));

/* these temporal invariants help prevent jumping states */

sec_tuple(WAIT, send_ack, ack_sent) until sec_tuple(MRCVD, rcv_msg, msg_rcvd);

sec_tuple(MRCVD, rcv_msg, msg_rcvd) until
     (sec_tuple(DONE, rcv_msg, msg_rcvd) or
      sec_tuple(OKAY, flip_bit, corr_mnum) or
      sec_tuple(NEXT, check_mnum, bad_mnum)
     );

sec_tuple(OKAY, flip_bit, corr_mnum) until sec_tuple(NEXT, flip_bit, corr_mnum);

(sec_tuple(NEXT, check_mnum, bad_mnum) or sec_tuple(NEXT, flip_bit, corr_mnum))
```

```
until
sec_tuple(WAIT, send_ack, ack_sent);

/* This invariant is supposed to say process must end in DONE state */

eventually sec_tuple(DONE, rcv_msg, msg_rcvd);


/*****************************************************************************
***/
/* these are the main predicates and logic */
/*****************************************************************************
***/

assert sec_tuple(BEGIN, 0, 0);


/* 2 == WAIT */
location assign state returns 2  {
    assert state_assign(WAIT);
    };

state_assign(WAIT) and sec_tuple(BEGIN, 0, 0 )
    {
    retract state_assign(WAIT);
    retract sec_tuple(BEGIN,0,0);
assert sec_tuple(WAIT, send_ack, ack_sent);  /* this is to begin the */
/* loop of checking the state */
    };

/* send_ack is the event, ack_sent is the condition */
state_assign(WAIT) and sec_tuple(NEXT, $a, $b) and send_ack() and ack_sent()
```

```
    {
    retract state_assign(WAIT);
    retract sec_tuple(NEXT, $a, $b);
    assert sec_tuple(WAIT, send_ack, ack_sent);
    retract send_ack();
    retract ack_sent();
    };


/* 501 == DONE */
location assign state returns 501  {
    assert state_assign(DONE);
    };

/* in_msg.type == DONE
   this location msg gets put in the inst.c code after
the exit(0) so the location call is never noticed, I am
   not sure how to deal with that and this condition is
   probably not needed for the next predicate, so I am going
   to comment out this location variable for now.  Note: the
   msg_done assertion in sender.c works because it is an
   assign and not a check of a variable's value.
*/
/*
location variable in_msg.type returns 501 {
    assert msg_done();
    };
*/

state_assign(DONE) and sec_tuple(MRCVD, rcv_msg, msg_rcvd)
/*    and msg_done() */
{
```

```
    retract state_assign(DONE);
    retract sec_tuple(MRCVD, rcv_msg, msg_rcvd);
    assert sec_tuple(DONE, rcv_msg, msg_rcvd);
    retract msg_done();
    };


location funcall sendto( ... ) returns $val if $val > -1
    {
    assert send_ack();  /* event */
    assert ack_sent(); /* condition */
    };

location funcall recvfrom( ... ) returns $val if $val > 0
    {
    assert rcv_msg();  /* event */
    assert msg_rcvd(); /* condition */
    };

/* 3 == MRCVD */
location assign state returns 3 {
    assert state_assign(MRCVD);
    };

state_assign(MRCVD) and  sec_tuple(WAIT, send_ack, ack_sent ) and rcv_msg() and
msg_rcvd()
    {
    retract state_assign(MRCVD);
    retract sec_tuple(WAIT, send_ack, ack_sent);
    assert sec_tuple(MRCVD, rcv_msg, msg_rcvd);
    retract rcv_msg();
    retract msg_rcvd();
```

```
    };

location assign correct_msgnum returns $val
    {
    corr_mnum($x) { retract corr_mnum($x); };
    assert corr_mnum($val);
    };

location assign r_seq_num returns $val
    {
    assert flip_bit();
    };

location assign state returns 7     /* 7 == OKAY */
    {
    assert state_assign(OKAY);
    };

state_assign(OKAY) and sec_tuple(MRCVD, rcv_msg, msg_rcvd) and corr_mnum(1) and
flip_bit()
    {
    retract state_assign(OKAY);
    retract sec_tuple(MRCVD, rcv_msg, msg_rcvd);
    assert sec_tuple(OKAY, flip_bit, corr_mnum);
    corr_mnum($x) { retract corr_mnum($x); };
    retract flip_bit();
    };

location assign state returns 8   /* 8 == NEXT */
    {
    /*state_assign(MRCVD) or state_assign(OKAY);
    state_assign(MRCVD) {retract state_assign(MRCVD);};
```

```
    state_assign(OKAY) {retract state_assign(OKAY);};*/
    assert state_assign(NEXT);
    };

state_assign(NEXT) and sec_tuple(MRCVD, rcv_msg, msg_rcvd) and corr_mnum(0)
    {
    retract state_assign(NEXT);
    retract sec_tuple(MRCVD, rcv_msg, msg_rcvd);
    assert sec_tuple(NEXT, check_mnum, bad_mnum);
    retract corr_mnum(0);
    };
state_assign(NEXT) and sec_tuple(OKAY, flip_bit, corr_mnum)
    {
    retract state_assign(NEXT);
    retract sec_tuple(OKAY, flip_bit, corr_mnum);
    assert sec_tuple(NEXT, flip_bit, corr_mnum);
    };

location assign hp returns $hp {
   assert h_addr($hp);
};
```

Below is the TASpec specifications for the sender side of the alternating bit protocol.

```
/* File : send.tas
   File of specifications for the sender.c program which is
half of the alternating bit protocol implementation.
   sec_tuple is a tuple containing the state, event to reach that
   state and the condition to reach that state.

   Note: fake values are asserted in the first READY state after the
        BEGIN state.  This is to begin the loop around process and
        make the sec_tuple asserts and the state assignments fit
```

```
        together properly.
*/

/*  from the inet.h file for the programs
#define         TRUE    1
#define         FALSE   0
#define         WAIT    2
#define         MRCVD   3
#define         READY   4
#define         MSNT    5
#define         ACKRCVD 6
#define         OKAY    7
#define         NEXT    8

#define         ACK             500
#define         DONE            501
 */

/***************************************************************************
    This section contains the invariants for the specifications
***************************************************************************/

/* invariant states that only one state_assign can exist at a time */

state_assign($x) and state_assign($y) implies $x = $y;

/* invariants assure the proper progression of states */

(state_assign(READY) implies sec_tuple(BEGIN, 0, 0)) or
    (state_assign(READY) implies sec_tuple(ACKRCVD, rcv_ack, ack_rcvd));

state_assign(DONE) implies sec_tuple(READY, flip_bit, corr_mnum);
```

```
(state_assign(MSENT) implies sec_tuple(READY, flip_bit, corr_mnum)) or
    (state_assign(MSENT) implies sec_tuple(ACKRCVD, rcv_ack, ack_rcvd));

state_assign(ACKRCVD) implies sec_tuple(MSENT, send_msg, msg_sent);

/* these temporal invariants help prevent jumping states */

sec_tuple(READY, flip_bit, corr_mnum) until
    (sec_tuple(DONE, send_msg, msg_sent) or
    sec_tuple(MSENT, send_msg, msg_sent)
    );

sec_tuple(MSENT, send_msg, msg_sent) until sec_tuple(ACKRCVD, rcv_ack,
ack_rcvd);

sec_tuple(ACKRCVD, rcv_ack, ack_rcvd) until
    (sec_tuple(READY, flip_bit, corr_mnum) or
     sec_tuple(MSENT, send_msg, msg_sent)
    );

/* This invariant is supposed to say process must end in DONE state */

eventually sec_tuple(DONE, send_msg, msg_sent);

/*******************************************************************************/
/* these are the main predicates and logic */
/*******************************************************************************/

/* use this because location declaration has not been implemented yet */
assert sec_tuple(BEGIN, 0, 0 );
```

```
/* 4 == READY  */
location assign state returns 4  {
    assert state_assign(READY);
    };

state_assign(READY) and sec_tuple(BEGIN, 0, 0 )
    {
    retract state_assign(READY);
    retract sec_tuple(BEGIN,0,0);
    assert sec_tuple(READY, flip_bit, corr_mnum); /*this is to begin */
/* the loop of checking the state */
    };

/* send_ack is the event and ack_sent is the condition */
state_assign(READY) and sec_tuple(ACKRCVD, rcv_ack, ack_rcvd)
      and flip_bit() and corr_mnum(1)
    {
    retract state_assign(READY);
    retract sec_tuple(ACKRCVD, rcv_ack, ack_rcvd );
    assert sec_tuple(READY, flip_bit, corr_mnum );
    retract flip_bit();
    retract corr_mnum(1);
    };

location funcall sendto( ... ) returns $val if $val > -1
    {
    assert send_msg(); /* event */
    assert msg_sent(); /* condition */
    };

location funcall recvfrom( ... ) returns $val if $val > 0
    {
```

```
    assert rcv_ack();  /* event */
    assert ack_rcvd(); /* condition */
    };

location assign stuff.type returns 501  /* 501 == DONE */
    {
    assert msg_done();
    };

/* processes must end with sec_tuple(DONE, send_msg, msg_sent ) */

/* 501 == DONE */
location assign state returns 501  {
    assert state_assign(DONE);
    };

state_assign(DONE) and sec_tuple(READY, flip_bit, corr_mnum) and
      send_msg() and msg_sent() and msg_done()
    {
    retract state_assign(DONE);
    retract sec_tuple(READY, flip_bit, corr_mnum);
    assert sec_tuple(DONE, send_msg, msg_sent);
    retract send_msg();
    retract msg_sent();
    retract msg_done();
    };

/* 5 == MSENT */
location assign state returns 5  {
    assert state_assign(MSENT);
    };
```

```
state_assign(MSENT) and sec_tuple(READY, flip_bit, corr_mnum)
      and send_msg() and msg_sent()
    {
    retract state_assign(MSENT);
    retract sec_tuple(READY, flip_bit, corr_mnum);
    assert sec_tuple(MSENT, send_msg, msg_sent);
    retract send_msg();
    retract msg_sent();
    };

/* corr_mnum(FALSE) is an additional condition here, maybe it is
   necessary to store this information in the sec_tuple(???). it
   tells how the process reached this state.  adding this information
   would probably entail adding a new state or at least a new
   sec_tuple for the MSENT state.  currently, this method pushes
   states together rather than breaking them up into distinct states
   like in the rcv.tas specifications.
*/
state_assign(MSENT) and sec_tuple(ACKRCVD, rcv_ack, ack_rcvd)
      and send_msg() and msg_sent() and corr_mnum(0)
    {
    retract state_assign(MSENT);
    retract sec_tuple(ACKRCVD, rcv_ack, ack_rcvd);
    assert sec_tuple(MSENT, send_msg, msg_sent);
    retract send_msg();
    retract msg_sent();
    retract corr_mnum(0);
    };

/* 6 == ACKRCVD */
location assign state returns 6  {
    assert state_assign(ACKRCVD);
```

```
    };

state_assign(ACKRCVD) and sec_tuple(MSENT, send_msg, msg_sent)
      and rcv_ack() and ack_rcvd()
    {
    retract state_assign(ACKRCVD);
    retract sec_tuple(MSENT, send_msg, msg_sent);
    assert sec_tuple(ACKRCVD, rcv_ack, ack_rcvd);
    retract rcv_ack();
    retract ack_rcvd();
    };

location assign correct_msgnum returns $val
    {
    corr_mnum($x) { retract corr_mnum($x); };
    assert corr_mnum($val);
    };

location assign s_seq_num returns $val
    {
    assert flip_bit();
    };
```

.3 Testing the Implementation

In testing the implementation against the specifications, we seeded flaws into the implementation.

We added the below code just after incrementing count in the sender's implementation.

```
if ( count == 6 ) {   /* sort of like the ack did not get back to sender */
    printf("Count is 6 \n");
    stuff.type = 0;
```

```
}
```

This caused the sender to act like it lost a message even though it did not. Running this code through the Tester's Assistant produced the following results.

```
?- eventually clause fact([sec_tuple,DONE,send_msg,msg_sent]) failed.
```

This told the tester that the program was stuck in an infinite loop because it stated that the program never reached its done state.

When the sender side was seeded with the below flaw it looked like it worked correctly when it did not. The flaw was to change the program such that the sender never left the READY state. The code below was contains the change from the original program.

```
state = READY;
s_seq_num = (s_seq_num + 1 ) % 2;
/* state = MSNT; */
break;
      case MSNT:
```

The specifications caught this problem, indicating that the process skipped the ACKRCVD state by stating that the invariant that states that the READY state_assign implies the ACKRCVD sec_tuple exists failed. A portion of the exact message given by the Testers' Assistant is:

```
?- sndn.tas, specification 3: invariant failed.

 READY = READY
 BEGIN = BEGIN
 ACKRCVD = ACKRCVD
 rcv_ack = rcv_ack
 ack_rcvd = ack_rcvd
yes
```

¿From the preprocessed specification file, we know that specification 3 is the one that deals with making sure the states progress from the READY to the MSENT state appropriately.

AppendixThe File Transfer Protocol

This appendix contains specifications for ftp. They were written by examination of RFC 959 and other descriptions of ftp. The source code referenced by the location specifiers in the specifications for the ftp client were obtained from ftp://ftp.linux.org/pub/network/telnet. The specifications were processed and the code evaluated by means of the procedure described in the last appendix. The second section of this appendix contains the fragments of the original ftp client source code which were seeded with flaws and the results of testing the specifications against the flawed implementation by means of the Testers' Assistant.

.4 The Specifications

```
/* File : ftp.tas
   File of specifications for the client side of the ftp protocol
   sec_tpl is a tuple containing the current state, event to
   reach that state, and the condition to reach that state.

   Note: fake values are asserted in the first CLOSE state after the
        BEGIN state.  This is to make the sec_tpl asserts and the state
        assignments fit together properly.
*/

/************************************************************************
   This section contains the invariants for the specifications
************************************************************************/

/* invariant states that only one sec_tpl can exist at a time for
   the connection */

sec_tpl($s1, $e1, $c1) and sec_tpl($s2, $e2, $c2) implies
   ($s1 = $s2 and $e1 = $e2 and $c1 = $c2 );

/* these temporal invariants help prevent jumping states */
```

```
sec_tpl(FTP_UNCONNECTED, $ev, $co) until
   (sec_tpl(FTP_CMD_QUIT, $ev, $co) or
    sec_tpl(FTP_CMD_DISCONNECT, $ev, $co) or
    sec_tpl(FTP_CMDLINE_OPEN, $ev, $co) or
    sec_tpl(FTP_CMD_OPEN, $ev, $co)
   );

sec_tpl(FTP_CMD_QUIT, $ev, $co) until sec_tpl(FTP_QUIT, $ev, $co);

sec_tpl(FTP_CMD_DISCONNECT, $ev, $co) until
    sec_tpl(FTP_UNCONNECTED, $ev, co);

sec_tpl(FTP_CMDLINE_OPEN, $ev, $co) until
   (sec_tpl(FTP_UNCONNECTED, $ev, $co) or
    sec_tpl(FTP_CONNECTED, $ev, $co) or
    sec_tpl(FTP_CMDLINE_LOGIN, $ev, $co)
   );

/*********************************************************/
/**** Catch the events and conditions in the source code **/

location assign connected returns 1
    {
    retract disconnected();
    assert connected();
    };

location assign connected returns 0
    {
    retract connected();
    assert disconnected();
    };
```

```
location assign code returns $x
    {
    code($y) { retract code ($y); } ;
    assert code($x);
    };

location assign autologin returns 1
    {
    assert autologin();
    };

location funcall setpeer( ... )
    {
    assert setpeer();
    };

location funcall getreply( ... ) returns $val
    {
    reply($y) { retract reply($y); };
    assert reply($val);
    };

location funcall dologin( ... ) returns $val
    {
    dologin($x) { retract dologin($x); };
    assert dologin($val);
    };

location funcall socket( ... ) returns $val
    {
    sock_call($x) { retract sock_call($x);};
```

```
    assert sock_call($val);
    };

location funcall connect( ... )
    {
    assert conn_call();
    };

location funcall vfprintf($a, $b, $c)
    {
    command($w, $x, $y, $z) { retract command($w, $x, $y, $z); };
    assert command($b[0], $b[1], $b[2], $b[3]);
    };

location funcall getcmd( ... ) returns $c
    {
    assert cmd_call($c->c_name);
    cmd_call($y) { retract cmd_call($y); };
    };


/*****************************************/
/*****************************************/

sec_tpl(FTP_CMD_OPEN, $ev, $co) until
   (sec_tpl(FTP_UNCONNECTED, $ev, $co) or
    sec_tpl(FTP_CONNECTED, $ev, $co) or
    sec_tpl(FTP_CMD_AUTOLOGIN)
   );

sec_tpl(FTP_CMDLINE_LOGIN, $ev, $co) until
   sec_tpl(FTP_CONNECTED, $ev, $co);
```

```
sec_tpl(FTP_CMD_AUTOLOGIN, $ev, $co) until
   sec_tpl(FTP_CONNECTED, $ev, $co);

sec_tpl(FTP_CONNECTED, $ev, $co) until
   (sec_tpl(FTP_CMD_USER, $ev, $co) or
    sec_tpl(FTP_CMD_QUIT, $ev, $co) or
    sec_tpl(FTP_CMD_DISCONNECT, $ev, $co) or
    sec_tpl(FTP_CMD_PUT, $ev, $co) or
    sec_tpl(FTP_CMD_APPEND, $ev, $co) or
    sec_tpl(FTP_CMD_GET, $ev, $co) or
    sec_tpl(FTP_CMD_REGET, $ev, $co)
   );

sec_tpl(FTP_CMD_USER, $ev, $co) until sec_tpl(FTP_CONNECTED, $ev, $co);


/* This invariant says process must end in QUIT state */
/* this state can only occur once (e.g. when a bye, quit or exit call is
   made).  So, when it occurs, we know the process has finished
   gracefully.
*/

eventually sec_tpl(FTP_QUIT, quitsent, disconnected);


/*******************************************************************
 These are the main predicates and logic.  Many of these are source
  code specific.
********************************************************************/

/* created for ftp, since the source code does not assign states */
```

```
/*
FTP_UNCONNECTED    1
FTP_CONNECTED      2
FTP_QUIT    3  (last state, asserted when exiting ftp)
FTP_CMD_QUIT       4
FTP_CMD_DISCONNECT 5
FTP_CMDLINE_OPEN   6
FTP_CMDLINE_LOGIN  7
FTP_CMD_OPEN    8
FTP_CMD_AUTOLOGIN  9
FTP_CMD_USER   10
*/

/****** Processing of the caught events  *****/
/****** so they can be used by the specs *****/

/* This initializes the process of stepping through the states */
/* When ftp is called, state is set to FTP_UNCONNECTED */
/* and disconnected is asserted.  We would like this to occur on
   declaration of the global variable disconnected, but that feature does
   not work yet in TASpec
*/
assert sec_tpl(FTP_UNCONNECTED, 0, 0);
assert disconnected();

 /* below spec prevents losing the fact that some command did not
    execute properly.  A command can only be asserted in
    TASpec if in the UNCONNECTED or CONNECTED state, otherwise
    there is an error or some other command has not finished yet
 */
cmd_call($y) and sec_tpl(FTP_UNCONNECTED, $ev, $co)
```

```
    {
    retract cmd_call($y);
    assert cmd($y);
    };

cmd_call($y) and sec_tpl(FTP_CONNECTED, $ev, $co)
    {
    retract cmd_call($y);
    assert cmd($y);
    };

cmd("exit")
    {
    retract cmd("exit");
    assert cmd("quit");
    };

cmd("bye")
    {
    retract cmd("bye");
    assert cmd("quit");
    };

cmd("close")
    {
    retract cmd("close");
    assert cmd("disconnect");
    };

cmd("image")
    {
    retract cmd("image");
```

```
    assert cmd("binary");
    };

cmd("dir")
    {
    retract cmd("dir");
    assert cmd("ls");
    };

cmd("recv")
    {
    retract cmd("recv");
    assert cmd("get");
    };

cmd("send")
    {
    retract cmd("send");
    assert cmd("put");
    };


/******************************************************************/
/* Put more state transitions here.
******************************************************************/

/*--- open the connection and login (or not) ---*/

sec_tpl(FTP_UNCONNECTED, $ev, $co) and sock_call($x)
   and not cmd("open")
    {
    $x > -1
```

```
{
retract sec_tpl(FTP_UNCONNECTED, $ev, $co);
retract sock_call($x);
assert sec_tpl(FTP_CMDLINE_OPEN, sock_call, not_cmd_open);
  };
    retract sock_call($x);
    };

sec_tpl(FTP_UNCONNECTED, $ev, $co) and sock_call($x)
   and cmd("open")
    {
    $x > -1
{
retract sec_tpl(FTP_UNCONNECTED, $ev, $co);
assert sec_tpl(FTP_CMD_OPEN, sock_call, cmd_open);
};
    retract sock_call($x);
    retract cmd("open");
    };

sec_tpl(FTP_CMDLINE_OPEN, $ev, $co) and conn_call() and connected()
   and autologin()
    {
    retract sec_tpl(FTP_CMDLINE_OPEN, $ev, $co);
    retract conn_call();
    assert sec_tpl(FTP_CMDLINE_LOGIN, cmdline_connected, autologin);
    };

sec_tpl(FTP_CMDLINE_OPEN, $ev, $co) and conn_call() and connected()
   and not autologin()
    {
    retract sec_tpl(FTP_CMDLINE_OPEN, $ev, $co);
```

```
    retract conn_call();
    assert sec_tpl(FTP_CONNECTED, cmdline_connected, no_login);
    };

sec_tpl(FTP_CMD_OPEN, $ev, $co) and conn_call() and connected()
   and autologin()
    {
    retract sec_tpl(FTP_CMD_OPEN, $ev, $co);
    retract conn_call();
    assert sec_tpl(FTP_CMD_AUTOLOGIN, cmd_connected, autologin);
    };

sec_tpl(FTP_CMD_OPEN, $ev, $co) and conn_call() and connected()
   and not autologin()
    {
    retract sec_tpl(FTP_CMD_OPEN, $ev, $co);
    retract conn_call();
    assert sec_tpl(FTP_CONNECTED, cmd_connected, nologin);
    };

sec_tpl(FTP_CMDLINE_LOGIN, $ev, $co) and command('U','S','E','R')
   and connected()
    {
    retract sec_tpl(FTP_CMDLINE_LOGIN, $ev, $co);
    retract command('U','S','E','R');
    assert sec_tpl(FTP_CONNECTED, cmdline_connected, login);
    };

sec_tpl(FTP_CMD_AUTOLOGIN, $ev, $co) and command('U','S','E','R')
   and connected()
    {
    retract sec_tpl(FTP_CMD_AUTOLOGIN, $ev, $co);
```

```
    retract command('U','S','E','R');
    assert sec_tpl(FTP_CONNECTED, cmd_connected, autologin);
    };

/*--- command quit ---*/

(sec_tpl(FTP_UNCONNECTED, $ev, $co) or sec_tpl(FTP_CONNECTED, $ev, $co))
   and cmd("quit")
    {
    sec_tpl($st, $ev, $co) { retract sec_tpl($st, $ev, $co); };
    retract command('Q','U','I','T');
    retract cmd("quit");
    assert sec_tpl(FTP_CMD_QUIT, quit, eitherstate);
    };

sec_tpl(FTP_CMD_QUIT, $ev, $co) and command('Q','U','I','T')
   and disconnected()
    {
    retract sec_tpl(FTP_CMD_QUIT, $ev, $co);
    retract command('Q','U','I','T');
    assert sec_tpl(FTP_QUIT, quitsent, disconnected);
    };

/*--- command disconnect ---*/

sec_tpl(FTP_CONNECTED, $ev, $co) and cmd("disconnect")
    {
    retract sec_tpl(FTP_CONNECTED, $ev, $co);
    retract cmd("disconnect");
    assert sec_tpl(FTP_CMD_DISCONNECT, disconnect, connected);
    };
```

```
sec_tpl(FTP_CMD_DISCONNECT, $ev, $co) and command('Q','U','I','T')
   and disconnected()
    {
    retract sec_tpl(FTP_CMD_DISCONNECT, $ev, $co);
    retract command('Q','U','I','T');
    assert sec_tpl(FTP_UNCONNECTED, quitsent, disconnected);
    };

/*--- command user ---*/

sec_tpl(FTP_CONNECTED, $ev, $co) and cmd("user")
    {
    retract sec_tpl(FTP_CONNECTED, $ev, $co);
    retract cmd("user");
    assert sec_tpl(FTP_CMD_USER, user, connected);
    };

sec_tpl(FTP_CMD_USER, $ev, $co) and reply($x)
   and command('U','S','E','R')
    {
    retract sec_tpl(FTP_CMD_USER, $ev, $co);
    retract reply($x);
    retract command('U','S','E','R');
    assert sec_tpl(FTP_CONNECTED, reply, command_USER);
    };

   /*** Note that in this particular implementation ***/
   /*** of ftp, only sendrequest, recvrequest, and ***/
   /*** ruserpass call fopen.  This might not work ***/
   /*** if other functions called fopen.  If TASpec ***/
   /*** had the ability to specify particular function ***/
   /*** calls, this might not be a problem.  For now,  ***/
```

```
   /*** ignore ruserpass's call to fopen.    ***/

location funcall fopen($local, $mode)
    {
    file($x) { retract file($x); };
    assert file($local);
    };

/*--- command put ---*/

   /*** I am ignoring put commands when the local "file" ***/
   /*** is a pipe or from stdin.  These commands do not ***/
   /*** call fopen, so I can not catch  them right now ***/
   /*** The specifications will return an error if one of ***/
   /*** these calls is made.  It is possible to catch them ***/
   /*** but for now I am not going to do so.  ***/

sec_tpl(FTP_CMD_PUT, $ev, $co) until sec_tpl(FTP_CONNECTED, $ev, $co);

location assign sunique returns 1
    {
    assert sunique();
    };

location assign sunique returns 0
    {
    retract sunique();
    };

location funcall sendrequest($cmd, $local, $remote, $p)
    {
    put_local($x) { retract put_local($x); };
```

```
    assert put_local($local);
    command($w, $x, $y, $z) { retract command($w, $x, $y, $z); };
    assert command($cmd[0], $cmd[1], $cmd[2], $cmd[3]);
    };

sec_tpl(FTP_CONNECTED, $ev, $co) and cmd("put")
    {
    retract sec_tpl(FTP_CONNECTED, $ev, $co);
    retract cmd("put");
    assert sec_tpl(FTP_CMD_PUT, user, connected);
    };

sec_tpl(FTP_CMD_PUT, $ev, $co) and reply($x)
   and command('S','T','O','R') and not sununique()
   and file($f) and put_local($l)
   {
   ( $f = $l )
    {
    retract sec_tpl(FTP_CMD_PUT, $ev, $co);
    retract reply($x);
    retract command('S','T','O','R');
    retract file($f);
    retract put_local($l);
    assert sec_tpl(FTP_CONNECTED, reply, command_PUT);
    };
   };

sec_tpl(FTP_CMD_PUT, $ev, $co) and reply($x)
   and command('S','T','O','U') and sununique()
   and file($f) and put_local($l)
   {
   ( $f = $l )
```

```
    {
    retract sec_tpl(FTP_CMD_PUT, $ev, $co);
    retract reply($x);
    retract command('S','T','O','U');
    retract file($f);
    retract put_local($l);
    assert sec_tpl(FTP_CONNECTED, file_ok, command_PUT_unique);
    };
   };

/*--- command append ---*/

sec_tpl(FTP_CMD_APPEND, $ev, $co) until sec_tpl(FTP_CONNECTED, $ev, $co);

sec_tpl(FTP_CONNECTED, $ev, $co) and cmd("append")
    {
    retract sec_tpl(FTP_CONNECTED, $ev, $co);
    retract cmd("append");
    assert sec_tpl(FTP_CMD_APPEND, append, connected);
    };

sec_tpl(FTP_CMD_APPEND, $ev, $co) and reply($x)
   and command('A','P','P','E')
   and file($f) and put_local($l)
   {
   ( $f = $l )
    {
    retract sec_tpl(FTP_CMD_APPEND, $ev, $co);
    retract reply($x);
    retract command('A','P','P','E');
    retract file($f);
    retract put_local($l);
```

```
    assert sec_tpl(FTP_CONNECTED, file_ok, command_APPEND);
    };
   };

/*--- command get ---*/

sec_tpl(FTP_CMD_GET, $ev, $co) until sec_tpl(FTP_CONNECTED, $ev, $co);

location assign runique returns 1
    {
    assert runique();
    };

location assign runique returns 0
    {
    retract runique();
    };

location funcall recvrequest($cmd, $local, $remote, $lm, $p)
    {
    get_local($x) { retract get_local($x); };
    assert get_local($local);
    command($w, $x, $y, $z) { retract command($w, $x, $y, $z); };
    assert command($cmd[0], $cmd[1], $cmd[2], $cmd[3]);
    };

sec_tpl(FTP_CONNECTED, $ev, $co) and cmd("get")
    {
    retract sec_tpl(FTP_CONNECTED, $ev, $co);
    retract cmd("get");
    assert sec_tpl(FTP_CMD_GET, get, connected);
    };
```

```
sec_tpl(FTP_CMD_GET, $ev, $co) and reply($x)
   and command('R','E','T','R') and not runique()
   and file($f) and get_local($l)
   {
   ( $f = $l )
    {
    retract sec_tpl(FTP_CMD_GET, $ev, $co);
    retract reply($x);
    retract command('R','E','T','R');
    retract file($f);
    retract get_local($l);
    assert sec_tpl(FTP_CONNECTED, file_ok, command_GET);
    };
   };

/*--- command reget ---*/
   /*** this is very similar to get, but first must ***/
   /*** check that the file exists already --- can't ***/
   /*** reget a file that doesn't already exist ***/

sec_tpl(FTP_CMD_REGET, $ev, $co) until sec_tpl(FTP_CONNECTED, $ev, $co);

location funcall stat($f, $b) returns $val
    {
    stat($x) {retract stat($x); };
    assert stat($val);
    };

sec_tpl(FTP_CONNECTED, $ev, $co) and cmd("reget")
    {
    retract sec_tpl(FTP_CONNECTED, $ev, $co);
```

```
    retract cmd("reget");
    assert sec_tpl(FTP_CMD_REGET, reget, connected);
    };

sec_tpl(FTP_CMD_REGET, $ev, $co) and reply($x)
   and command('R','E','T','R') and not runique()
   and file($f) and get_local($l) and stat(0)
   {
   ( $f = $l )
    {
    retract sec_tpl(FTP_CMD_REGET, $ev, $co);
    retract reply($x);
    retract command('R','E','T','R');
    retract file($f);
    retract get_local($l);
    retract stat(0);
    assert sec_tpl(FTP_CONNECTED, file_ok, command_REGET);
    };
   };

sec_tpl(FTP_CMD_REGET, $ev, $co) and stat( $x )
    {
    $x = -1
{
retract sec_tpl(FTP_CMD_REGET, $ev, $co);
retract stat( -1 );
assert sec_tpl(FTP_CONNECTED,file_nonexistent, command_REGET);
};
    };
```

.5 Testing the Implementation

In order to test the ftp implementation against specifications, we forced the file open commands used to send the data in files to open a specific file. As a result, when the executable is run, the ftp client sends the wrong file and the server catches this. We tested receiving a file in a similar fashion. Below are the results that the Testers' Assistant generated.

AppendixThe TCP/IP Protocol

This appendix contains partially completed specifications for the TCP/IP communications protocol. They were written though using RFC793 for TCP. They require location specifiers to complete them. They were written similarly to the ftp and alternating bit protocol specifications.

```
/* File : tcp.tas

   File of specifications for the tcp/ip protocol

   secf_tpl is a tuple containing the current state, event to
   reach that state, the condition to reach that state and the file
   descriptor of the socket in that state.  ( the file descriptor
   should be unique for every socket )

    Note: fake values are asserted in the first CLOSE state after the
        BEGIN state.  This is to make the secf_tpl asserts and the state
        assignments initialize properly.
*/

/*************************************************************************
   This section contains the invariants for the specifications
*************************************************************************/

/* This invariant states that only one state_assign can exist at a time for
```

```
   each socket */

st_asgn($s1,$f1) and st_asgn($s2, $f2) and $f1 <>$f2;

/* invariants to assure the proper progression of states */

st_asgn(TCP_ESTABLISHED,$fd) implies
   (secf_tpl(TCP_SYN_RECV,snd_SYN_ACK,rcv_SYN, $fd ) or
    secf_tpl(TCP_SYN_RECV,snd_ACK,rcv_SYN, $fd) or
    secf_tpl(TCP_SYN_SENT,snd_SYN,send, $fd) or
    secf_tpl(TCP_SYN_SENT,snd_SYN,open_active,$fd)
   );

st_asgn(TCP_SYN_SENT, $fd) implies
   (secf_tpl(TCP_CLOSE, $ev, $co, $fd) or
    secf_tpl(TCP_LISTEN, 0, open_passive, $fd)
   );

st_asgn(TCP_SYN_RECV, $fd) implies
   (secf_tpl(TCP_LISTEN, 0, open_passive, $fd) or
    secf_tpl(TCP_SYN_SENT,snd_SYN,open_active, $fd) or
    secf_tpl(TCP_SYN_SENT,snd_SYN,send, $fd)
   );

st_asgn(TCP_FIN_WAIT1, $fd) implies
   (secf_tpl(TCP_SYN_RECV,snd_SYN_ACK,rcv_SYN,$fd) or
    secf_tpl(TCP_SYN_RECV,snd_ACK,rcv_SYN, $fd) or
    secf_tpl(TCP_ESTABLISHED,0,rcv_ACKofSYN, $fd) or
    secf_tpl(TCP_ESTABLISHED,snd_ACK,rcv_SYN_ACK,$fd)
   );

st_asgn(TCP_FIN_WAIT2, $fd) implies secf_tpl(TCP_FIN_WAIT1,snd_FIN, close ,
```

```
$fd);

st_asgn(TCP_TIME_WAIT, $fd) implies
   (secf_tpl(TCP_FIN_WAIT2, 0, rcv_ACKofFIN, $fd) or
    secf_tpl(TCP_CLOSING, snd_ACK, rcv_FIN, $fd)
   );

st_asgn(TCP_CLOSE, $fd) implies
   (not secf_tple($st, $ev, $co , $fd) or
    secf_tpl(TCP_LAST_ACK,snd_FIN,close,$fd) or
    secf_tpl(TCP_TIME_WAIT,snd_ACK,rcv_FIN,$fd) or
    secf_tpl(TCP_TIME_WAIT,0,rcv_ACKofFIN,$fd) or
    secf_tpl(TCP_LISTEN, 0, open_passive, $fd) or
    secf_tpl(TCP_SYN_SENT, snd_SYN, send,$fd) or
    secf_tpl(TCP_SYN_SENT, snd_SYN, open_active,$fd)
   );

st_asgn(TCP_CLOSE_WAIT,$fd) implies
   (secf_tpl(TCP_ESTABLISHED,0,rcv_ACKofSYN, $fd) or
    secf_tpl(TCP_ESTABLISHED,snd_ACK,rcv_SYN_ACK,$fd)
   );

st_asgn(TCP_LAST_ACK, $fd) implies secf_tpl(TCP_CLOSE_WAIT,snd_ACK,rcv_FIN,
$fd);

st_asgn(TCP_LISTEN, $fd) implies secf_tpl(TCP_CLOSE, $ev, $co, $fd);

st_asgn(TCP_CLOSING, $fd) implies secf_tpl(TCP_FIN_WAIT1, snd_FIN, close, $fd);


/* These temporal invariants help prevent jumping states */
```

```
secf_tpl(TCP_ESTABLISHED, $co, $ev,$fd ) until
   (secf_tpl(TCP_FIN_WAIT1, snd_FIN, close, $fd) or
    secf_tpl(TCP_CLOSE_WAIT, snd_ACK, rcv_FIN, $fd)
   );

secf_tpl(TCP_SYN_SENT, $co, $ev, $fd ) until
   (secf_tpl(TCP_SYN_RECV, snd_ACK, rcv_SYN,$fd) or
    secf_tpl(TCP_ESTABLISHED, snd_ACK, rcv_SYN_ACK, $fd) or
    secf_tpl(TCP_CLOSE, 0, close,$fd)
   );

secf_tpl(TCP_SYN_RECV, $co, $ev, $fd ) until
   (secf_tpl(TCP_ESTABLISHED, 0, rcv_ACKofSYN, $fd) or
    secf_tpl(TCP_FIN_WAIT1, snd_FIN, close, $fd)
   );

secf_tpl(TCP_FIN_WAIT1, $co, $ev, $fd ) until
   (secf_tpl(TCP_FIN_WAIT2, 0, rcv_ACKofFIN, $fd) or
    secf_tpl(TCP_CLOSING, snd_ACK, rcv_FIN, $fd)
   );

secf_tpl(TCP_FIN_WAIT2, $co, $ev, $fd ) until
   (secf_tpl(TCP_TIME_WAIT, snd_ACK, rcv_FIN, $fd)
   );

secf_tpl(TCP_TIME_WAIT, $co, $ev, $fd ) until
   (secf_tpl(TCP_CLOSE, 0, timeout_2MSL, $fd)
   );

secf_tpl(TCP_CLOSE, $co, $ev, $fd ) until
   (secf_tpl(TCP_LISTEN, 0, open_passive, $fd) or
    secf_tpl(TCP_SYN_SENT, snd_SYN, open_active, $fd) or
```

```
    secf_tpl(TCP_DONE, 0, 0, $fd )
   );

secf_tpl(TCP_CLOSE_WAIT, $co, $ev, $fd ) until
   (secf_tpl(TCP_LAST_ACK, snd_FIN, close, $fd)
   );

secf_tpl(TCP_LAST_ACK, $co, $ev, $fd ) until
   (secf_tpl(TCP_CLOSE, 0, rcv_ACKofFIN, $fd)
   );

secf_tpl(TCP_LISTEN, $co, $ev, $fd ) until
   (secf_tpl(TCP_SYN_RECV, snd_SYN_ACK, rcv_SYN, $fd) or
    secf_tpl(TCP_SYN_SENT, snd_SYN, send, $fd) or
    secf_tpl(TCP_CLOSE, 0, close, $fd)
   );

secf_tpl(TCP_CLOSING, $co, $ev, $fd ) until
   (secf_tpl(TCP_TIME_WAIT, 0, rcv_ACKofFIN, $fd)
   );


/* This invariant states the process must end in CLOSED state. */

eventually secf_tpl(TCP_DONE, 0, 0, $fd);

/******************************************************************************
 These are the main predicates and logic.
******************************************************************************/

/* When the socket is created, state is assigned the value TCP_CLOSE.
*/
```

```
/************************************************************/
/****** Catch the state assignments ******/

/* $sock is of type "struct sock *"  */
/* should ->file be used for the file descriptor or is ->socket enough?
   This just needs a value unique to every socket, so the program can
   identify it.  Maybe this code should be creating the unique identifiers
   each time a socket is created. ( and also monitoring the identifiers )
*/

/*  The below is just another idea for monitoring unique socket identifiers */
/*  secf_tpl($st,$ev,$co,$fd) and
    current->files->fd[$fd] = $sock->socket->file */

location assign $sock->state returns $val  {
    $sock->socket <> 0 {
$val = 1 { assert st_asgn(TCP_ESTABLISHED, $sock->socket->file);};
        $val = 2 { assert st_asgn(TCP_SYN_SENT, $sock->socket->file);};
        $val = 3 { assert st_asgn(TCP_SYN_RECV, $sock->socket->file);};
        $val = 4 { assert st_asgn(TCP_FIN_WAIT1, $sock->socket->file);};
        $val = 5 { assert st_asgn(TCP_FIN_WAIT2, $sock->socket->file);};
        $val = 6 { assert st_asgn(TCP_TIME_WAIT, $sock->socket->file);};
        $val = 7 { assert st_asgn(TCP_CLOSE, $sock->socket->file);};
        $val = 8 { assert st_asgn(TCP_CLOSE_WAIT, $sock->socket->file);};
        $val = 9 { assert st_asgn(TCP_LAST_ACK, $sock->socket->file);};
        $val =10 { assert st_asgn(TCP_LISTEN, $sock->socket->file);};
        $val =11 { assert st_asgn(TCP_CLOSING, $sock->socket->file);};
    };
};
```

```
/****** Catch the events and conditions in the source code ******/


/****** Allow the state progressions ******/

/**** state TCP_ESTABLISHED ****/
st_asgn(TCP_ESTABLISHED, $fd) and secf_tpl(TCP_SYN_RECV,$ev,rcv_SYN,$fd) and
  rcv_ACKofSYN($fd) {
    retract st_asgn(TCP_ESTABLISHED, $fd);
    retract secf_tpl(TCP_SYN_RECV,$ev,rcv_SYN,$fd);
    retract rcv_ACKofSYN($fd);
    assert secf_tpl(TCP_ESTABLISHED, 0, rcv_ACKofSYN, $fd);
};

st_asgn(TCP_ESTABLISHED, $fd) and secf_tpl(TCP_SYN_SENT,snd_SYN,$co,$fd) and
  snd_ACK($fd) and rcv_SYN_ACK($fd) {
    retract st_asgn(TCP_ESTABLISHED, $fd);
    retract secf_tpl(TCP_SYN_SENT,snd_SYN,$co,$fd);
    retract snd_ACK($fd);
    retract rcv_SYN_ACK($fd);
    assert secf_tpl(TCP_ESTABLISHED, snd_ACK, rcv_SYN_ACK, $fd);
};


/**** state TCP_SYN_SENT ****/
st_asgn(TCP_SYN_SENT, $fd) and secf_tpl(TCP_CLOSE, $ev, $co, $fd) and
  snd_SYN($fd) and open_active($fd) {
    retract st_asgn(TCP_SYN_SENT, $fd);
    retract secf_tpl(TCP_CLOSE,$ev,$co,$fd);
    retract snd_SYN($fd);
    retract open_active($fd);
    assert secf_tpl(TCP_SYN_SENT, snd_SYN, open_active, $fd);
```

```
};

st_asgn(TCP_SYN_SENT, $fd) and secf_tpl(TCP_LISTEN, $ev, $co, $fd) and
  snd_SYN($fd) and send($fd) {
    retract st_asgn(TCP_SYN_SENT, $fd);
    retract secf_tpl(TCP_LISTEN,$ev,$co,$fd);
    retract snd_SYN($fd);
    retract send($fd);
    assert secf_tpl(TCP_SYN_SENT, snd_SYN, send, $fd);
};


/**** state TCP_SYN_RECV ****/
st_asgn(TCP_SYN_RECV, $fd) and secf_tpl(TCP_LISTEN, $ev, $co, $fd) and
  snd_SYN_ACK($fd) and rcv_SYN($fd) {
    retract st_asgn(TCP_SYN_RECV, $fd);
    retract secf_tpl(TCP_LISTEN,$ev,$co,$fd);
    retract snd_SYN_ACK($fd);
    retract rcv_SYN($fd);
    assert secf_tpl(TCP_SYN_RECV, snd_SYN_ACK, rcv_SYN, $fd);
};

st_asgn(TCP_SYN_RECV, $fd) and secf_tpl(TCP_SYN_SENT, $ev, $co, $fd) and
  snd_SYN($fd) and rcv_SYN($fd) {
    retract st_asgn(TCP_SYN_RECV, $fd);
    retract secf_tpl(TCP_SYN_SENT,$ev,$co,$fd);
    retract snd_SYN($fd);
    retract rcv_SYN($fd);
    assert secf_tpl(TCP_SYN_RECV, snd_SYN, rcv_SYN, $fd);
};
```

```
/**** state TCP_FIN_WAIT1 ****/
st_asgn(TCP_FIN_WAIT1, $fd) and secf_tpl(TCP_SYN_RECV, $ev, $co, $fd) and
  snd_FIN($fd) and close($fd) {
    retract st_asgn(TCP_FIN_WAIT1, $fd);
    retract secf_tpl(TCP_SYN_RECV,$ev,$co,$fd);
    retract snd_FIN($fd);
    retract close($fd);
    assert secf_tpl(TCP_FIN_WAIT1, snd_FIN, close, $fd);
};

st_asgn(TCP_FIN_WAIT1, $fd) and secf_tpl(TCP_ESTABLISHED, $ev, $co, $fd) and
  snd_FIN($fd) and close($fd) {
    retract st_asgn(TCP_FIN_WAIT1, $fd);
    retract secf_tpl(TCP_ESTABLISHED,$ev,$co,$fd);
    retract snd_FIN($fd);
    retract close($fd);
    assert secf_tpl(TCP_FIN_WAIT1, snd_FIN, close, $fd);
};


/**** state TCP_FIN_WAIT2 ****/

st_asgn(TCP_FIN_WAIT2, $fd) and secf_tpl(TCP_FIN_WAIT1, $ev, $co, $fd) and
  rcv_ACKofFIN($fd) {
    retract st_asgn(TCP_FIN_WAIT2, $fd);
    retract secf_tpl(TCP_FIN_WAIT1,$ev,$co,$fd);
    retract rcv_ACKofFIN($fd);
    assert secf_tpl(TCP_FIN_WAIT2, 0, rcv_ACKofFIN,$fd);
};


/**** state TCP_TIME_WAIT ****/
```

```
st_asgn(TCP_TIME_WAIT, $fd) and secf_tpl(TCP_FIN_WAIT2, $ev, $co, $fd) and
   snd_ACK($fd) and rcv_FIN($fd) {
    retract st_asgn(TCP_TIME_WAIT, $fd);
    retract secf_tpl(TCP_FIN_WAIT2,$ev,$co,$fd);
    retract snd_ACK($fd);
    retract rcv_FIN($fd);
    assert secf_tpl(TCP_TIME_WAIT, snd_ACK, rcv_FIN,$fd);
};

st_asgn(TCP_TIME_WAIT, $fd) and secf_tpl(TCP_CLOSING, $ev, $co, $fd) and
   rcv_ACKofFIN($fd) {
    retract st_asgn(TCP_TIME_WAIT, $fd);
    retract secf_tpl(TCP_CLOSING,$ev,$co,$fd);
    retract rcv_ACKofFIN($fd);
    assert secf_tpl(TCP_TIME_WAIT, 0, rcv_ACKofFIN,$fd);
};

/* This starts up the process of stepping through the states. */
/**** state TCP_CLOSE ****/

st_asgn(TCP_CLOSE, $fd) and not secf_tpl($st,$ev,$co,$fd) {
    retract st_asgn(TCP_CLOSE, $fd);
    assert secf_tpl(TCP_CLOSE, 0, 0, $fd);
};

st_asgn(TCP_CLOSE, $fd) and secf_tpl(TCP_LAST_ACK,snd_FIN,close,$fd) and
  rcv_ACKofFIN($fd) {
    retract st_asgn(TCP_CLOSE, $fd);
    retract secf_tpl(TCP_LAST_ACK,snd_FIN,close,$fd);
    retract rcv_ACKofFIN($fd);
    assert secf_tpl(TCP_CLOSE, 0, rcv_ACKofFIN, $fd);
};
```

```
st_asgn(TCP_CLOSE, $fd) and secf_tpl(TCP_TIME_WAIT,$ev,$co,$fd) and
  timeout(MSL_2, $fd) {
    retract st_asgn(TCP_CLOSE, $fd);
    retract secf_tpl(TCP_TIME_WAIT,$ev,$co,$fd);
    retract timeout(MSL_2, $fd);
    assert secf_tpl(TCP_CLOSE, 0, timeout_2MSL, $fd);
};

st_asgn(TCP_CLOSE, $fd) and secf_tpl(TCP_LISTEN, 0, open_passive, $fd) and
  close($fd) {
    retract st_asgn(TCP_CLOSE, $fd);
    retract secf_tpl(TCP_LISTEN, 0, open_passive,$fd);
    retract close($fd);
    assert secf_tpl(TCP_CLOSE, 0, close, $fd);
};

st_asgn(TCP_CLOSE, $fd) and secf_tpl(TCP_SYN_SENT, snd_SYN, $co,$fd) and
  close($fd) {
    retract st_asgn(TCP_CLOSE, $fd);
    retract secf_tpl(TCP_SYN_SENT, snd_SYN, $co,$fd);
    retract close($fd);
    assert secf_tpl(TCP_CLOSE, 0, close, $fd);
};


/**** state TCP_CLOSE_WAIT ****/
st_asgn(TCP_CLOSE_WAIT, $fd) and secf_tpl(TCP_ESTABLISHED, $ev, $co, $fd) and
  snd_ACK($fd) and rcv_FIN($fd) {
    retract st_asgn(TCP_CLOSE_WAIT, $fd);
    retract secf_tpl(TCP_ESTABLISHED, $ev, $co,$fd);
    retract snd_ACK($fd);
```

```
    retract rcv_FIN($fd);
    assert secf_tpl(TCP_CLOSE_WAIT, snd_ACK, rcv_FIN, $fd);
};


/**** state TCP_LAST_ACK ****/
st_asgn(TCP_LAST_ACK, $fd) and secf_tpl(TCP_CLOSE_WAIT, $ev, $co, $fd) and
  snd_FIN($fd) and close($fd) {
    retract st_asgn(TCP_LAST_ACK, $fd);
    retract secf_tpl(TCP_CLOSE_WAIT, $ev, $co,$fd);
    retract snd_FIN($fd);
    retract close($fd);
    assert secf_tpl(TCP_LAST_ACK, snd_FIN, close, $fd);
};


/**** state TCP_LISTEN ****/
st_asgn(TCP_LISTEN, $fd) and secf_tpl(TCP_CLOSE, $ev, $co, $fd) and
  open_passive($fd) {
    retract st_asgn(TCP_LISTEN, $fd);
    retract secf_tpl(TCP_CLOSE, $ev, $co,$fd);
    retract open_passive($fd);
    assert secf_tpl(TCP_LISTEN, 0, open_passive, $fd);
};


/**** state TCP_CLOSING ****/
st_asgn(TCP_CLOSING, $fd) and secf_tpl(TCP_FIN_WAIT1, $ev, $co, $fd) and
  snd_ACK($fd) and rcv_FIN($fd) {
    retract st_asgn(TCP_CLOSING, $fd);
    retract secf_tpl(TCP_FIN_WAIT1, $ev, $co,$fd);
    retract snd_ACK($fd);
```

```
    retract rcv_FIN($fd);
    assert secf_tpl(TCP_CLOSING, snd_ACK, rcv_FIN, $fd);
};

/**** state TCP_DONE ****/
/* this is a state not in the rfc specifications.  I use it to make sure the
   program ends some time.  I can't use TCP_CLOSE to do this because the sockets
   start in state TCP_CLOSE.  So, the first time the socket enters
   state TCP_CLOSE after the initialization to state TCP_CLOSE, the
   specifications assume the program is done with the socket and consequently
   enters state TCP_DONE.   */

secf_tpl(TCP_CLOSE, 0, $co, $fd ) {
    $co <> 0 {
retract secf_tpl(TCP_CLOSE, 0, $co, $fd );
assert secf_tpl(TCP_DONE, 0, 0, $fd );
    };
};
```

AppendixUsing the Testers' Assistant

This appendix details the exact steps taken to process specifications written in TASpec and then use them to test the implementation of a protocol.

```
To use these files with the Testers' Assistant, follow these steps:
1. execute 'make rtasn' == 'taspec.exe rcvn.tas'
2. fix the rcvn.tas.pl file's ta_and statements
3. add all the externs to the receiver1.c file ( already done)
4. execute 'make rinstn' ==
    'locparser.exe -s/dev/null -loc rcvn.tas.locY
-pl rcvn.inst.c.pl receiver1.c > rcvn.inst.c'
5. add the #defines and #includes back into rcvn.inst.c from the
    file 'addlindef'
6. execute 'make rcvn' ( compiles rcvn.inst.c )
```

```
7. run the executable  ( rcvn needs sndn to run )
8. execute 'rcvnrun' ( this runs the appropriate .pl files through
    prolog which checks the execution of the run.  This produces
    output in the file 'rcvn.inst.c.pl.out' which indicates whether
    the run violated the invariants in the specifications or not.
9. examine the 'rcvn.inst.c.pl.out' file

Other stuff:
rcvn.tas.pl  -- specifications that need more parentheses:
14,15,17,21,25,27,28
sndn.tas.pl  -- specifications that need more parentheses:
12,13,18,20,21,23
```

Bibliography

[1] Joel Dodson. Specification and Classification of Generic Security Flaws for the Testers' Assistant Library. Masters' Thesis, Department of Computer Science, University of California at Davis, Davis California 95616, 1996.

[2] Biswaroop Guha. Vulnerability Analysis of the TCP/IP Suite. Masters' Thesis, Department of Computer Science, University of California at Davis, Davis California 95616, 1995.

[3] George Fink. ————-. Phd Thesis, Department of Computer Science, University of California at Davis, Davis California 95616, 1996.

[4] George Fink. ——————

[5] ¡where obtained tcp/ip from¿

[6] Z book

[7] http://ds.internic.net/rfc/rfc793.txt. RFC for TCP/IP.

[8] http://ds.internic.net/rfc/rft959.txt. RFC for the file transfer protocol.

[9] ftp://ftp.linux.org/pub/network/telnet. Source code for ftp.

[10] John V. Guttag, J.J. Horring. Larch:Languages and Tools for Formal Specification: Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[11] Steve King. Specification Case Studies: The use of Z in the Restructure of IBM CICS. Prentice Hall, 1994.

[12] David Luckham, S. Sanker, S. Takahash. Two Dimensional Pinpointing: Debugging with Formal Specifications. IEEE Software, volume 8(1):74-84, January 1991.

[13] Pamela Zave. An operational approach to requirements specification for embedded systems. IEE Transactions on Software Engineering, volume SE-8(3):250-259, May 1982.

[14] Hanene Ben-Abdalla, I. Lee. The Integrated Specification and Analysis of Functional, Temporal, and Resource Requirements. Third IEEE International Symposium on Requirements Engineering, January 6-10, 1997. IEEE Computer Society Press. page 198.

[15] Hanene Ben-Abdalla, J.Y. Choi, I. Lee. A Process Algebraic Method for the specification and Analysis of Real-Time systems. In C. Heitmeyer and D. Mandrioli, editors, Formal Methods for Real-Time Computing, Chapter 7. John Wiley and Sons, January 1996.

[16] Cliff B. Jones. Systematic Software Development using VDM. Prentice Hall, 1990.

[17] Dorothy Elizabeth Rabling Denning. Cryptography and Data Security. Addison- Wesley Publishing Company, 1983.

[18] Douglas Comer. Internetworking with TCP/IP. Prentice Hall, 1988.

[19] Eugene Spafford. Common System Vulnerabilities. Future Directions in Intrusion and Misuses Detection, 1992.

[20] S.M. Belovin. Security Problems in the TCP/IP Protocol Suite. Computer Communications Review, April 1989.