Using SeSFJava in Teaching Introductory Network Courses

Tamer Elsharnouby
Department of Computer Science,
University of Maryland,
College Park, MD 20742
sharno@cs.umd.edu

A. Udaya Shankar
Department of Computer Science,
University of Maryland,
College Park, MD 20742
shankar@cs.umd.edu

ABSTRACT

Networking course projects are usually described by an informal specification and a collection of test cases. Students often misunderstand the specification or oversimplify it to fit just the test cases. Using formal methods eliminates these misunderstandings and allows the students to test their projects thoroughly, but at the expense of learning a new language. SeSF (Services and Systems Framework) is one way to overcome this obstacle. In SeSF, both implementations and services are defined by programs in conventional languages, thereby, eliminating the need to teach the students a new language. SeSF is a markup language that can be integrated with any conventional language. The integration of SeSF and Java is called SeSFJava. SeSFJava provides a technique to mechanically test whether student projects conform to their corresponding specifications, thereby, providing the instructors with a technique for semi-automated grading.

We present a four-phase transport protocol project, and describe how SeSFJava is used in specifying, testing and grading the different phases of this project. The use of SeSF significantly (1) increased the percentage of students who completed the projects, (2) reduced their email queries about the specification, and (3) reduced the grading time.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; D.2.5 [**Software Engineering**]: Testing And Debugging—*distributed debugging, testing tools*; F.4 [**Theory Of Computation**]: Mathematical Logic And Formal Languages

General Terms: Design, Verification.

1. INTRODUCTION

The goal of the programming assignments of the introductory networking senior-level course at University of Maryland is to teach the students the following:

- The role of network protocols.
- The different roles of the layers of the network and how they stack above each other.
- How to enhance the performance of the network in the face of changing network conditions.
- How to implement a distributed multi-threading applications, for example, client-server or peer-to-peer applications.

In fall 1999, we introduced a three-phase project that takes the above goals into account. The project was to implement client and server TCP sockets. Phase I implements a data transfer protocol. Phase II implements congestion control in order to enhance the performance of the data transfer protocol. Phase III implements the connection management and the two-way data transfer protocols of TCP/IP. All project specifications were described informally, and test cases were provided.

During the course, a number of problems emerged. Some students misunderstood the specification or oversimplified it to just fit the test cases provided with the project assignment. Other students did not test their projects thoroughly with various inputs. Others did not finish the project because they did not budget enough time, especially in phase III which involved much more work than the other two phases.

The teaching assistants (TAs) spent excessive time in testing and grading the student projects.

These problems prompted us to integrate formal methods into the networks course. Formal methods, in theory, removes all misunderstandings about the project specifications. It provides techniques to test the projects extensively, which helps the students to discover more bugs. It permits division of the project into more phases, for example, the third phase can be divided into two phases: one that implements connection management, and another that puts everything together. Formal methods also provides a testing harness on the actual platform.

But on the other hand, formal methods is not without drawbacks. One drawback is that students have to learn a new formal language, which paves the way to more misunderstandings by the students, because of their lack of expertise. Another drawback is that they have to learn new techniques to test their implementation against the project specifications. All these have to be learned under the tight time constraints of the semester.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'05, February 23–27, 2005, St. Louis, Missouri, USA.
Copyright 2005 ACM 1-58113-997-7/05/0002 ...$5.00.

To overcome these drawbacks, we used a framework that we have developed, called SeSF (**Se**rvices and **S**ystems **F**ramework), that (1) allows definitions in conventional languages of implementations and services of distributed systems, (2) formalizes the notion of an implementation satisfying its services, and (3) provides a means for mechanical testing [9]. SeSF is an imperative, or procedural, version of the formalism in [4]. The main difference between SeSF and most other formalisms [4–7], is that SeSF stays close to the programming world.

The remainder of the paper is organized as follows. Section 2 describes an overview of SeSF. Section 3 describes an overview of SeSFJava and SeSFJava Harness. Section 4 introduces the network project. Section 5 describes our experience with the students. Section 6 concludes.

2. SESF OVERVIEW

Like most formalisms, SeSF provides a **compositional methodology** for the design and implementation of concurrent systems. Compositionality means that the design and implementation of a concurrent system can be broken up into the design and implementation of component concurrent systems. We refer to implementations as **systems** and external behavior specifications as **services**. In SeSF, both systems and services are specified by programs in conventional programming languages.

A system specification is intended for execution. Hence, its programs must satisfy the computational, synchronization, and other constraints of the underlying platform – for example, accounting for whether the platform has a single processor, a multi-processor with shared memory, or a set of loosely-coupled message-passing processors.

The service specification states all (and only) the desired properties of the system's execution, unencumbered by internal structure and computational, implementation and synchronization issues. In most formalisms, the service defines the permissible interactions between the system and its environment. However, our interest is in **layered compositionality**. Here, a composite system consists of layers of component systems, and services define the allowed sequences of interactions between layers.

Roughly speaking, a system **satisfies** its services above and below if the interactions it initiates are allowed by the services, *assuming* the interactions initiated by the system's environment are allowed by the services. Our **compositionality property** is that, given a composite system consisting of layers of component systems with services in between, if every component system in isolation satisfies its services, then the composite system as a whole satisfies its services.

Because services are defined by conventional programming languages, they are **executable**. The adoption of executable services, in general and in SeSF in particular, has the following consequences. First, the notion of a system satisfying a service is equivalent to the composite program of the system and service satisfying certain correctness properties. Second, developers can *test* a concurrent system against its service simply by executing the composite program of the system and the service, and checking whether those properties are satisfied.

Using conventional languages for specifying services, instead of a high-level specification language, has certain advantages and disadvantages. One advantage is that the service specification language is familiar to programmers, perhaps even the same language as that of implementation. This reduces the possibility of the service specification being misunderstood by implementors. Another advantage is that it allows actual implementations to be tested, rather than an abstract model. The main disadvantage is that most programming languages suffer from inconsistencies and ambiguities, and one has to avoid such constructs in service specifications. For example, Java has an ambiguous memory model, and different Java implementations have different memory models.

3. SESFJAVA OVERVIEW

SeSF is a markup language that can be integrated with any programming language. **SeSFJava** [3] is the integration of SeSF with Java. Java is chosen because of its relatively precise semantics, popularity, and built-in concurrency constructs. A SeSFJava program is a Java program with SeSF tags inserted as Java comments. Hence, a SeSFJava program can be compiled and executed as a Java program. But because of the SeSF tags, it can also be tested. We have developed a testing harness, called **SeSFJava Harness**, that can execute a distributed system of SeSFJava programs and check whether the resulting execution satisfies the relevant services and any other desired correctness assertions (also specified in SeSFJava).

SeSFJava Harness is able to handle general Java programs (e.g., parameterized unbounded-state programs) and general safety and progress assertions (e.g., parameterized invariant and leads-to assertions). It tests the implementation on the actual final platform, without altering the program to run on a simplified platform (e.g., over TCP/IP network sockets rather than a thread-based emulation). It helps the programmer check systems during the development phase; we are not concerned with black-box testing.

To test a system against its service, we execute the services and the systems together with a harness process. The harness process runs on an arbitrary machine and ensures that only one thread in the distributed system is proceeding at any time. This approach is rather conservative (because it prevents parallel execution of processes). However, it is simple and provides the global snapshots needed to check the assertions.

To participate in the testing, the system and service programs need to be instrumented, and there is a SeSFJava Preprocessor for this purpose [2, 3]. But in the case of the networking projects, we gave the students preprocessed code to manually insert in their system programs, thereby relieving them of the preprocessing hassle.

4. PROJECT OVERVIEW

The goal of the project is to build a full-fledged transport protocol between a client entity and a server entity over unreliable channels that (exactly like IP) can lose, reorder and duplicate messages in transit subject to a maximum message lifetime. The transport service [8] consists of connection management between the client and server entities augmented with reliable two-way flow-controlled data transfer. A reliable data transfer from a source to a sink ensures that data is delivered in the same sequence it was sent and without loss.

The project is divided into four phases. Each phase is independently tested for correctness.

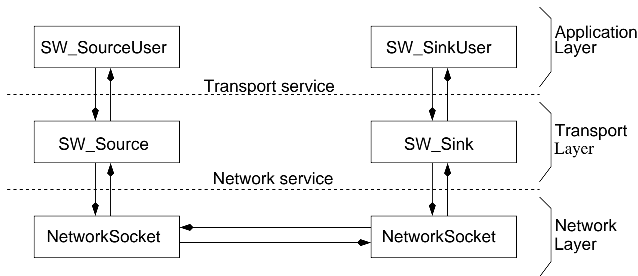

Figure 1: Phase I Overview

4.1 Phase I: Data Transfer Protocol (Correctness)

In this phase, the student implements a protocol that achieves reliable data transfer over unreliable network channels. Specifically, the project consists of two interacting programs, a Source and a Sink, as shown in fig. 1. The Source consists of three components: SW_SourceUser, SW_Source and NetworkSocket. The Sink consists of three components: SW_SinkUser, SW_Sink and NetworkSocket. SW_SourceUser passes data to SW_Source. SW_Source buffers the data (in a send window) and transfers it to SW_Sink, resending until it is acknowledged by SW_Sink. SW_Sink buffers data received out of sequence (in a receive window) and delivers data in sequence to SW_SinkUser.

The students are provided with:

- The applications, SW_SourceUser and SW_SinkUser, which transfer a file from the source to the sink.

- The NetworkSocket entity which provides the unreliable channels to be used by the transport entities. NetworkSocket entity is a wrapper to the standard sockets. It is used instead of the usual UDP sockets, because in a LAN environment, the standard sockets display hardly any loss, reordering or duplication. The students can change the probabilities of loss, reordering and duplication on the fly, which is important for testing.

- The SeSFJava Harness module and the data transfer service specification, which defines the signature of the interactions between the layers, as well as the permissible sequences of these interactions (e.g., the data sequence delivered to SW_Sink must be a prefix of the data sequence accepted from SW_Source).

The students are to implement SW_Source and SW_Sink so that they conform to the provided data transfer service. The students are free to choose the particulars of the design, including message types and formats, sequence number space, data block size, retransmission policy, acknowledgment (cumulative and/or selective) policy, round-trip time estimator, etc.

4.1.1 Testing Phase I

The data transfer service and the SeSFJava Harness are illustrated in file Harness.java (fig. 3). For full version of the harness, see the class homepage [1]. This file consists of the following parts:

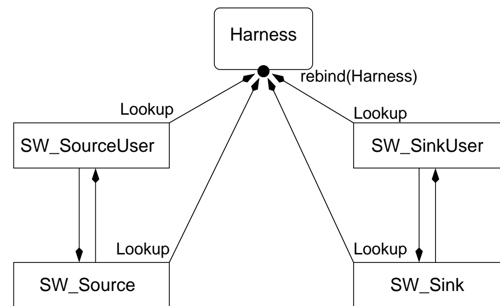

Figure 2: Phase I Harness Outline

- The main method which binds an instance of the Harness to Remote Method Invocation (RMI) port "Harness".

- Lock and unlock methods for the Harness main lock, for synchronizing the programs and threads of the project. When a thread acquires the main lock, no other thread in the network system can proceed, until the lock is released. This allows a global snapshot of the network system to be collected at the Harness and evaluated against the service. (This is a simplified version of the general SeSFJava Harness.)

- Methods that represent the interactions between the transport layer and the application layer. There are three of them: sendData and ackData on the source side, and deliverData on the sink side.

- Invariants of the data transfer protocol, for example, the number of bytes delivered to Sink's user cannot exceed the number of bytes sent by Source's user.

After developing the source and sink entities, the student connects the distributed system (SW_SourceUser, SW_Source, SW_SinkUser and SW_Sink) to the Harness, as shown in fig. 2. Initially, the student inserts, in the constructor of each network entity, an RMI "lookup" call for the Harness RMI port. For each of the interactions mentioned above, the student inserts code to obtain the Harness lock and issue an RMI call in the corresponding method in class Harness.

For example, SW_Source.sendData method after the student insertion is as follows:

```
// Inside SW_Source.java
void sendData (byte []data) throws Exception {
    harness.lock();              // obtain Harness main lock
    harness.sendData(data);      // RMI call of Harness method
                                 // with same parameters
    . . .                        // sendData method body
    harness.unlock();            // release Harness main lock
}
```

Consequently, a student can determine the correctness of both source and sink sides by checking that no errors were thrown during the execution of Harness. (To detect deadlocks, we add an extra condition: a file sent by the source has to be received.)

The program is executed as follows: (1) Execute the Harness module, so it can bind to port "Harness", (2) Execute the sink side so it can hook to the Harness class, (3) Execute

the source side to start sending the file. A log file is recorded for every execution.

4.1.2 *Grading Phase I*

The TAs grade the data in a semi-mechanical way. They run scripts to execute the projects with different input files and different network conditions. Each execution is stored in a different log file, which is checked for thrown errors. If there is an error, the TA checks the log file to print out the trace that has generated this error, and determines the grade accordingly. The student can resort to a very simple solution, say a send window size of 1, but they will then suffer in Phase II.

4.2 Phase II: Data Transfer Protocol (Performance)

This phase emphasizes the protocol's performance; that is, the grade is primarily based on the throughput achieved under varying network conditions, which in turn depends on how well the protocol adapts to congestion, the overhead of the congestion control mechanism, etc.

The students strip the RMI calls inserted in Phase I, and enhance their code to perform better. Enhancements are of two kinds: (1) network optimizations, for example, adding Tahoe congestion control, and (2) code optimizations, for example, reducing the thread-switching in their code. In this phase, the NetworkSocket has the ability to play scenarios that emulate real-life network traffic. Thus, the students can view how their code performs under various conditions.

The TAs grade this project by running scripts that execute the students projects a number of times for every test scenario, and record the throughput for each run. The average throughput is computed and the students are classified according to the performance into four groups, from fast to slow, and the grade is determined accordingly.

4.3 Phase III: Connection Protocol

In this phase, the students build a connection management protocol over unreliable network channels. The grade in this phase is primarily based on the protocol's correctness. Specifically, the project consists of two interacting programs, a Client and a Server, as shown in fig. 4. Client consists of three components: ClientUser, CM_Client and NetworkSocket. Server consists of three components: ServerUser, CM_Server and NetworkSocket.

The students are to implement CM_Client and CM_Server which are the transport entities at the two ends. They are provided with the other entities. ClientUser and ServerUser are the users of the transport entities. These applications open and close hundreds of connections under different circumstances. The pair of NetworkSockets are as in phases I and II. The specifications formally describe the three-way handshaking connection establishment, and the two-way handshaking of the disconnection procedure. Similar to that of phase I, the service specifications and the Harness are provided in Harness file, and the network system is constructed as in fig. 5, The testing and grading are carried out similarly to that of phase I. Because of limited space, we will not describe the methods in detail.

4.4 Phase IV: Putting It All Together

In this phase, the students build a full-fledged transport service over unreliable network channels, specifically, com-

```
import . . .;
class Harness  extends UnicastRemoteObject implements HarnessInterace{
  // HarnessInterface contains the headers of all the methods defined
  // in this file except methods correctData and checkAssertions.
  . . .
  Harness() throws RemoteException {super();}
  public static void main(String args[]) throws Exception {
    . . .
    Naming.rebind("Harness", new Harness());
  }

  public void lock () throws RemoteException   { . . . }
  public void unlock () throws RemoteException { . . . }

  // Source entity variables.
  ByteArrayOutputStream srcHist = new ByteArrayOutputStream ();
  long srcBufSize = 32*1024;
  int  srcBufUsed;
  long srcNumSent, srcNumAcked;
  long sinkNumDelivered;    // Sink entity variable

  //  Sends data from source user to source entity to deliver it to
  //  remote user.
  //  Called by SW_Source.sendData
  public void sendData(byte []data) throws RemoteException {
    synchronized (lock){
      checkAssertions();
      if (srcBufUsed + data.length <= srcBufSize && data.length > 0){
        srcHist.write (data, 0, data.length);
        srcNumSent + = data.length ;
        srcBufUsed + = data.length ;
      } else System.out.println("usr sendDataSource failed ");
    }
  }

  //  Notifies user that n bytes have been acked.
  //  Called by SW_SourceUser.ackData
  public void ackData(int n) throws RemoteException {
    synchronized(lock){
      checkAssertions();
      if (srcNumAcked + n  <=srcNumSent ){
        srcBufUsed = srcBufUsed − n ;
        srcNumAcked = srcNumAcked + n ;
      } else System.out.println("usr ackedData failed ");
    }
  }

  //  Delivers "data" received to entity user.
  //  Called by SW_SinkUser.deliverData
  public void deliverData(byte []data) throws RemoteException {
    synchronized(lock){
      checkAssertions();
      if (sinkNumDelivered + data.length <= srcNumSent &&
          data.length > 0 && correctData(data))
        sinkNumDelivered = sinkNumDelivered + data.length ;
      else System.out.println("usr deliverData failed ");
    }
  }

  boolean correctData (byte []data) {
    byte srcData[] = srcHist.toByteArray();
    for (int i = 0; i < data.length; i++)
      if (srcData [((int) sinkNumDelivered) + i] != data[i])
        return false ;
    return true ;
  }

  // ASSERTIONS section
  void checkAssertions() throws RemoteException{
      if (!((srcBufUsed >= 0) && (srcBufUsed <= srcBufSize)))
        System.out.println(":bufCondition(false)");
  }
}
```

Figure 3: Harness program (file Harness.java)

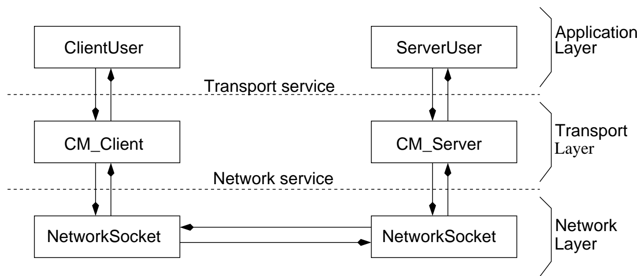

Figure 4: Phase III Overview

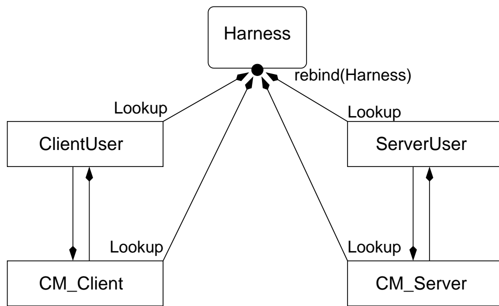

Figure 5: Phase III Harness Outline

bining phases II and III (after stripping the RMI calls). The grade of this project is based on the correctness and the performance of the students' implementations.

5. EXPERIENCE WITH THE STUDENTS

We have been using SeSFJava in the senior-level undergraduate computer networks course for the past three years. The projects are mandatory: no student can pass the course without passing the projects. The average number of students per class is 50. Most students have not been exposed to formal methods before taking this course.

Using SeSFJava significantly improves the performance of the students. Table 1 compares the use of detailed informal description of the projects (without SeSF) against the use of SeSF in specifying these projects. The number of students who completed all the phases of their projects almost doubled. Their questions about the specifications decreased by 40%. The student drop rate decreased by almost half.

	Without SeSF	With SeSF	Improv.
% of students who completed their projects	45%	88%	95%
# of email queries per students	16	10	60%
% of students dropping the class	27%	14%	93%

Table 1: Improvement using SeSFJava

From the TA perspective, using SeSF reduces the grading time per student, because considerable amount of the grading is carried mechanically. The number of regrading requests fell by 60%. We think this is because a student can test his/her implementation against the project specification, and because the TA provides the student with the trace demonstrating any errors (and thus grade penalties).

6. CONCLUSION

This work presented a design for an incremental transport protocol project. We integrated formal methods, SeSFJava in particular, into this project to achieve several goals. First, misunderstandings of the project specifications are almost entirely eliminated, because the specifications are presented formally in a language that is familiar to the students. Second, we are able to test and grade each phase thoroughly and independently, which results in better management of time for the students and the TAs. Third, the students and the TAs are able to mechanically test whether the project implementations conform to the specifications.

SeSFJava and the Harness have been used in computer network courses [1, 10]. It is not limited to networking projects, and can be used in introductory concurrent programming courses.

Acknowledgment

This work was supported in part by the Maryland Information and Network Dynamics (MIND) Laboratory, Fujitsu Laboratories of America, and by the Department of Defense through a University of Maryland Institute for Advanced Computer Studies (UMIACS) contract.

7. REFERENCES

[1] T. Elsharnouby. Class homepage on computer networks (CMSC417) at University of Maryland, 2003. http://www.cs.umd.edu/class/spring2003/cmsc417/.

[2] T. Elsharnouby and A. U. Shankar. SeSFJava: A framework for design and testing of concurrent systems. Technical Report CS-TR 4619, UMIACS-TR 2004-61, University of Maryland, 2004.

[3] T. Elsharnouby and A. U. Shankar. SeSFJava harness: Service and assertion checking for protocol implementations. *IEEE Journal on Selected Areas in Communucations*, 2004.

[4] S. S. Lam and A. Shankar. A theory of interfaces and modules I − composition theorem. *IEEE Transactions on Software Engineering*, 20(1):55–71, January 1994.

[5] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., August 1987.

[6] J. Misra. *A Discipline of Multiprogramming*. Springer-Verlag, 2001.

[7] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, 1998.

[8] A. U. Shankar. Transport layer principles. published in The Communication Handbook, CRC Press, 1996.

[9] A. U. Shankar. *Concurrent Systems and Services: Design, Verification and Testing*. in preparation, 2005.

[10] Y. Yang. Class homepage on computer networks (CS433) at Yale University, 2003. http://zoo.cs.yale.edu/classes/cs433/assignments/prog1.