The Application of Object-Oriented Analysis to Sockets System Calls Library Testing

by

Almadena Yurevna Chtchelkanova, M.S., Ph.D.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

May 1996

The Application of Object-Oriented Analysis to Sockets System Calls Library Testing

Approved by
Supervising Committee:

_____

_____

The effort of completion of this research work was possible by the support of my loving family

Acknowledgments

I'd like to acknowledge faculty and staff members of the Department of Computer Sciences, University of Texas at Austin for providing a great environment for study and research.

I am very greatful to Professor Robert van de Geijn for his continuous support.

I am also grateful to Professor James C. Browne for his invaluable advice when supervising my research.

Almadena Yurevna Chtchelkanova

The University of Texas at Austin

May 1996

The Application of Object-Oriented Analysis to Sockets System Calls Library Testing

Almadena Yurevna Chtchelkanova, M.A.
The University of Texas at Austin, 1996

Supervisor: J.C. Browne

Object-oriented analysis (OOA) is an orderly and systematic approach for the development of software systems. Software systems developed in the OOA method are readily tested and validated. There are, however, many systems that were developed either previous to the availability of the OOA methodology or without its use. Many of these systems still exist, are frequently modified and thus must be retested after modification. There is a significant need for a capability for organized and systematic testing of existing software.

In this thesis we show how the OOA methodology can be used to develop a test suite for existing software to facilitate maintenance and modification of the existing software. The current practice in testing existing software systems is largely an ad hoc process of trial and error, ranging from random testing to exhaustive testing. Use of the OOA methodology provides an orderly and systematic process for hierarchical development of test suites, even for existing software. This concept is illustrated by development of a test suite for the Unix Sockets Library system. Sockets are abstract objects which implement interprocess communication between unrelated processes in Unix.

Test development for the socket library using the OOA methodology can be defined in four steps:

(1) Construct an information and state model for the socket system from the existing documentation.

(2) Construct a state model for a test process object based upon the state model for the socket object. Each instance of a test process object will generate events, driving a socket object through a pre- defined set of states.

(3) Derive a main driver program which will instantiate multiple copies of the test process object which will then instantiate and drive multiple instances of the socket object.

These first three steps are executed using the SES Objectbench tool to create the objects and state models and to verify their consistency and validity.

(4) Generate actual test code by coding the main test object and translating the process object models to C code or any other programming language. In this translation the events in the action language programs of the state model are replaced by appropriate system calls to the socket system. Validation checks are placed after each system call to insure that each step is properly executed.

A detailed statement of the problem, an overview of functional testing, and an overview of Berkeley Unix Sockets are given in Chapter 1. An OOA Shlaer–Mellor approach is described in Chapter 2. In Chapter 3, abstract models of a connection-based socket and a test process are constructed based on the socket system call specifications provided by the Berkeley Unix System. Implementation of the test suite is described in Chapter 4. Sample test programs, and results of testing sockets are presented in Appendix A.

Contents

List of Figures

Chapter 1

Problem Overview

1.1 Problem Statement

There is a vast body of existing software which is modified and extended on a regular basis. Testing of these modified and extended software systems is a major problem. Most of the time the testing is done on a black box basis, that is the software is invoked through its external interface using a broad spectrum of parameters. This form of testing is both uncertain as to its result and tedious in execution.

The purpose of this paper is to demonstrate that an object-oriented analysis methodology can be used to derive a covering set of tests for existing software systems. OOA provides a systematic approach to specification of the existing system at a high level of abstraction so that entire families of behavioral tests can be designed and executed. This approach is especially important when the source code of a software system is not available or the structure is complex and difficult to understand.

The requirement for deriving an appropriate test suite is knowledge of the structure and behavior of the system. It is the fuzzy understanding of structure and behavior (as well as complexity) which makes testing difficult.

OOA begins by specifying a system as a collection of objects and relationships between the objects, [12], [13], [11], and [3]. Every object is described by a set of attributes. At every moment, an object is in some well defined state which can be fully described by the values of its attributes. All possible states of an object are defined in the information model of this object at an abstract level. Associated with each state there exists an event which causes a transition of an object from this state to another (possibly the same) state. There is also associated with each state of the object an action which is executed upon arrival at the state. Events may be generated either by other objects comprising the system, or may come from external objects. Objects can be created (instantiated) and destroyed. Upon creation, each instance of an object gets and begins to execute its own private copy of its state model – a state machine.

The critical factor in the use of the OOA methodology to develop a test suite is that it provides an orderly and systematic means of defining the structure and behavior, even of an existing system. The procedure is to take the requirements statement and/or the documentation describing an existing software system and to construct an object-oriented analysis level model which reflects the structure and behavior of the system and to then use this model to define the tests. This thesis demonstrates this approach to test suite generation through a case study.

For our case study we select a BSD Unix socket as a test object. Sockets are used for interprocess communications. Sockets system calls have a very well defined and relatively simple interface. The sockets system calls library is constantly under construction. Adding new communication protocols and new socket types requires testing the full functionality of the library. A major part of the interprocess communication software (IPC) is machine independent, but there is a hardware-dependent part which must be tested when the OS is ported to a new hardware platform.

Socket system library calls represent events which cause a socket to make a

transition from one state to another, and invoke some actions associated with the transitions. To test a socket object we need another object, a test driver, to generate system calls for the socket. Our system thus will consist of two objects, a socket and a driver. To develop a model of this system we used the SES/*objectbench* tool based on the Shlaer/Mellor approach to OOA. This tool allows one to capture a model of a system, to simulate and animate interactions of model objects, and to easily analyze and validate the dynamic behavior of the system.

Our first aim is to construct an information and state model of a socket at a high level of abstraction, based on the description of the UNIX socket call library. The model needs to be expressed in terms of the SES/*objectbench* graphical and action language. Having a model of a socket object, it is then conceptually easy to construct a state model of a test driver generating an appropriate sequence of valid events navigating the socket model through a sequence of allowed states. The driver object model has to be expressed in terms of the same action language. The SES/*objectbench* is then used to debug both models and to verify that the model driver correctly steers the model socket through a pre-defined sequence of states. The benefit of using SES/*objectbench* is obvious - creation and validation of a driver is first made on a high level of abstraction, before actual software design starts. Using the SES/*objectbench* and a code generator CodeGenesis, developed for using with it, it would be possible to automate a functional test process design and to reduce the time spent on test creation.

After the model of the test is developed and validated using SES/*objectbench*, we have to translate it into a desirable target code. This can be done by hand or by using a code generator. For large models, the second approach has obvious advantages. The overhead is in writing the code generator for a target language. Once written, the code generator can be reused for subsequent models.

1.2 Functional Test Design Strategies Overview

More than 80% of the lifecycle time of a software product is spent on debugging and testing [2]. This is why proper organization and design of tests is very important. There is no single systematic way for generating test programs. Methods vary from random tests to exhaustive tests. There is no systematic way to test the testing software.

In this paper, we show how to apply Object-Oriented Analysis to the functional testing of a software system as an orderly and systematic way for generating sets of covering tests. We assume that there exists a usable specification of the software system from which an OOA model can be constructed.

Functional (black-box), or behavioral testing, checks if a software system conforms to its specification. Functional testing doesn't test the correctness of the specification of the software. It is assumed that a model is correct, and that bugs are in the software implementation of the model. Functional testing checks, if in the response of a given input, the state of the system is changed according to the specifications, and proper actions are executed after arriving at a new state. Functional testing drives the system with inputs, and all outputs are verified for conformance to specified behavior. Functional testing takes the users' point of view. We don't need to know anything about actual design and implementation of software in order to be able to test it. Behavioral testing is based on a model of software, not on the software. Verification of the model includes the testing of the control flow, the state transitions and changing of state-related attributes. The key part of the verification is that the model properly reflects the behavior embedded in the actual software.

In the case of testing the correctness of the system call library we have to distinguish

- testing of the return value of the system call. The library calls are tested for both success and failure conditions. Failure is expected when incorrect parameters are passed to a system call, or a system condition under which

the failure is expected is simulated (example - exhausting a per-process file descriptors number by opening a number of files);

- testing whether the state of the system is changed according to the specifications and proper actions are associated with each state.

We start with the second approach. When a state model for a software system is constracted based on its specifications there is a finite number of events which can be accepted in every state of the model. That means that our problem space is partitioned into small well-defined problems. By covering all possible state/event combinations, we derive exhaustive tests on each state of the model, and thus the model itself.

Description of the testing techniques can be found in [2], [6], [5], [9], [10], [4], [8], [15].

1.3 Sockets Overview

In this section a brief overview of sockets is given.

Sockets were introduced as a part of Interprocess Communication Facilities (IPC) implemented in 4.3 BSD Unix. Sockets became a part of the standard IPC in the System V and all modern flavors of Unix. Sockets allow unrelated processes to communicate regardless of whether they are running on the same host or across a network.

A socket is an abstract object from which messages are sent and received. All sockets are typed according to their communication semantics. Types are defined by the subset of properties a socket supports. These properties are:

- in-order delivery of data;
- unduplicated delivery of data;
- reliable delivery of data;

- preservation of message boundaries;
- support of out-of-band messages;
- connection-oriented communication.

Sockets are created within a communication domain much as files are created within a filesystem. Sockets exist only as long as they are referenced. A communication domain embodies the standard semantics of communication and naming.

A socket must be created with a *socket()* system call. We are going to skip the parameters for the following system calls. The type of socket is selected according to the characteristic properties required by the application. The next step depends on the type of socket being used. The most commonly used type of socket requires a connection before it can be used. Creation of a connection between two sockets requires that each socket have an address bound to it. The format of addresses can vary among domains. Socket addresses may be reused if the communication domain permits, although domains normally ensure that a socket address is unique on each host, so that the association between two sockets is unique within the communication domain. To bind an address to a socket a system call *bind()* is used. A system call, *connect()*, initiates a connection with another socket. A system call, *listen()*, marks a socket as receiving connection requests. A system call, *accept()*, creates a new socket which is connected to a socket requesting a connection, and the original socket is listening for new connections to come. The system calls listed above are used for establishing connections between two sockets.

The main use of sockets is sending and receiving data. For connected sockets, *send()* and *recv()* calls are used.

Inquiry system calls give information about socket attributes without changing a state of a the socket: *getsockname()* returns a socket address, and *getpeername()* returns a name of a peer socket (the socket on the other end of a connection). The *shutdown()* system call is used to terminate data transmission or reception at a

socket. *Getsockopt()* and *setsockopt()* are used to set and retrieve various parameters that control the operation of a socket or underlying network protocols. Sockets are discarded with the normal *close()* system call. A detailed description of socket system calls can be found in [14].

The interprocess-communication facilities are layered on the top of the networking facilities. Data flows from the application through the socket layer to the networking support, and vice versa. Sockets and network facilities are implemented within the kernel. A description of the implementation of the sockets can be found in [7].

Chapter 2

Shlaer/Mellor Approach to OOA

2.1 OOA Overview

Object-Oriented Analysis (OOA) is a method for identifying significant entities in a real-world problem and for understanding and explaining how they interact with each other [12], [13], [3], [11].

An Object-Oriented Analysis process defines

- the conceptual entities of the system as objects with semantics defined by attributes (Object Information Model);
- the relationships among the conceptual entities in terms of binary relationships or associative objects (Object Information Model);
- the behavior of the conceptual entities as a response to events (or incidents) causing state transitions and actions associated with arrival in each state (State Model);
- the interaction between conceptual entities in terms of events generated and accepted (Object Communication Model);

- the fundamental and reusable processes into which actions can be dissected.

An Object-Oriented Design process defines

- a set of templates for realizing objects as entities in a programming language;
- a set of data structures corresponding to the attributes of the object definitions;
- realizations for actions as executable entities in the form of methods;
- mechanisms for definition of control flow among the actions of the realized objects.

If OOA provides an execution environment specified separately from the application, the design representation of the model can be obtained by translation.

2.2 Identifying Objects

An Object in OOA represents a single typical but unspecified instance of a conceptual entity. Most of the objects fall into the following categories: tangible objects, roles, incidents, interactions, specifications.

Each object has a set of attributes. An attribute is an abstraction of a single characteristic possessed by all entities that were themselves abstracted as an object (a logical state variable). The range of legal values that an attribute can take is called its domain. There are three types of attributes:

- naming attributes - to establish identity of the object;
- descriptive - to provide intrinsic facts to each instance of the object;
- referential - to establish relationship.

To separate instances of the same object an identifier is used. An identifier is a set of one or more attributes whose values uniquely distinguish each instance of an object.

2.3 Identifying Relationships

A relationship is an abstraction of a set of associations that systematically hold between different kinds of things in the real world. The relation can be formalized by their multiplicity and conditionality. The three basic types of multiplicity are: one-to-one, one-to-many and many-to-many. The three types of conditionality are: unconditional, conditional and biconditional. If every instance of both objects is required to participate, the relationship has unconditional form. If there are some instances of one object that do not participate, the relationship has conditional form. If there are some instances of both objects that do not participate, the relationship has biconditional form.

The relationship has a unique identifier. To formalize a one-to-one relationship, referential attributes may be added to either object (but not both). In a one-to-many relationship, referential attributes must be added to the object on the "many" side. To formalize a many-to-many relationship, a separate associative object must be created that contains references to the identifiers of each of the participating instances. The associative object is then treated as a regular object, with a name, object description, additional attributes (if any) and may participate in relationships with other objects.

An associative object may be used to formalize any relationship, not only many-to-many relationships. A relationship with dynamic behavior must be formalized by means of an associative object.

In many problems, distinct specialized objects that have certain common attributes can be found. In this case, a more general object can be abstracted to represent common characteristics shared by the specialized objects. These objects are related through a subtype-supertype relationship.

2.4 Specification of Behavior: State Model

The abstraction of the behavioral pattern of an object includes creation and deletion of the object and changes in values assigned to attributes of object instance. Each instance of an object is always in some well defined state. A state represents a condition of the object in which a defined set of rules, policies, and physical laws applies. Transitions among these states are specified by a State Model. In the Shlaer/Mellor approach to OOA, a State Model is formalized as the Moore State machine and includes:

- a set of states;
- a set of allowed transitions between states;
- events which cause transitions among the states;
- actions which are executed when a state is entered.

An event or incident models changes in the external environment or the system resources. Event data must include the target of an event. Every state transition is initiated by one or more events.

An action models a program executed by an instance of an object upon entry to a state. One action is associated with each state. Actions include receiving event data, creating object instances, accessing object instances, generating events, and modification of object instances. Actions must be context free.

There are some rules for State Models:

A given state machine executes only one action at a time.

Multiple state machines can be simultaneously active (for different objects or different instances of the same object).

An action takes time to execute.

Actions are atomic.

Events are never lost.

Events are consumed by the execution of the receiving action.

Generated events are instantaneously available.

There exists a state machine for each object instance. A state machine is a private copy of the state model executed by an object instance. A state machine always accepts pending events as quickly as possible.

Events from a given source are received in the order generated.

Event receipt from multiple sources is nondeterministic.

There is always only one recipient for any event.

Although all objects have lifecycles, it is necessary to build state models to formalize the lifecycles for only some objects which show dynamic behavior.

To construct the State Model one must define initial state for each object, list all reachable states for the top level objects, construct state transition diagrams where each node is an assignment of values to dynamic attributes and each arc carries the event which causes a state transition. Also, one must define for each arc the methods (actions) triggered by the event which affects the change of state.

2.5 Object Communication Model

An Object Communication Model (OCM) provides a summary of event communication between state models and external entities. An OCM is a directed graph where external agents are included as sources of events, objects are nodes, and arcs carry events across objects.

The OCM is typically the top level of observation of a simulated execution. To execute a model, the initial object population must be established, the starting state of the system must be specified, and starting events generated.

During simulation, evaluation of the execution behavior can be monitored in terms of state consistency, concurrency among state model instances, proper event generation and consumption, and values of attributes which determine the path through the state model.

Chapter 3

Building a Socket Model

3.1 Information Model

In this section we describe the Information Model of our system in Fig. 3.1. We build a model for the socket which is used in connection-oriented communication. This type of socket can be used for sending and receiving information only after a connection with another socket (peer) is established.

We excluded the network communication level from our consideration by assuming that it works properly, and that an instance of a socket generates an event to another instance of a socket.

In SES/*objectbench* notation each object has an abbreviation indicated in parentheses after the object name.

There are two objects in our Model – Socket (S) and Test_Process (TP). Sockets and Test_Processes can be uniquely distinguished by their identifiers Socket_ID and Process_ID, respectively.

There is a class of sockets used only for accepting communication – Public_Socket (PS) and a class of sockets used for sending and receiving data – Regular_Socket (RS). The specialization of the Socket Object is formalized as an R2 supertype/subtype relationship. For a subtype/supertype object both a supertype

Figure 3.1: Information Model

and subtype instances must be created. In the case of a Socket Object, subtype instances are active and have a lifecycle. Supertype instances are passive and just store information. More about supertype/subtype objects can be found in [12], [1].

The attributes of a Public_Socket are:

Address – an address to which a Public Socket is bound.

Limit – the number of incoming connections allowed to be queued for processing imposed by the operating system.

N_of_connections – a number of connections queued for processing in a given time.

N_ of_requests – a number of connections accepted for processing during the Public_Socket lifetime.

Current_request – a number of an accepted request which is currently being processed.

A Regular_Socket is connected to another Regular_Socket. This relationship, R3, is formalized by adding an attribute Peer_Address.

Other attributes of a Regular_Socket Object are:

Whoami defines the role a socket plays in establishing the connection – an Active_Client or an Active_Server.

A Regular_Socket has two buffers - for sending and receiving messages, Buffer_in and Buffer_out. In our Model buffers are represented by integers for simplicity.

A Test_Process has a Role it is playing in the communication between two processes – a Client or a Server. A Client is an initiator of the communication and a Server is a recipient of a communication request.

Each process knows a Well_Known_Address – a Socket_ID of the instance of the Socket accepting connections. When referring to the address of the Socket we have in mind Socket_ID. To simplify our model, we assumed that the address space of the socket has a one-to one correspondence to the identifier space.

A Test_Process can have many Sockets. This one-to-many relationship, R1, is formalized by adding a referential attribute Process_ID to the Socket Object.

In our model, each Test_Process has one instance of Socket connected to another instance of Socket belonging to another Test Process. We describe it as a relationship R4. The attribute Active_Socket_ID of the Test_Process formalizes this relationship.

A Test_Process receives from and sends messages to another Test_Process. To simplify our model, we assume that a Message is some integer.

3.2 Public_Socket State Model

Each event in OOA indicates a recipient to which this event is addressed. In SES/*objectbench* notation an event has a form

{object abbreviation}{event number} : {event name}({event data}).

In our state model, every socket library call is represented by at least two events. The first event models the execution of a system call by a driver process. The last event models the return of the control back to the driver.

There are two subtypes of a Socket Object – a Public_Socket and a Regular_Socket (see above). A Public_Socket State Model is shown in Fig. 3.2. A Regular_Socket State Model is shown in Fig. 3.3.

An instance of a Public_Socket in the state Created(1) is created by the event *PS1: Socket_create()*. The state Created(1) is a *creation state*. A transition into a creation state is depicted as a transition from a special "dot" state. A newly created instance of a Public_Socket generates an event *TP1: Created_socket()* to the instance of the Test_Process which generated the event *PS1: Socket_create()*. These two events correspond to the actual system call *socket()*.

A system call *bind()* is represented by the two events, *PS16: Socket_bind()* and *TP2: Binded_socket()*. The event *PS16* changes the state of a Public_Socket from the state Created(1) to the state Binded(2). And the event *TP2* returns control to the Test_Process.

A Public_Socket changes its state from Binded(2) to Listeningi(3) after re-

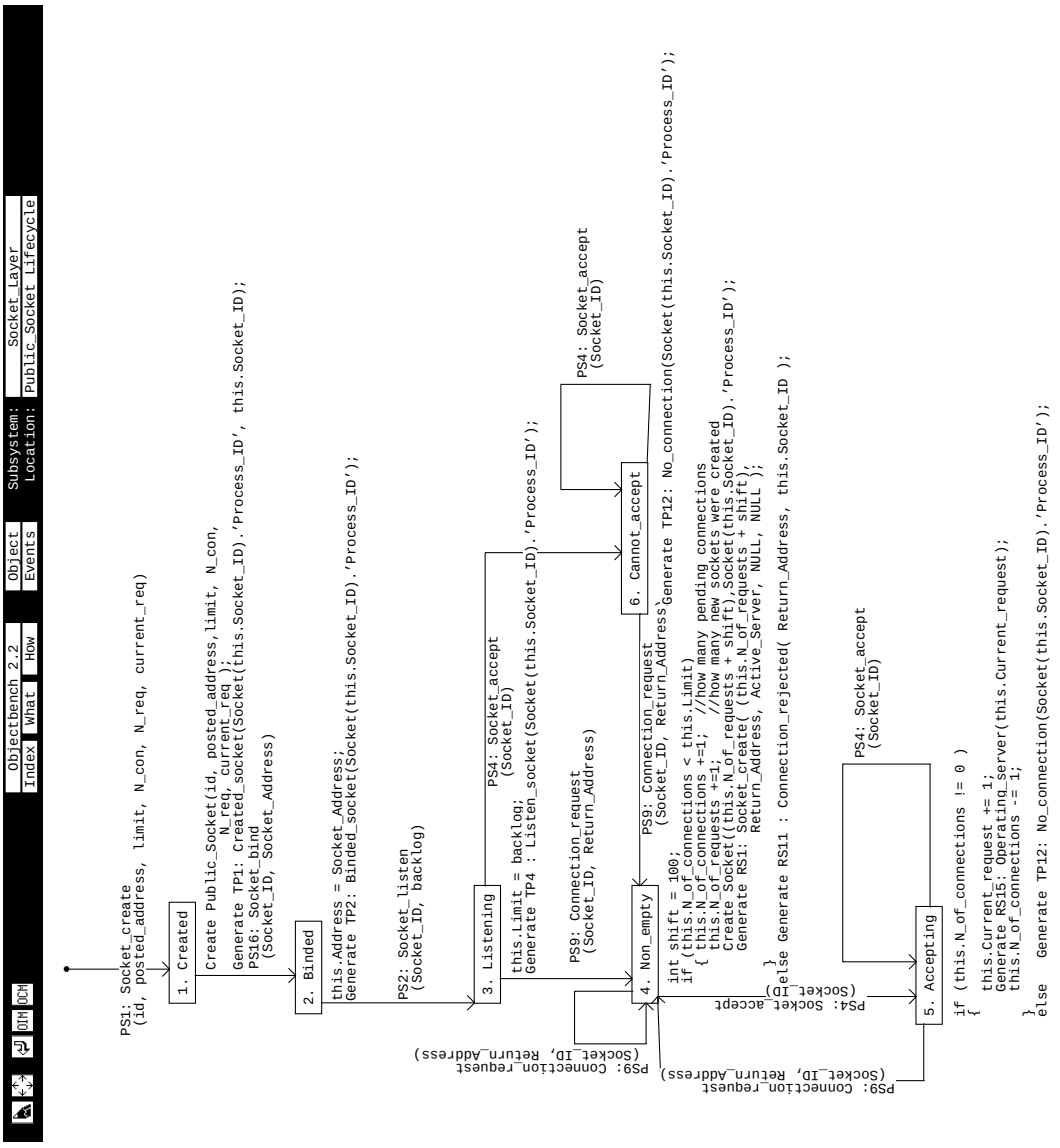

Figure 3.2: Public Socket State Model

ceiving an event *PS2: Socket_listen()*, and returns control to the Test_Process by generating an event *TP4: Listen_socket()*. These two events model *listen()* system call.

When a Public_Socket is in the state Listening(3), it can accept two events: *PS4: Socket_accept()* generated by a Test_Process and *PS9: Connection_request* generated by an instance of a Regular_Socket. The first event changes a state of a Public_Socket to Cannot_accept(6) where an event *TP12: No_connection* is generated. *SP4* and *TP12* model non-blocking *accept()* system call. It returns a negative integer if there are no pending connections. After receiving *PS9*, a Public_Socket changes state from Listening(3) to Non_empty(4). Upon arrival in the state Non_empty(4), a Public_Socket processes the connection request – creates a new Regular_Socket by generating an event *RS1* and updates the values of the attributes. The action for this state includes if-logic because the state Non_empty(4) can be reached from many states by receiving an event *PS9*. If a number of pending connections is less than a Limit a connection is processed, and is otherwise rejected by generating *RS11*.

A Public_Socket remains in the state Non_empty(4) when receiving an event *PS9*, or makes a transition to the state Accepting(5) by receiving event *PS4*. Upon arrival in this state a Public_Socket updates its attributes and transfers control to a previously created instance of a Regular_Socket by generating an event *RS15*. A system call *accept()* is modeled by a sequence of events. The return of control to the Test_Process when a system call succeeds is done by a Regular_Socket by generating an event *TP5*.

3.3 Regular_Socket State Model

To refer to an instance of an object having some defined value of an attribute the following notation is used:

{object name}({attribute name} = {attribute value}).

An instance of a Regular_Socket in the creation state Created(1) is created by an event *RS1: Socket_create()*. In this section refer to Fig. 3.3. An event *RS1* can be generated by an instance of a Test_Process (Role = Client), or by an instance of a Public_Socket. We will refer to an instance of a Test_Process (Role = Client) as a Client, and a Test_Process (Role = Server) as a Server. We will refer to an instance of a Regular_Socket (Whoami = Active_Server) as an Active_Server, and to an instance of a Regular_Socket (Whoami = Active_Client) as an Active_Client. A newly created instance of an Active_Client generates an event *TP1: Created_socket()* to the instance of the Client which generated the event *PS1*. Events *RS1* and *PS1* model the actual system call *socket()*. After receiving an event *RS3: Socket_connect()*, an instance of a Active_Client changes its state from Created(1) to Connecting(3). Upon arriving in the state Connecting(3) it generates an event *PS9: Connection_request* to a Public_Socket.

An Active_Server changes its state from Created(1) to Is_connected(2) after receiving an event *RS15: Operating_server()* from a Public_Socket. Note that Active_Server and Public_Socket belong to the same Server. An event *RS15* transfers control from a Public_Socket to an Active_Server. An Active_Server proceeds with establishing a connection with an Active_Client. Upon arriving at the state Is_connected(2), an Active_Server generates two events. An event *RS10: Connection_accepted()* is directed to an instance of an Active_Client requesting a connection. Another event *TP5: Accepted_socket()* returns control to an instance of the Server. After receiving an event *RS10*, an instance of an Active_Client changes its state from the state Connecting(3) to the state Connected(4). Upon arriving at the state Connected(4), an Active_Client generates an event *TP3: Connected_socket()* which returns control to the Client. At this point a connection between two Regular_Sockets is established and processes can send and receive messages. From Fig. 3.3 it is clear that state machines for an Active_Client and an Active_Server are identical and we can again refer to both instances as a Regular_Socket. From the state

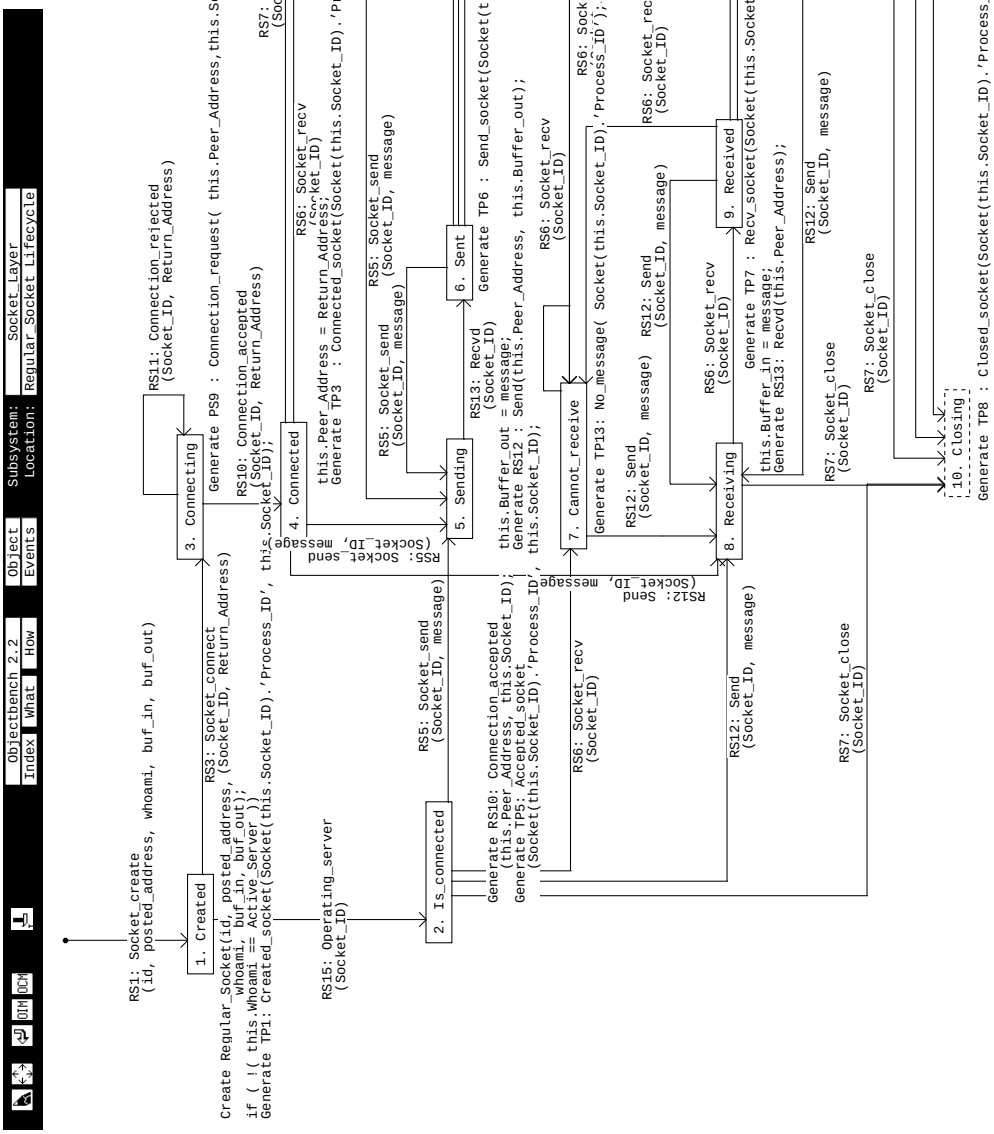

Figure 3.3: Regular Socket State Model

Connected(3) and the state Is_connected(2), allowed transitions are the same. A Regular_Socket can accept four events: *RS5: Socket_send()*, *RS6: Socket_recv*, and *RS7: Socket_close()*, generated by a Test_Process, and *RS12: Send()*, generated by its peer. A peer is a Regular_Socket on the other end of the connection as we already mentioned. The transitions between states, and actions of a Regular_Socket upon arrival in a new state are strightforward.

3.4 Building Test_Process State Model

At this point, constructed State Models for a Regular_Socket (Fig. 3.2) and a Public_Socket (Fig. 3.3) allow derivation of a Test_Process State Model, see Fig. 3.4.

When an instance of a socket is in a certain state, only a few allowable transitions can change its current state. By bringing an instance of a socket into each state, and testing all allowable transitions leading from this state, we cover all possible legal state/event combinations for our abstract model. The number of this transitions is finite, and, by executing them all, we completely test the model. To achieve complete node coverage, multiple copies of a Socket driven by multiple copies of a Test_Process are required.

In the State Model for a Test_Process, each state of the Test_Process corresponds to a certain state of a Socket. A Test_Process generates only those events which can be accepted by the Socket in a given state. (Tests of filtering of events can be implemented in a similar way to verify that each state rejects invalid events). When receiving an event from a Socket, an instance of a Test_Process changes its state, which corresponds to the new state of a Socket, and in which another finite number of events can be generated.

The method of mapping state machines executed by instances of a Test_Process can be any method allowing coverage of all possible state/event combinations. In our State Model for the Test_Process, we use the value of the attribute *Process_ID*, a

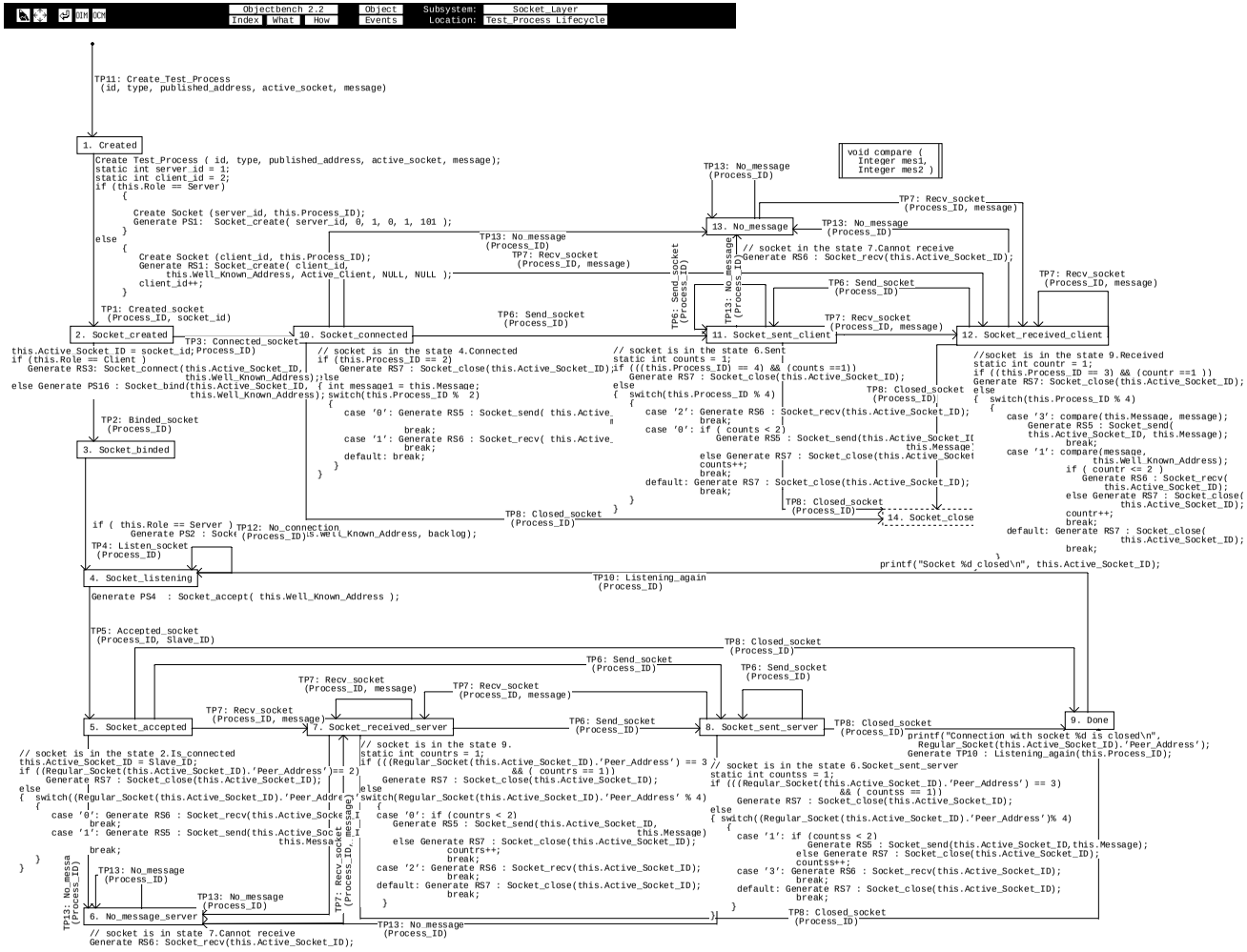

Figure 3.4: Test Process State Model

unique identifier, to map different generated events on instances of the Test_Process.

We constructed a model of a socket used for communication between two processes. Communication includes establishing a connection and an exchange of messages between processes. To test communication, at least one instance of a Test_Process (Role = Server) generating events for the Public_Socket and for the Regular_Socket (Whoami = Active_Server) is required. The State Model for the Server includes states 1 through 9 in Fig. 3.4. The State Model for the Client includes states 1, 2, and 10 through 14 in Fig. 3.4. In order to establish a connection, a Public_Socket, listening for a connection request, must be created first. This can be achieved by creating a Public_Socket, binding it to some Well_Known_Address, and then executing *listen()*, *accept()*.

Consider the state Socket_connected(10) of the Test_Process on Fig. 3.4. It corresponds to the state Connected(4) of a Regular_Socket (Whoami = Active_Client). In the state Connected(4), an Active_Client can accept three events generated by an external entity, Test_Process, and one event generated by another instance of a Regular_Socket (Whoami = Active_Server). In our model, to test the transitions of the Active_Client from the state Connected(4), a Test_Process (Process_ID = 2), generates an event *RS7: Socket_close()*, instances of the Test_Process with even Process_ID generate an event *RS5: Socket_send()*, and instances of the Test_Process with odd Process_ID generate an event *RS6: Socket_recv()*. One of the attributes of the Test_Process is an attribute Active_Socket_ID. Each Test_Process generates events to the instance of an Active_Client (Socket_ID = this.Active_Socket_ID). The keyword **this** identifies a special type of pointer to a currently active instance of an object. Any of these events causes an instance of Active_Client to change state from Connected(4) to some other state. For example, an event *RS7: Socket_close()*, brings an instance of Active_Client (Socket_ID = 2) to the state Closing(10). Upon arrival in this state, Active_Client (Socket_ID = 2) generates an event *TP8: Closed_socket()* which returns control back to the Test_Process. The instance

of the Test_Process changes its state to the terminal state Socket_closed(14), where no further events are generated.

An event *RS5: Socket_send()* forces an instance of an Active_Client to make a transition from the state Connected(4) to the state Sending(5), in which only event, *RS13: Recvd*, generated by an instance of an Active_Server, can be accepted. After accepting this event, an instance of an Active_Client arrives in the state Sent(6), and returns control to the Test_Process by generating an event *TP6: Send_socket()*.

After receiving an event *RS6: Socket_recv()*, an instance of a Regular_Socket changes its state from the state Connected(4) to the state Cannot_receive(7), where it generates an event *TP13: No_message()* to return control to the Test_Process.

At this point, we have completely covered the allowable transitions from the state Connected(4). The same approach is used to cover all states of a Regular_Socket and a Public_Socket, and to complete the state model for the Test_Process.

By instantiating multiple copies of the Test_Process object, which will instantiate and drive multiple copies of the Socket object through all possible state/event combinations, we test the Socket model for consistency and validity.

If a system was modified, and the modifications are proven to be confined to a given state or a set of states in the OOA State Model then test generation can be focused on that state or those states.

3.5 Object Communication Model

The Object Communication Model is shown on Fig. 3.5. It captures interactions between objects.

Ovals represent objects and arrows represent events sent from a source object to a recipient. Events' labels include abbreviations of recipient objects.

According to the OOA notation, events, generated and consumed by the same object, are not depicted. This is why events modeling interactions between two different instances of a Regular_Socket are not shown.

The Object Communication Model was very useful at the stage of debugging the model.

The Models of the Socket and the Test_Process Objects described in this chapter were compiled into SES/*objectbench* simulation language, and animated. It has been verified that the model of test driver correctly navigates the model of the socket through a pre-defined sequence of states.

The next step – translation of the model of the test driver into the test suite is described in the following chapter.

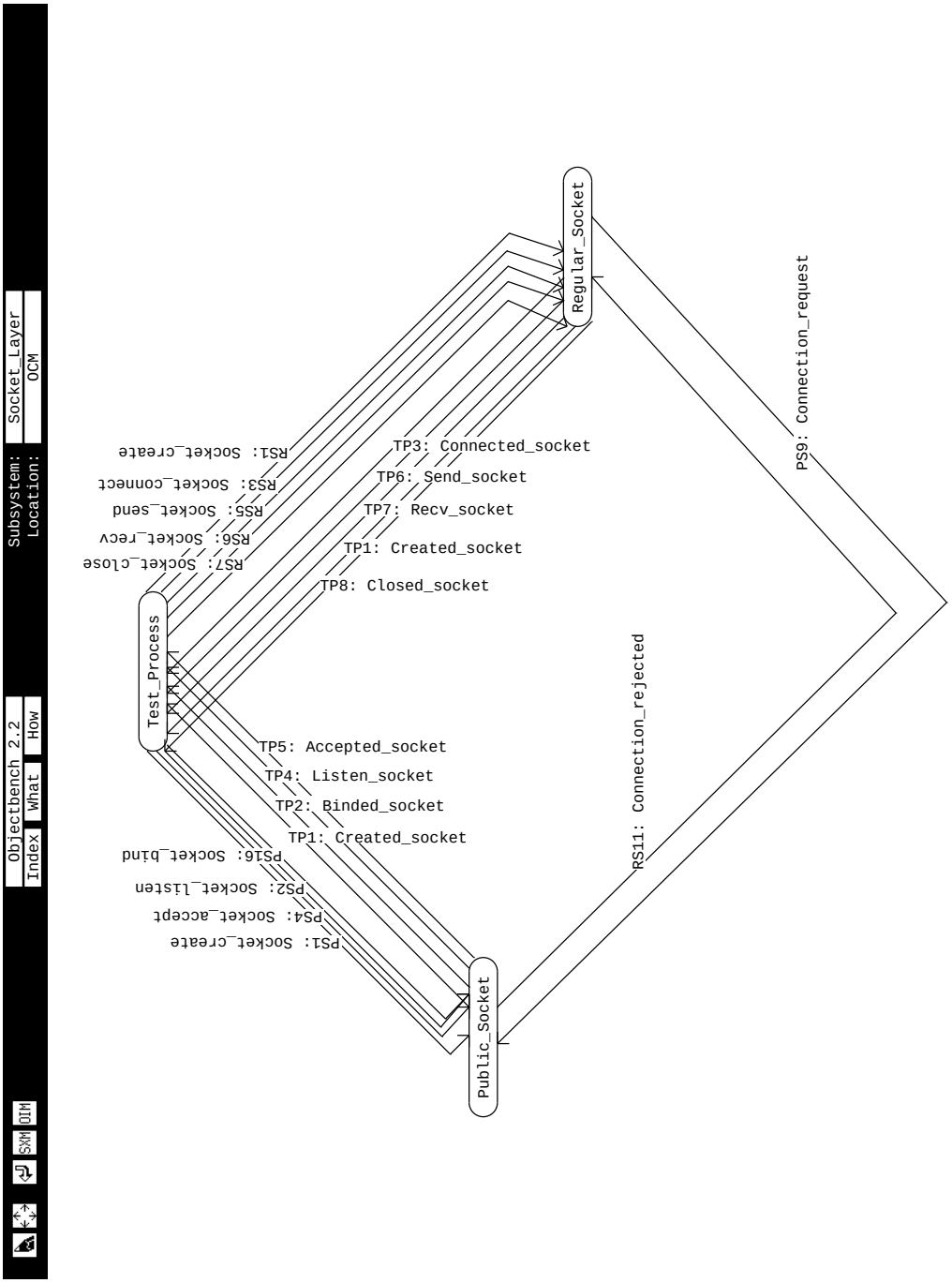

Figure 3.5: Object Communication Model

Chapter 4

Implementing a Test Suite

In this chapter the translation of a Test_Process State Model into a test suite for conectioin-oriented sockets is described. C code for the test suite and sample outputs of tests are presented in Appenix A.

Our Test_Process State Model was constructed to test the socket abstraction with the assumption that execution environment for sockets is correct: communication protocols, communication domains, addressing schemes in communication domains, buffer management, memory management.

In order to translate the model into a test program, however, we need to resolve all these matters. Useful examples of how to use network routines are presented in [14].

We translated our model for a particular case of of connection-oriented sockets created with the system call *socket(AF_INET, SOCK_STREAM, 0)*. Parameters supplied for this particular system call indicate that a socket ought to be created in the Internet communication domain, be connection-oriented (SOCK_STREAM), and use the TCP protocol - a default protocol for connection-oriented sockets in the Internet communication domain.

The following system header files define socket-related utility functions and data structures used when operating on sockets:

```
<sys/socket.h>
<sys/socketvar.h>
<netinet/in.h>
<arpa/inet.h>
```

Many of the socket calls require a pointer to a socket address structure as an argument. The definition of this structure is in

```
<sys/socket.h>
struct sockaddr {
        u_short sa_family;      /* address family */
        char sa_data[14];       /* protocol-specific address */
};
```

The contents of a protocol-specific address are interpreted according to the type of address. For the Internet family the following structures are defined in

```
<netinet/in.h>:
struct sockaddr_in {
        u_char  sin_len;
        u_char  sin_family;         /* AF_INET  */
        u_short sin_port;           /* 16-bit port number */
        struct  in_addr sin_addr;   /* 32-bit netid/hostid */
        char    sin_zero[8];        /* unused         */
};
```

To operate on the socket address structure the following standard C library functions are used:

bzero – writes the specifies number of null bytes to a specified destination.

htonl – converts an unsigned long integer from host byte order to Internet network byte order.

htons – converts an unsigned short integer from host byte order to Internet network byte order.

Address conversion routines between Internet addresses are written in dotted-decimal format XXX.XX.XXX.XXX and in_addr structure.

inet_addr – converts an Internet address into Internet numbers.

inet_ntoa – converts an Internet address into an ASCII string. C definitions and data type definitions are given in the header file

`<sys/types.h>`.

It has already been mentioned, that at least two events in our model correspond to each system call. A Test_Process generates an event to a subtype of a Socket object and waits for return of control – an event generated by a subtype of a Socket object. A system call returns a non-negative integer if successful. A correspondence between system calls and modeling events can be found in Sections 3.2 and 3.3.

After addressing and networking issues were resolved the rest of implementation of a test suite was very easy.

The scenario for testing was as follows: a server starts on foghorn.cs.utexas.edu. The network address of a server's host and a port of its Well_Known_Socket is hard-coded in "inet.h" file (see Appendix A.3). Clients from another machine request connection, send and receive messages, and quit. Each client process executes its own sequence of system calls corresponding to the possible state machines for the Test_Process State Model.

For the Socket object to cover all possible state/event combinations at least 7 different instances of the Client process must be invoked. A number is assigned to each instance of a Client process. Depending on this number a Client process executes its own sequence of system calls, corresponding to the possible state machines for the Test_Process (Role = Client) State Model. Validating checks are placed after each systen call to insure that each step is properly executed.

We are testing communication between two processes using sockets. We designed our test suite such that client and server complement each other's actions. If a client executes a *send()* system call, a server executes a *receive()*, and vice versa. Received messages are printed out. The cases when Client number 5 executes or Server serves Client number 6 are used for testing the connection – a message that was sent by one process is received by its peer, and sent back. The first process compares the message it sent against the received message and prints both messages.

Testing was performed on a network of AIX and Sun workstations at the Computer Science Department of the University of Texas at Austin.

In Appendix A.1, a C code for a client part of the test suite is given. In Appendix A.2 a C code for a server part of the test suite is given. A header file used by both programs is given in Appendix A.3. In Appendix A.4 a sample output of running the server part of the test suite is presented. In Appendix A.5 a sample output of running the client part of the test suite is presented.

Chapter 5

Conclusion and Future Plans

In this thesis we used the Shlaer/Mellor approach to the OOA methodology to develop a test suite for existing software. We used this approach to create a test suite for the Unix Sockets system calls.

We showed that that black-box testing of a system can be partitioned into a subset of small tests which cover all possible state/event combinations for the model of the system, and that there is no need to perform random or exhaustive testing.

First, a model of an existing software system (Socket) was constructed at a high level of abstraction using the software documentation.

Second, a state model for a driver (Test_Process) based upon the state model for the socket object was constructed. Each instance of a test process object generates events, driving a socket object through a pre-defined set of states.

The model was captured, animated, and the dynamic behavior of the model was verified using the SES/*objectbench* tool. Different scenarios (main driver program) were executed to validate te correctness of the Test_Process State Model.

Third, the state model for the test process was translated into C language code by replacing "create" object instances and "generate" event statements of the OOA with the appropriate calls to socket library routines, and generating loops over state variables. Appropriate tests for validation at different points in action

language programs associated with each state were added.

Fourth, a developed test suite was tested on a network of workstations.

In future we plan to create a complete test suite for sockets library system calls by adding more system calls and taking into account datagram (connectionless) sockets. We are also planning to add testing of filtering of events to verify whether a given state rejects invalid events. We would like to use the code generator CodeGenesis recently developed to use with SES/*objectbench*2.2 for automatic translation of a driver state machine into C++ language code.

Appendix A

Test Code

A.1 client_test.c

```
/*************************************************************************
 * Connection-oriented Client.
 * Communication Domain - AF_INET.
 * Communication Protocol - TCP.
 *************************************************************************/
#include <stdio.h>
int main(int argc, char *argv[])
{
   int howmany;  /* how many clients are connecting   */
   int j;
   int client(int number);
       printf("How many clients:");
       scanf("%d", &howmany);           /* get number of clients */
       printf("\n");
       for ( j = 1; j <= howmany; j++)
       {
```

```
            printf("***************************\n");
            printf("Client %d\n", j);
            client(j);                  /*   invoke a client        */
        }
       exit (0);
}

int client(int number)
/*********************************************************************
*  Client initiates connection with a server.
*  Different instances of client execute different state machines.
*********************************************************************/
{
#include        "inet.h"
#define MAXSIZE  512           /* buffer size  */

int sockfd;
struct sockaddr_in serv_addr; /* Well-Known-Socket address*/
char sendbuf[512], recvbuf[512];    /* buffers for data        */
static char message[] = "Old McDonalds had a farm";
static char message_server[] = "Frosty, the snowman, had a soul";
int msglength, msgserlength, k;       /* length of the messages */
void print_message(char *buffer, int count); /* prints buffer   */
int send_client(int sockfd, char * buffer, int message_length,
                                            char *message);
int recv_client(int sockfd, char * buffer, int message_length);
/*********************************************************************
 *  Specify Well-Known-Socket address.
```

```
 ************************************************************************/
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
serv_addr.sin_port        = htons(SERV_TCP_PORT);
/************************************************************************
 * Create a socket.
 ************************************************************************/
if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
      printf("Client: error opening stream socket\n");
else printf("Client: opened stream socket\n");

msgserlength = strlen( message_server);
msglength = strlen(message);
/************************************************************************
 * Connect to the server.
 ************************************************************************/
if (connect(sockfd, (struct sockaddr *) &serv_addr,
                      sizeof(serv_addr)) < 0)
         printf("Client: error connecting  to server\n");
else
{
   printf("Client: connected to server\n");
/************************************************************************
 * Execute state machine for a given instance of a client
 ************************************************************************/
   printf("Client: my number is %d\n", number);
   if ( number == 1) ;
```

```
   else if ( number == 2 ) recv_client(sockfd, recvbuf, msgserlength);
   else if ( number == 3 )
              send_client(sockfd, sendbuf, msglength, message);
   else if ( number ==  4) {
      for ( k = 1; k<=2; k++ )
                 recv_client(sockfd, recvbuf, msgserlength);
   }
   else if ( number == 5 ) {
         send_client(sockfd, sendbuf, msglength, message);
         recv_client(sockfd, recvbuf, msglength);
   }
   else if ( number == 6 ) {
         recv_client(sockfd, recvbuf, msgserlength);
         send_client(sockfd, sendbuf, msgserlength, recvbuf);
   }
   else if ( number == 7) {
       for ( k = 1; k<=2; k++ )
         send_client(sockfd, sendbuf, msglength, message);
   }
   else {
       for ( k = 1; k<= (number/2 ); k++)
         {
              send_client(sockfd, sendbuf, msglength, message);
              recv_client(sockfd, recvbuf, msglength);
          }
   }

/*****************************************************************
```

```
 * Close  socket.
 ************************************************************************/
        if (close(sockfd) < 0)
                  printf("Client: error closing socket\n");
          else  printf("Client:  socket closed\n");
    }
}

void print_message(char *buffer, int count)
/************************************************************************
* Prints the content of the buffer
************************************************************************/
{
  char *p = buffer;
  int i = 0;

    while ( (*p != '\0') && (i < count ))
    {
       putchar(*p);
       p++;
       i++;
     }
  putchar('\n');
}

int send_client(int sockfd, char * buffer,
       int message_length, char *message)
/************************************************************************
```

```
* Copies a message to the buffer, and sends it
*****************************************************************/
{
    void print_message(char *buffer, int count);
                if (strcpy(buffer, message) < 0)
                   printf("Client: problem copying a message\n");
                if (send(sockfd, buffer, message_length, 0) < 0 )
                   printf("Client: error sending message\n");
                else {
                   printf("Client: message sent\n");
                   print_message(buffer, message_length);
          }
}

int recv_client(int sockfd, char * buffer, int message_length)
/*****************************************************************
* Receives a message of a length message_lengt in the buffer
*****************************************************************/
{
    void print_message(char *buffer, int count);
          if ( recv(sockfd, buffer, message_length, 0) < 0 )
                         printf("Client: error receiving message\n");
          else  {
                         printf("Client: message received\n");
                         print_message(buffer, message_length);
          }
}
```

A.2 server_test.c

```
/*****************************************************************
 * Connection-oriented ITERATIVE Server.
 * Communication Domain - AF_INET.
 * Communication Protocol - TCP.
 *****************************************************************/

#include "inet.h"
#define MAXSIZE  512           /* buffer size  */
#define DEBUG

int main(int argc, char *argv[])

{
       int sockfd, newsockfd, clilen, childpid;
       struct sockaddr_in cli_addr, serv_addr;
       static char message[] = "Frosty, the snowman, had a soul";
       static char message_client[] = " Old McDonalds had a farm";

       char sendbuf[512], recvbuf[512]; /* buffers for data        */
       int msglength, msgclilength;     /*  length of the messages  */
       int counter = 1;
       int howmany, j;              /* how many clients will connect */
       void print_message(char *buffer, int count);
       int send_server(int sockfd, char * buffer, int message_length,
                                            char *message);
       int recv_server(int sockfd, char * buffer, int message_length);
```

```
       printf("How many clients:");
       scanf("%d",&howmany);
       printf("\n");
/*****************************************************************
 * Create a socket.
 *****************************************************************/
if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
printf("Server: error opening stream socket\n");
        else printf("Server: stream socket opened\n");
/*****************************************************************
 * Bind  socket to a  Well-Known-Socket address.
 *****************************************************************/
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port        = htons(SERV_TCP_PORT);
if (bind(sockfd, (struct sockaddr *) &serv_addr,
                                          sizeof(serv_addr)) < 0)
printf("Server: error binding local address\n");
         else printf("Server: stream socket binded\n");
/*****************************************************************
 *  Listen  for connections
 *****************************************************************/
       if (listen(sockfd, 5) < 0)
                 printf("Server: listen error\n");
       else printf("Server: stream socket is listening\n");
/*****************************************************************
 *  Accept a connection
```

```
 ****************************************************************/
        msgclilength = strlen(message_client);
        msglength = strlen(message);
        while ( counter <= howmany)
{  clilen = sizeof(cli_addr);
   newsockfd = accept(sockfd,
                        (struct sockaddr *) &cli_addr, &clilen);
           if (newsockfd < 0)
               printf("Server: accept error");
           else
           {
     printf("Server: stream socket accepted\n");
     printf("Counter %d\n", counter);
     if (counter == 1 )  ;
     else if (counter == 2)
          send_server(sockfd, sendbuf, msglength, message);
     else if (counter == 3)
                recv_server(newsockfd, recvbuf, msgclilength);
     else if (counter == 4) {
               for ( j = 1; j<=2; j++ )
                send_server(newsockfd, sendbuf, msglength, message);
              }
     else if (counter == 5)
              {
                recv_server(newsockfd, recvbuf, msgclilength);
                send_server(newsockfd, sendbuf, msglength, recvbuf);
               }
              else if (counter == 6)
```

```
                {
                  send_server(newsockfd, sendbuf, msglength, message);
                  recv_server(newsockfd, recvbuf, msglength);
                 }
        else if (counter == 7) {
          for ( j = 1; j<=2; j++ )
             recv_server(newsockfd, recvbuf, msgclilength);
                }
                else {
          for ( j = 1; j<= (counter/ 2); j++)
                   {
             recv_server(newsockfd, recvbuf, msgclilength);
             send_server(newsockfd, sendbuf, msgclilength, recvbuf);
                    }
                  }

               counter++;
        }
/*****************************************************************
 *  Close socket.
 *****************************************************************/

        close(newsockfd);
                printf("Server: stream socket closed\n");
                    /* end else statement */
}
     close(sockfd);
     exit (0);
```

```
}

void print_message(char *buffer, int count)
/*********************************************************************
* Prints the content of the buffer
*********************************************************************/
{
    char *p = buffer;
    int i = 0;

    while ( (*p != '\0') && (i < count ))
        {
          putchar(*p);
          p++;
          i++;
        }
   putchar('\n');
}


int send_server(int sockfd, char * buffer,
        int message_length, char *message)
/*********************************************************************
* Copies a message to the buffer, and sends it
*********************************************************************/
{
    void print_message(char *buffer, int count);
```

```
                if (strcpy(buffer, message) < 0)
                  printf("Server: problem copying a message\n");
                if (send(sockfd, buffer, message_length, 0) < 0 )
                  printf("Server: error sending message\n");
                else {
                  printf("Server: message sent\n");
                  print_message(buffer, message_length);
                }
}

int recv_server(int sockfd, char * buffer, int message_length)
/*****************************************************************
* Receives a message of a length message_lengt in the buffer
*****************************************************************/
{
    void print_message(char *buffer, int count);

                if ( recv(sockfd, buffer, message_length, 0) < 0 )
                   printf("Server: error receiving message\n");
                else  {
                   printf("Server: message received\n");
                   print_message(buffer, message_length);
                }
}
```

A.3 inet.h

```
/*************************************************************
 * Definitions are taken from [14444].
 *************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERV_TCP_PORT 4001          /* TCP port  */
#define       SERV_HOST_ADDR  "128.83.143.205"
     /* host addr for server, foghorn.cs.utexas.edu */
```

A.4 client_run

```
owl% script client_run
Script started, file is client_run
% cli_test
How many clients:8

***************************
Client 1
Client: opened stream socket
Client: connected to server
Client: my number is 1
Client:  socket closed
***************************
Client 2
Client: opened stream socket
Client: connected to server
Client: my number is 2
Client: message received
Frosty, the snowman, had a soul
Client:  socket closed
***************************
Client 3
Client: opened stream socket
Client: connected to server
Client: my number is 3
Client: message sent
Client:  socket closed
***************************
```

```
Client 4
Client: opened stream socket
Client: connected to server
Client: my number is 4
Client: message received
Frosty, the snowman, had a soul
Client: message received
Frosty, the snowman, had a soul
Client:  socket closed
**************************
Client 5
Client: opened stream socket
Client: connected to server
Client: my number is 5
Client: message sent
Old McDonalds had a farm
Client: message received
Old McDonalds had a farm
Client:  socket closed
**************************
Client 6
Client: opened stream socket
Client: connected to server
Client: my number is 6
Client: message received
Frosty, the snowman, had a soul
Client: message sent
Frosty, the snowman, had a soul
```

```
Client:  socket closed
**************************
Client 7
Client: opened stream socket
Client: connected to server
Client: my number is 7
Client: message sent
Old McDonalds had a farm
Client: message sent
Old McDonalds had a farm
Client:  socket closed
**************************
Client 8
Client: opened stream socket
Client: connected to server
Client: my number is 8
Client: message sent
Old McDonalds had a farm
Client: message received
Old McDonalds had a farm
Client: message sent
Old McDonalds had a farm
Client: message received
Old McDonalds had a farm
Client: message sent
Old McDonalds had a farm
Client: message received
Old McDonalds had a farm
```

```
Client: message sent
Old McDonalds had a farm
Client: message received
Old McDonalds had a farm
Client:  socket closed
%
script done on Mon Apr  8 17:24:41 1996
```

A.5 server_run

```
foghorn% script server_run
Script started, file is server_run
% ser_test
How many clients:8

Server: stream socket opened
Server: stream socket binded
Server: stream socket is listening
Server: stream socket accepted
Counter 1
Server: stream socket closed
Server: stream socket accepted
Counter 2
Server: message sent
Server: stream socket closed
Server: stream socket accepted
Counter 3
Server: message received
Old McDonalds had a farm
Server: stream socket closed
Server: stream socket accepted
Counter 4
Server: message sent
Server: message sent
Server: stream socket closed
Server: stream socket accepted
Counter 5
```

```
Server: message received
Server: message sent
Old McDonalds had a farm
Server: stream socket closed
Server: stream socket accepted
Counter 6
Server: message sent
Frosty, the snowman, had a soul
Server: message received
Frosty, the snowman, had a soul
Server: stream socket closed
Server: stream socket accepted
Counter 7
Server: message received
Old McDonalds had a farm
Server: message received
Old McDonalds had a farm
Server: stream socket closed
Server: stream socket accepted
Counter 8
Server: message received
Old McDOld McDonalds had a farm
Server: stream socket closed
Server: stream socket accepted
Counter 8
Server: message received
Old McDonalds had a farm
Server: message sent
```

```
Old McDonalds had a farm
Server: message received
Old McDonalds had a farm
Server: message sent
Old McDonalds had a farm
Server: message received
Old McDonalds had a farm
Server: message sent
Old McDonalds had a farm
Server: message received
Old McDonalds had a farm
Server: message sent
Old McDonalds had a farm
Server: stream socket closed
% Script done, file is server_run
```

Bibliography

[1] *SES/objectbench User's Guide.* 1993.

[2] Boris Beizer. *Black-Box Testing. Techniques for Functional Testing of Software and Systems.* Wiley, 1992.

[3] Grady Gooch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.

[4] Infotech. *Software Testing.* Maidenhead, 1979.

[5] Cem Kaner. *Testing Computer Software.* Blue Rigde Sum, 1988.

[6] Edward Kit. *Software Testing in the Real World: Improwing the Process.* Wokingham, 1995.

[7] Samuel Leffler. *4.3 BSD UNIX Operating System.* Addison-Wesley, 1988.

[8] Clenford Myers. *The Art of Software Testing.* New York, 1979.

[9] William Perry. *Effective Methods for Software Testing.* New York, 1995.

[10] Thomas Royer. *Software Testing Management: Life on the Critical Path.* Englewood, 1993.

[11] James Rumbaugh. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[12] Sally Shlaer and Steven Mellor. *Object-Oriented Systems Analysis. Modeling the World in Data.* Prentice-Hall, 1988.

[13] Sally Shlaer and Steven Mellor. *Object Lifecycles. Modeling the World in States.* Prentice-Hall, 1992.

[14] Richard Stevens. *UNIX Network Programming.* Prentice-Hall, 1990.

[15] Pavan Vohra. *Software Testing with a Test Data Generation Tool.* THESIS, 1987.

Vita

Almadena Yurevna Chtchelkanova was born in Tashkent, USSR on March 11, 1962, the daughter of Svetlana Y. Chtchelkanova and Yury A. Chtchelkanov. After completing her work at 187 High School, Tashkent, USSR, in 1978, she entered Moscow Lomonosov State University in Moscow, USSR. She received her degree of Master of Science in Astronomy, in January, 1984, and her Ph.D. in Physics and Mathematics, in November, 1988 from Moscow Lomonosov State University.

During 1987 - 1991 she was employed as a Research Scientist in the Intstitute of Scientific and Technical Information of Academy of Sciences in Moscow, USSR. She published 12 papers in russian astronomical journals. In January, 1995, she entered The Graduate School at The University of Texas.

Permanent Address: 3355-D Lake Austin Blvd.
Austin, TX 78703

This thesis was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.